Programming the OpenMP API

Introduction

Michael Klemm

Principal Member of Technical Staff
Compilers, Languages, Runtimes & Tools
Machine Learning & Software Engineering

Chief Executive Office
OpenMP Architecture Review Board

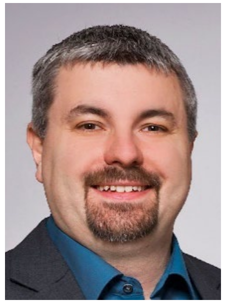
Credits...

Michael Klemm — OpenMP AMD

Christian Terboven — NHR4CES | RWTH AACHEN UNIVERSITY

Bronis R. de Supinski — LAWRENCE LIVERMORE NATIONAL LABORATORY

Xavier Teruel — BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación



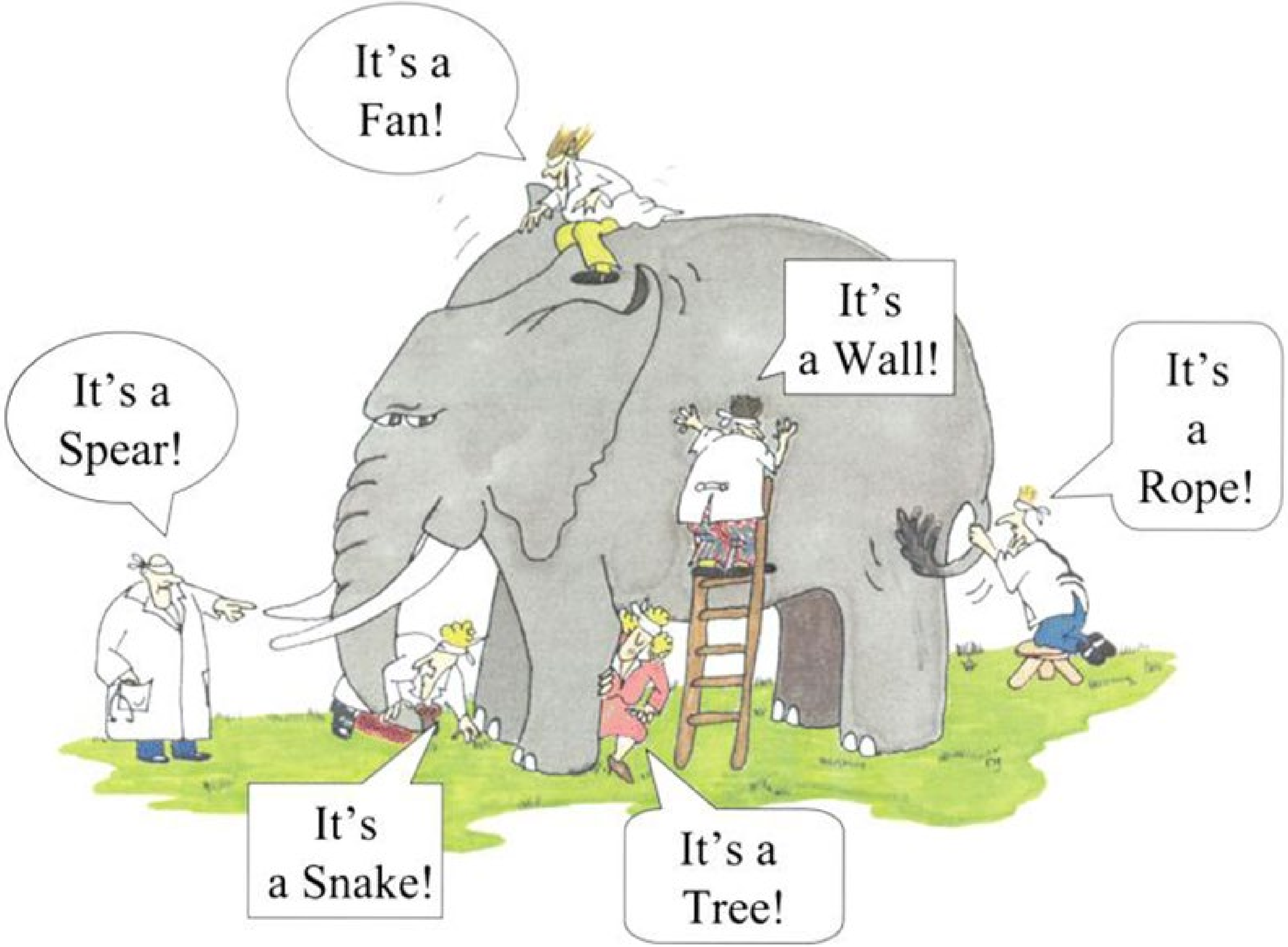
History

- 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
- 11/2021: OpenMP 5.2

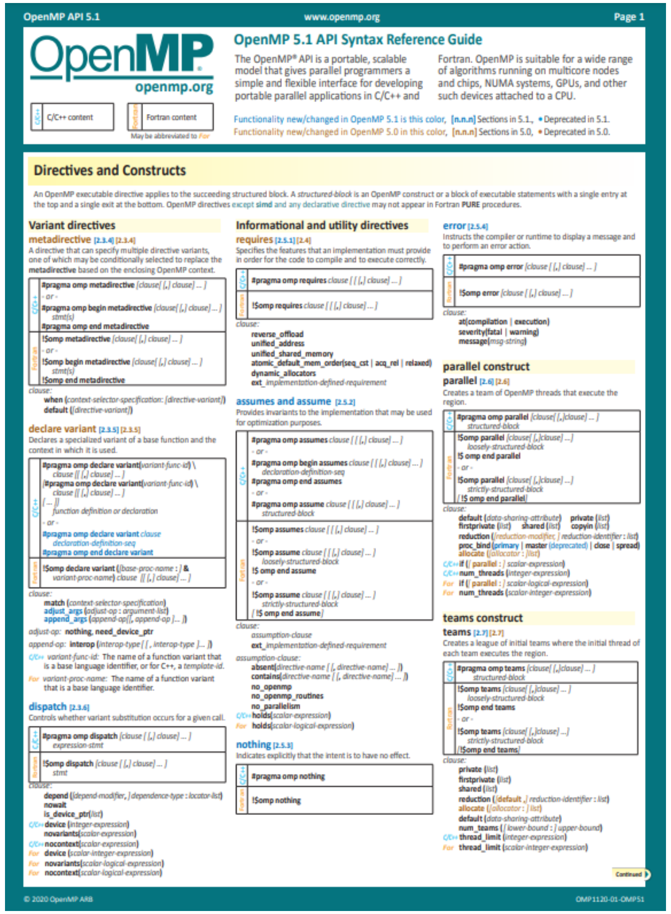
http://www.OpenMP.org

What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- …



Get your C/C++ and Fortran Reference Guide!
Covers all of OpenMP 5.1/5.2!



OpenMP API 5.1 www.openmp.org Page 1

openmp.org

OpenMP 5.1 API Syntax Reference Guide

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.

C/C++ | C/C++ content
Fortran | Fortran content
May be abbreviated to For

Functionality new/changed in OpenMP 5.1 is this color, [n.n.n] Sections in 5.1., • Deprecated in 5.1.
Functionality new/changed in OpenMP 5.0 in this color, [n.n.n] Sections in 5.0, • Deprecated in 5.0.

Directives and Constructs

An OpenMP executable directive applies to the succeeding structured block. A *structured-block* is an OpenMP construct or a block of executable statements with a single entry at the top and a single exit at the bottom. OpenMP directives except **simd** and any declarative directive may not appear in Fortran **PURE** procedures.

Variant directives

metadirective [2.3.4] [2.3.4]
A directive that can specify multiple directive variants, one of which may be conditionally selected to replace the **metadirective** based on the enclosing OpenMP context.

C/C++
```
#pragma omp metadirective [clause[ [,] clause] ... ]
- or -
#pragma omp begin metadirective [clause[ [,] clause] ... ]
    stmt(s)
#pragma omp end metadirective
```
Fortran
```
!$omp metadirective [clause[ [,] clause] ... ]
- or -
!$omp begin metadirective [clause[ [,] clause] ... ]
    stmt(s)
!$omp end metadirective
```
clause:
