

Programming OpenMP

Christian Terboven

Michael Klemm



Agenda



	Day 1	Day 2	Day 3
09:00- 10:30 CET		Tasking 1 • Tasking Intro • Lab 1	GPUs OpenMP for Compute Accelerators
10:45- 12:15 CET		Tasking 2 Taskloop Dependencies Cancellation Lab 2 	 Tools for Perf. and Correctness VI-HPS Tools for Performance VI-HPS Tools for Correctness
13:00- 14:45 CET	Introduction to OpenMP	 Host Perf.: SIMD Vectorisation Lab 3 	MISC. OpenMP 5.0 Features DOACROSS Loops
15:00- 16:00 CET	Hands-on: Introduction to OpenMP	 Host Perf.: NUMA Memory Access Task Affinity Memory Management Lab 4 	Roadmap / Outlook Open Discussion OpenMP 5.0 and beyond End: approx. 16:30 CET

OpenMP Tutorial Members of the OpenMP Language Committee



Programming OpenMP

An Overview Of OpenMP

Christian Terboven

Michael Klemm





History

- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1
- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5
- 11/2018: OpenMP 5.0
- 11/2020: OpenMP 5.1





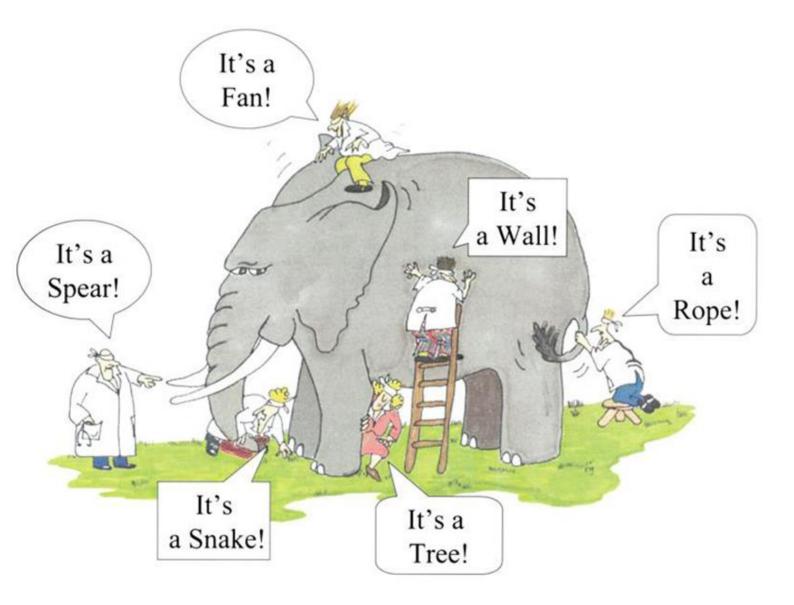
What is OpenMP?

- Parallel Region & Worksharing
- Tasking

•

...

- SIMD / Vectorization
- Accelerator Programming





Get your C/C++ and Fortran Reference Guide! Covers all of OpenMP 5.1!

penMP API 5.1	www.openmp.org	Page
	OpenMP 5.1 API Syntax Referen	ce Guide
OpenMP openmp.org	The OpenMP* API is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications in C/C++ and	Fortran. OpenMP is suitable for a wide range of algorithms running on multicore nodes and chips, NUMA systems, GPUs, and other such devices attached to a CPU.
G/C++ content May be abbreviated to For	Functionality new/changed in OpenMP 5.1 is this o Functionality new/changed in OpenMP 5.0 in this o	
Directives and Constructs		
An OpenMP executable directive applies to the succeeding the top and a single exit at the bottom. OpenMP directive	g structured block. A structured-block is an OpenMP construct except simd and any declarative directive may not appear in	t or a block of executable statements with a single entry at Fortran PURE procedures.
Variant directives	Informational and utility directives requires (25.3) (2.4) Specifies the features that an implementation must provide	error (2.5.4) Instructs the compiler or runtime to display a message and to perform an error action.
ne of which may be conditionally selected to replace the netadirective based on the enclosing OpenMP context.	in order for the code to compile and to execute correctly.	#pragma omp error (clouse / (,] clouse))
Apragma omp metadirective (clause[(,) clause])	#pragma omp requires clause [[[,] clause]]	15omp error (clouse [],] clouse]]
Apragma omp begin metadirective (clause! (,) clause)) somt(s) Apragma omp end metadirective ISomp metadirective [clause! (,) clause)] or.	ISomp requires clause [{ { , } clause] } clause: reverse_offload unified_started, memory	clouse: at(compilation execution) seventy(fatal waming) message(may string)
ISomp begin metadirective [cloure] [,] cloure]] strat(s) ISomp end metadirective puse: when {context-selector-specification: [directive-variant]]	atomic default, men unversion defined requirement dynamic allocators ext. Implementation defined requirement assumes and assume (25.2)	parallel construct parallel (2.6) (2.6) Creates a team of OpenMP threads that execute the region.
default ((directive-vorient)) eclare variant (2.3.5) eclares a specialized variant of a base function and the intext in which it is used.	Provides invariants to the implementation that may be used for optimization purposes.	Bragma omp parallel (clouse/ []cloure]] structured-block Some parallel (clouse/ []cloure]] loosely-structured-block
Hpragma cmp declare variant(variant/surieit) \ closes (((,) closes)) (#pragma orme declare variant(variant func-id) \ closes ((,) closes))	Apragma omp begin assumes clouse [[[,] clouse]] declaration deficition seq #pragma omp end assumes . or - #pragma omp assume clouse [] [,] clouse]]	Somp and parallel - or - Somp parallel (clouse([,,]cinuse]) strictly-structured block /15 omp end parallel
function definition or declaration or - or - figurage a comp declare variant clause declaration-definition-seq #paragema comp end declare variant	structured-block Somp assumes clouse [[[,] clouse]] or - Isomp assume clouse [[[,] clouse]]	clouse: default (doto-shoring-attribute) private (list) Instprivate (list) shared (list) copyin (list) reduction (/reduction-modifier, / reduction-identifier : //o proc, bind (primary moster (doprecated) close sprea allocatir (ioliocator:)(rot)
ISomp declare variant (/base-prac.nome : / & variant-prac.nome) clause [[[,] clause]] ause: match (context-selector-specification)	loosely-structured block IS omp end assume - ar - IS omp assume clause [[[,] clause]] strictly-structured block	CC++#f (parallel :) solar-expression) CC++mm_threads (integer-expression) Fmr_Hf (parallel :) solar-integer-expression) Fmr_mm_threads (scalar-integer-expression)
adjust: args (odjust op: orgument Mit) append_args (oppend op/[, oppend op]) djust op: nothing, need_device_ptr gpend op: interop (interop-type [, interop-type]])	[15 charge and assume] clause: assumption-clause exc_inplementation-defined-requirement	teams construct teams [2:3][2:3] Creates a league of initial teams where the initial thread
control of the second secon	<pre>ossumption-close: absent[directive-name [, directive-name]]) contains[directive-name [, directive-name]]) no_openmp_routines</pre>	each team executes the region.
ispatch (2.3.6) introfs whether variant substitution occurs for a given call.	no_parallelism Cho-holds(scalar-expression)	loosely-structured-block ISomp end teams - 0r -
<pre>#pragma omp dispatch (clouse [[,] clouse]] expression atmt</pre>	nothing (2.5.3)	²⁸ ISomp teams [cknuse] (,)chuse]] strictly-structured-block [ISomp ond teams]
ISomp dispatch (clouse ([,] clouse)) strut	Indicates explicitly that the intent is to have no effect.	chove: private (list) firstarivate (list)
depend (dispend modifier, / dependence-type : locator-list) movait is_device_prof.(int) =device_integr-expression) =novariantif_conter-expression) =nocentestif_conter-expression)	Isomp nothing	thistprivate (iii) shared (bicebar); if inf) eduction ((default, / induction-identifier: list) allocate (observor:) (inf) default (inter-sharing-attributer) mum_team () lown-board () apport board () thread_limit (integr-represention) for thread_limit (conter-store-courses)

OpenMP API 5.1	www.openmp.org	Paj
Directives and Constructs (conti	nued)	
masked construct masked (ragins) masked (ragin	exofoldance (p. as all p. as) concerning and the enclosed structured block into apparent units of work, each executed only once by ere thread. More subcarrier block more provides More subcarrier block More subcarrier block More subcarrier block	for simd and do simd [1:15.2] [2:8.3.2] Specifies that the iterations of suscitated loops will be encoured in parallel by threads in the term and the terratione security is able thread on this be encour- ted by able thread on this be encour- ted by the similar terration of the similar terration (security of the similar terration of the similar light security of the similar terration of the similar (security of the clause accepted by the simil, for, on de directions)
strictly-structured-block / !\$omp end masked)	Worksharing-loop construct for and do [2:1:.4] [2:9.2] Specifies that the iterations of associated loops will be	declare simd [2:11:5:3] [2:9:3:3] Applied to a function or a subroutine to enable the
Coope construct Coope (r,a) Defines a structured block that is executed by all threads in a train ball where additional OpenMire Operations can be specified. Image: Image additional OpenMire Operations (r, b) boxery structured block boxery structured block boxery structured block or the operation of the operation of the operation boxery structured block or the operation of the operation of the operation of the operation of the operation of the operation of the operation of the operation of the operation of the (ISong end stope (newski))	excected in parallel by freach in the team.	cration of one or more versions that can process matched argument (gal 500 instructions from a sin mercentine in a 5100 logic [] [] [] [] [] [] [] [] [] [] [] [] [] [
Close: private (ist) reduction (/reduction modifier, / reduction-identifier : list) (/to+nowait	order-modifier: reproducible, unconstrained Values for schedule kind: • static: Iterations are divided into chunks of size chunk_size and assigned to threads in the team in round-robin fashion in order of thread number.	uniform (argument-list) inbranch notinbranch distribute loop constructs
Worksharing constructs sections [2:0:1] [2:0:1] A non-iterative worksharing construct that contains a set of structured block that are to be distributed among and executed by the threads in a team.	round-doan taking in order of meaka number. dynamic: Each thread executes a chunk of terations then requests another chunk until non-meakan then requests another chunk until no chunks remain to be assigned. Chunk size is different for each chunk, with each successive chunk smaller than the last.	distribute [2:11.6:1][2:9.4:1] Specifies loops which are executed by the initial teams
#pragma omp sections (closus([,] closus(] #gragma omp section() structured block sequence #gragma omp section structured block sequence	aute: Compiler and/or runtime decides. runtime: Uses run sched-yun ICV. Values for sched-yun ICV. Walser for schedelse modifier: monotonic: Each thread executes the chunks that it is assigned in increasing logical Iteration order. A schedule (static) clause or order clause imples monotonic.	(* [15cmp end distribute) clouse: private (list) firstprivate (list) lastprivate (list) collapse (i) dist_schedue (list), change (list); allocate (joitecator: jinst) order ([/orde-mod/jirs:] concurrent)
Isone sections (closer) [.dtum?] [.dtum?] [Isone section block: sequence structure block: sequence [Isone sections lowesit] structure block: sequence] Closer: structure block: sequence] Isone sections lowesit] structure block: sequence]	 nomenonatoric: Chunks are assigned to threads in any order and the behavior of an application that depends on execution order of the chunks is unspecified. simd: ignored when the loop is not associated with a SIMC contruct, otherwise the new chunk; size/ for all except the first and last chunks is [chunk; size/ simd; with] "simd; with where simd; width is an implementation defined value. 	order-modifier: reproductibe unconstrained distribute simd [pi1.6.1] [p.6.2] Specifies a log but will be distributed across the prim threads of the trans region and executed concurrent ang SMD intruston.
(/C+ nowait single [2.10.2] [2.8.2]	SIMD directives and constructs simd [211.5.1] [2.9.3.1]	[ISomp end distribute simd] clouse: Any of the clauses accepted by distribute or sin
specifies that the associated structured block is executed by only one of the threads in the team in the structure block is the team in the structure block is the structure block is the structure block is the structure block is the structure block is the structure block is t	Applied to a loop to indicate that the loop can be transformed into 2 MMO loop.	distribute parallel for and distribute parallel do parallel do parallel do parallel do parallel do do parallel do do parallel do dorchives.
© 2020 OpenMP ARB		OMP1120-01-0

Recent Books About OpenMP





A printed copy of the 5.1 specifications, 2020

A book that covers all of the OpenMP 4.5 features, 2017

Ruud van der Pas, Eric Stotzer, and Christian Terboven

USING OPENMP-

THE NEXT STEP

Affinity, Accelerators, Tasking, and SIMD

OPE

THE OPENMP COMMON THE OPENMP **COMMON CORE** Making OpenMP Simple Again Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges

A new book about the OpenMP Common Core, 2019



Programming OpenMP

Parallel Region

Christian Terboven

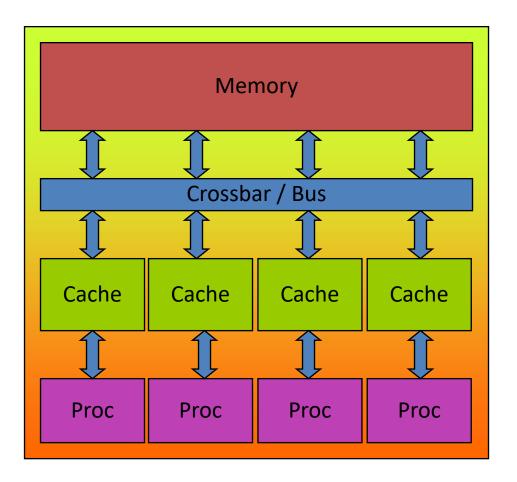
Michael Klemm



OpenMP's machine model



• OpenMP: Shared-Memory Parallel Programming Model.



All processors/cores access a shared main memory.

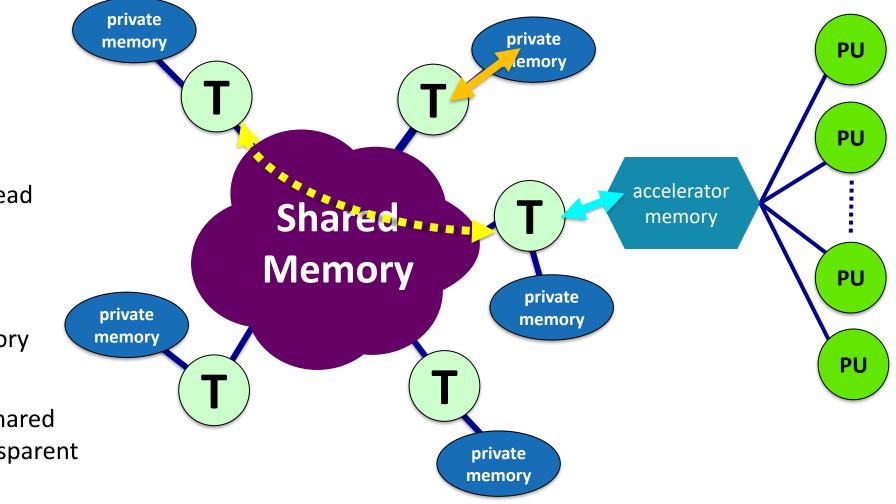
Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.



The OpenMP Memory Model

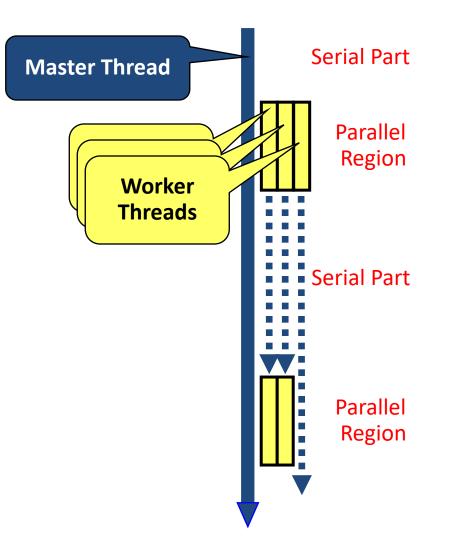
- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application





The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Master*.
- Worker threads are spawned at Parallel Regions, together with the Master they form the Team of threads.
- In between Parallel Regions the Worker threads are put to sleep.
 The OpenMP *Runtime* takes care of all thread management work.
- Concept: Fork-Join.
- Allows for an incremental parallelization!





Parallel Region and Structured Blocks

• The parallelism has to be expressed explicitly.

C/C++	Fortran
#pragma omp parallel	!\$omp parallel
{	 structured block
••• structured block	structured brock
• • •	!\$omp end parallel
}	

- Structured Block
 - Exactly one entry point at the top
 - Exactly one exit point at the bottom
 - Branching in or out is not allowed
 - Terminating the program is allowed (abort / exit)

- Specification of number of threads:
 - Environment variable: OMP_NUM_THREADS=...
 - Or: Via num_threads clause:
 add num_threads (num) to the parallel construct

OpenMP

Starting OpenMP Programs on Linux

• From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4
./program
```

• From within a shell, one-time setting of the number of threads:

OMP_NUM_THREADS=4 ./program





Hello OpenMP World



Programming OpenMP

Worksharing

Christian Terboven

Michael Klemm





For Worksharing

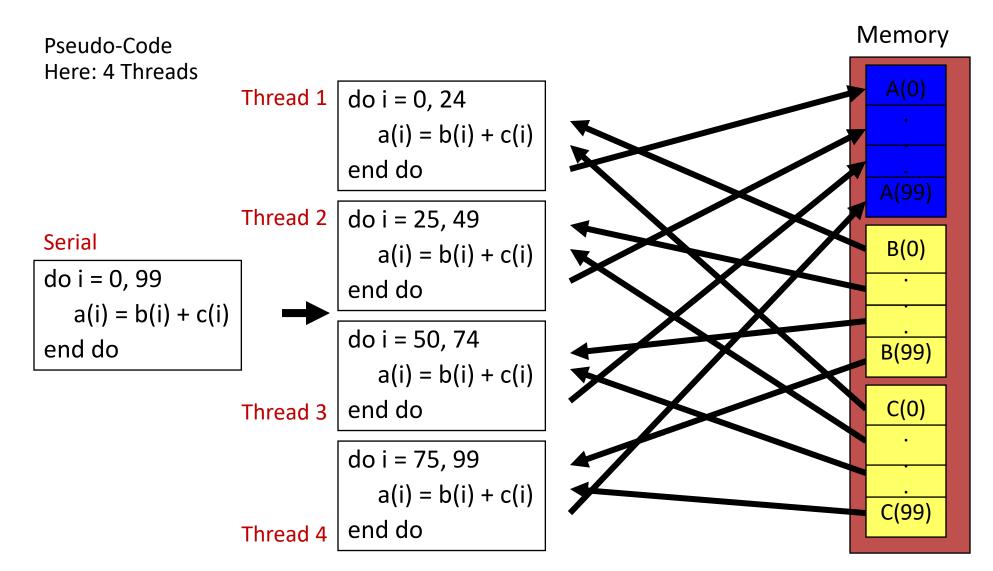
- If only the *parallel* construct is used, each thread executes the Structured Block.
- Program Speedup: *Worksharing*
- OpenMP's most common Worksharing construct: for

C/C++	Fortran
<pre>int i; #pragma omp for for (i = 0; i < 100; i++) { a[i] = b[i] + c[i];</pre>	<pre>INTEGER :: i !\$omp do DO i = 0, 99 a[i] = b[i] + c[i] END DO</pre>
}	

- Distribution of loop iterations over all threads in a Team.
- Scheduling of the distribution can be influenced.
- Loops often account for most of a program's runtime!

Worksharing illustrated





3



The Barrier Construct

- **OpenMP** barrier (implicit or explicit)
 - Threads wait until all threads of the current *Team* have reached the barrier

C/C++ #pragma omp barrier

• All worksharing constructs contain an implicit barrier at the end



The Single Construct

C/C++	Fortran
<pre>#pragma omp single [clause] structured block</pre>	<pre>!\$omp single [clause] structured block</pre>
	!\$omp end single

- The single construct specifies that the enclosed structured block is executed by only on thread of the team.
 - It is up to the runtime which thread that is.
- Useful for:
 - I/O
 - Memory allocation and deallocation, etc. (in general: setup work)
 - Implementation of the single-creator parallel-executor pattern as we will see later...



The Master Construct

C/C++	Fortran
<pre>#pragma omp master[clause] structured block</pre>	<pre>!\$omp master[clause] structured block</pre>
	!\$omp end master

- The master construct specifies that the enclosed structured block is executed only by the master thread of a team.
- Note: The master construct is no worksharing construct and does not contain an implicit barrier at the end.

Demo

7



Vector Addition



Influencing the For Loop Scheduling / 1

- *for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:
 - schedule(static [, chunk]): Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
 - schedule(dynamic [, chunk]): Iteration space divided into blocks of chunk (not specified: 1) size,
 blocks are scheduled to threads in the order in which threads finish previous blocks.
 - schedule(guided [, chunk]): Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- **Default is** schedule (static).

Influencing the For Loop Scheduling / 2



Static Schedule

→ schedule(static [, chunk])

→ Decomposition

depending on chunksize

→ Equal parts of size 'chunksize' distributed in round-robin

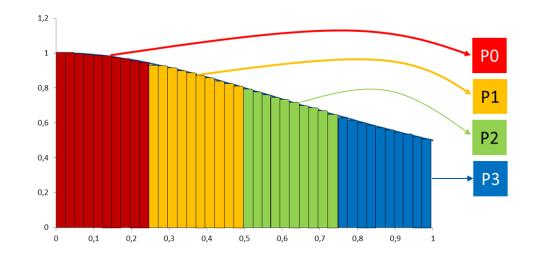
fashion

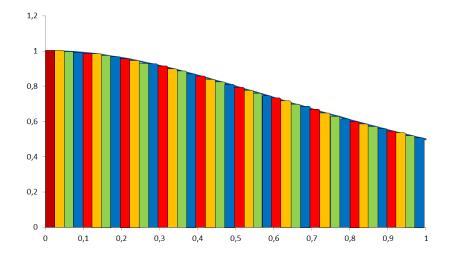
Pros?

 \rightarrow No/low runtime overhead

Cons?

 \rightarrow No dynamic workload balancing







Influencing the For Loop Scheduling / 3

- Dynamic schedule
 - schedule(dynamic [, chunk])
 - Iteration space divided into blocks of chunk size
 - Threads request a new block after finishing the previous one
 - Default chunk size is 1
- Pros ?
 - Workload distribution
- Cons?
 - Runtime Overhead
 - Chunk size essential for performance
 - No NUMA optimizations possible



Synchronization Overview

- Can all loops be parallelized with for-constructs? No!
 - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent.
 BUT: This test alone is not sufficient:

```
C/C++
int i, int s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    s = s + a[i];
}</pre>
```

• *Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).



Synchronization: Critical Region

• A Critical Region is executed by all threads, but by only one thread simultaneously (Mutual Exclusion).

C/C++
#pragma omp critical (name)
{
 ... structured block ...
}

• Do you think this solution scales well?

C/C++	
<pre>int i, s = 0; #pragma omp parallel for (i = 0; i < 100; {</pre>	
<pre>#pragma omp critical { s = s + a[i]; }</pre>	}

12



Programming OpenMP

Scoping

Christian Terboven

Michael Klemm



OpenMP Tutorial Members of the OpenMP Language Committee

Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.
- *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
 - *private*-list and *shared*-list on Parallel Region
 - *private*-list and *shared*-list on Worksharing constructs
 - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
 - Loop control variables on *for*-constructs are *private*
 - Non-static variables local to Parallel Regions are *private*
 - *private*: A new uninitialized instance is created for the task or each thread executing the construct
 - *firstprivate*: Initialization with the value before encountering the construct
 - *lastprivate*: Value of last loop iteration is written back to Master
 - Static variables are *shared*

Tasks are introduced later



Privatization of Global/Static Variables

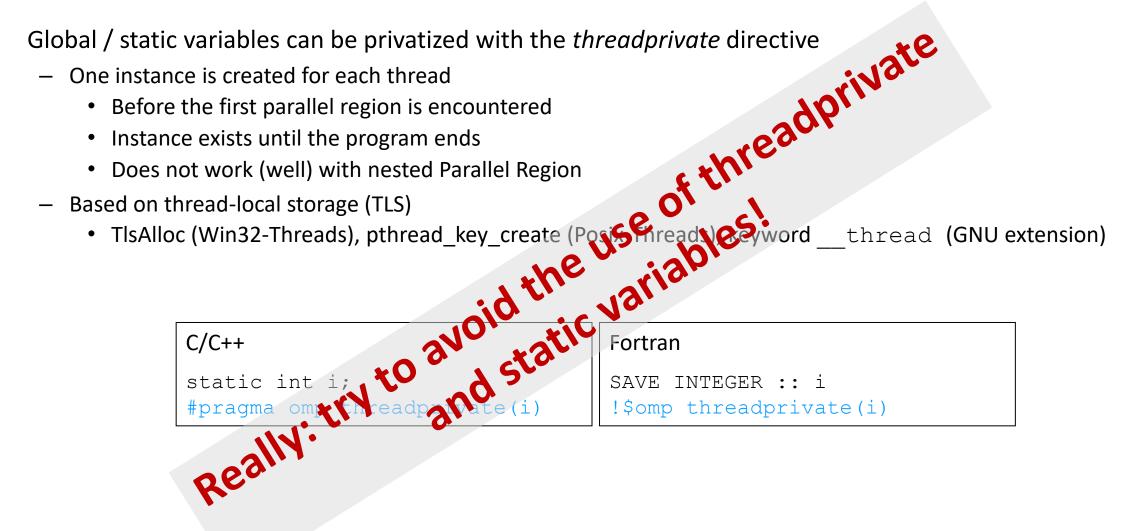


- Global / static variables can be privatized with the *threadprivate* directive
 - One instance is created for each thread
 - Before the first parallel region is encountered
 - Instance exists until the program ends
 - Does not work (well) with nested Parallel Region
 - Based on thread-local storage (TLS)
 - TIsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword ____thread (GNU extension)

C/C++	Fortran
	SAVE INTEGER :: i !\$omp threadprivate(i)

Privatization of Global/Static Variables

- ٠





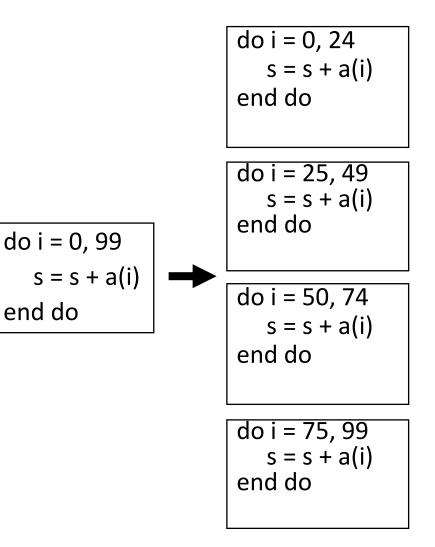
Back to our example

C/C++

```
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
        { s = s + a[i]; }
}
```

It's your turn: Make It Scale!





#pragma omp parallel

#pragma omp for
 for (i = 0; i < 99; i++)
 {</pre>

$$s = s + a[i];$$

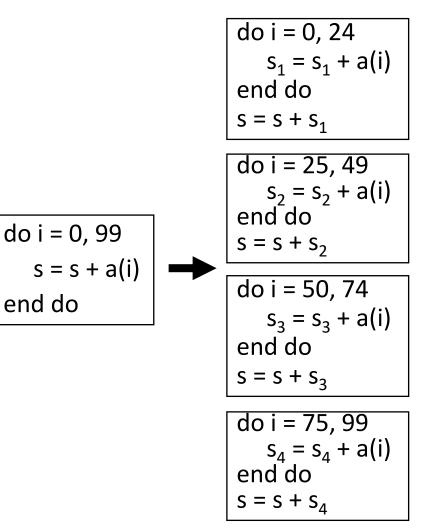
} // end parallel

6

(done)



#pragma omp parallel { double ps = 0.0; // private variable #pragma omp for for (i = 0; i < 99; i++)ps = ps + a[i];#pragma omp critical s += ps; // end parallel





The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
 - reduction(operator:list)
 - The result is provided in the associated reduction variable

```
C/C++
int i, s = 0;
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
    s = s + a[i];
}</pre>
```

Possible reduction operators with initialization value:

+ (0), * (1), - (0), & (~0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)

Remark: OpenMP also supports user-defined reductions (not covered here)

Example





Example: Pi (1/2)

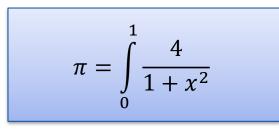
```
double f(double x)
  return (4.0 / (1.0 + x^*x));
```

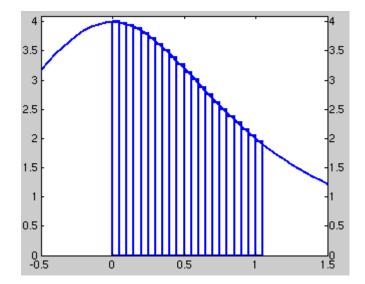
```
double CalcPi (int n)
```

```
const double fH = 1.0 / (double) n;
double fSum = 0.0;
double fX;
int i;
```

```
#pragma omp parallel for
  for (i = 0; i < n; i++)
    fX = fH * ((double)i + 0.5);
    fSum += f(fX);
  }
  return fH * fSum;
```







10

Example: Pi (2/2)

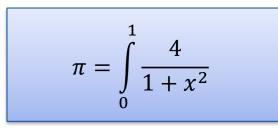
```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

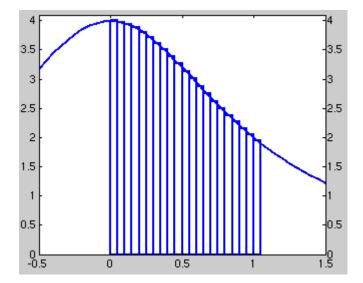
```
double CalcPi (int n)
```

```
const double fH = 1.0 / (double) n;
double fSum = 0.0;
double fX;
int i;
```

```
#pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}</pre>
```









Programming OpenMP

OpenMP Tasking Introduction

Christian Terboven Michael Klemm



What is a Task in OpenMP?



- Tasks are work units whose execution
 - \rightarrow may be deferred or...
 - \rightarrow ... can be executed immediately
- Tasks are composed of
 - > code to execute, a data environment (initialized at creation time), internal control variables (ICVs)
- Tasks are created...
 - ... when reaching a parallel region \rightarrow implicit tasks are created (per thread)
 - ... when encountering a task construct \rightarrow explicit task is created
 - ... when encountering a taskloop construct \rightarrow explicit tasks per chunk are created
 - ... when encountering a target construct \rightarrow target task is created

Tasking Execution Model



- Supports unstructured parallelism
 - → unbounded loops

while (<expr></expr>)	{
}			

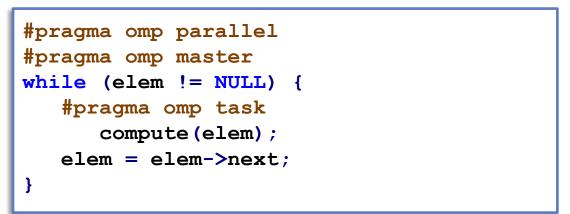
 \rightarrow recursive functions

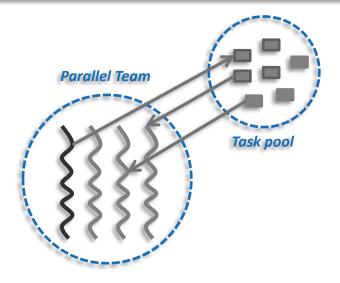
```
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```

Several scenarios are possible:

- → single creator, multiple creators, nested tasks (tasks & WS)
- All threads in the team are candidates to execute tasks

Example (unstructured parallelism)





OpenMP Tasking Idiom



- OpenMP programmers need a specific idiom to kick off task-parallel execution: parallel master
 - \rightarrow OpenMP version 5.0 introduced the parallel master construct
 - → With OpenMP version 5.1 this becomes parallel masked

```
int main(int argc, char* argv[])
                                                                 int main(int argc, char* argv[])
 1
                                                             1
 2
                                                             2
    ł
                                                                 ł
 3
         [...]
                                                             3
                                                                      \left[ \ldots \right]
         #pragma omp parallel
                                                                     #pragma omp parallel
 4
                                                             4
 5
                                                             5
 6
            #pragma omp master
                                                                         #pragma omp single
                                                             6
 7
                                                             7
                 start task parallel execution();
                                                                              start_task_parallel_execution();
 9
                                                             9
 9
                                                             9
             }
10
                                                            10
         [...]
                                                                      [...]
11
                                                            11
12 }
                                                            12 }
```

Fibonacci Numbers (in a Stupid Way ^(C))

```
int main(int argc,
                                         int fib(int n)
                                                           {
                                     14
                                              if (n < 2) return n;
         char* argv[])
                                     15
                                     16
                                              int x, y;
                                              #pragma omp task shared(x)
    [...]
                                     17
    #pragma omp parallel
                                     18
                                              {
                                                  x = fib(n - 1);
                                     19
       #pragma omp master
                                     20
                                     21
                                              #pragma omp task shared(y)
           fib(input);
                                     22
                                              {
                                                  y = fib(n - 2);
                                     23
                                     24
    [...]
                                     25
                                              #pragma omp taskwait
                                     26
                                                  return x+y;
                                     27 }
```

- Only one thread enters fib() from main().
- That thread creates the two initial work tasks and starts the parallel recursion.
- The taskwait construct is required to wait for the result for x and y before the task can sum up.

1

2

3

4

5

6 7

8 9

10

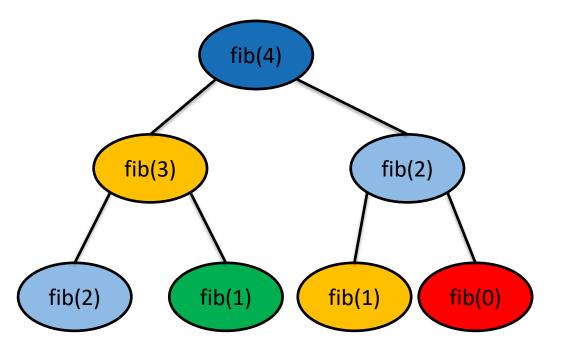
11

12

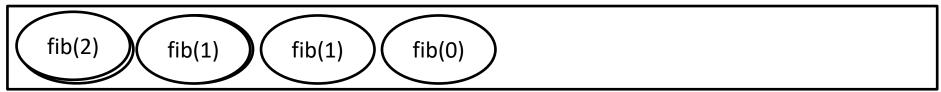
13 }



- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 T4 execute tasks









- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 T4 execute tasks

fib(3) fib(2) fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

fib(4)

. . .



Programming OpenMP

Using OpenMP Compilers

Christian Terboven Michael Klemm



Production Compilers w/ OpenMP Support



GCC

- clang/LLVM
- Intel Classic and Next-gen Compilers
- AOCC, AOMP, ROCmCC
- IBM XL
- ... and many more

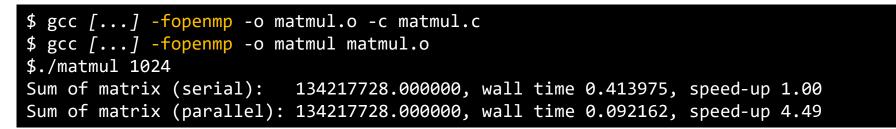
See https://www.openmp.org/resources/openmp-compilers-tools/ for a list

Compiling OpenMP



- Enable OpenMP via the compiler's command-line switches
 - → GCC: -fopenmp
 - → clang: -fopenmp
 - → Intel: -fopenmp or -qopenmp (classic) or -fiopenmp (next-gen)
 - \rightarrow AOCC, AOCL, ROCmCC: -fopenmp
 - → IBM XL: -qsmp=omp

Switches have to be passed to both compiler and linker:



10



Programming OpenMP

Hands-on Exercises

Christian Terboven Michael Klemm



11

Webinar Exercises



- We have implemented a series of small hands-on examples that you can use and play with.
 - → Download: git clone https://github.com/cterboven/OpenMP-tutorial-PRACE.git
 - → Build: make
 - \rightarrow You can then find the compiled code in the "bin" folder to run it
 - → We use the GCC compiler mostly, some examples require Intel's Math Kernel Library
- Each hands-on exercise has a folder "solution"
 - \rightarrow It shows the OpenMP directive that we have added
 - \rightarrow You can use it to cheat \odot , or to check if you came up with the same solution