when (*context-selector-specification*: [*directive-variant*])
default ([*directive-variant*])

declare variant [2.3.5] [2.3.5]
Declares a specialized variant of a base function and the context in which it is used.

C/C++
```
#pragma omp declare variant(variant-func-id) \
    clause [[ [,] clause] ... ]
[#pragma omp declare variant(variant-func-id) \
    clause [[ [,] clause] ... ]
[ ... ]]
    function definition or declaration
- or -
#pragma omp declare variant clause
    declaration-definition-seq
#pragma omp end declare variant
```
Fortran
```
!$omp declare variant ([base-proc-name : ] &
    variant-proc-name) clause [[ [,] clause] ... ]
```
clause:
match (*context-selector-specification*)
adjust_args (*adjust-op* : *argument-list*)
append_args (*append-op*[[, *append-op*]...])

adjust-op: **nothing, need_device_ptr**
append-op: **interop** (*interop-type* [[, *interop-type*]...])
C/C++ *variant-func-id:* The name of a function variant that is a base language identifier, or for C++, a *template-id*.
For *variant-proc-name:* The name of a function variant that is a base language identifier.

dispatch [2.3.6]
Controls whether variant substitution occurs for a given call.

C/C++
```
#pragma omp dispatch [clause [ [,] clause] ... ]
    expression-stmt
```
Fortran
```
!$omp dispatch [clause [ [,] clause] ... ]
    stmt
```
clause:
depend ([*depend-modifier*,] *dependence-type* : *locator-list*)
nowait
is_device_ptr(*list*)
C/C++ **device** (*integer-expression*)
novariants(*scalar-expression*)
C/C++ **nocontext**(*scalar-expression*)
For **device** (*scalar-integer-expression*)
For **novariants**(*scalar-logical-expression*)
For **nocontext**(*scalar-logical-expression*)

Informational and utility directives

requires [2.5.1] [2.4]
Specifies the features that an implementation must provide in order for the code to compile and to execute correctly.

C/C++
```
#pragma omp requires clause [ [ [,] clause] ... ]
```
Fortran
```
!$omp requires clause [ [ [,] clause] ... ]
```
clause:
reverse_offload
unified_address
unified_shared_memory
atomic_default_mem_order(**seq_cst** | **acq_rel** | **relaxed**)
dynamic_allocators
ext_*implementation-defined-requirement*

assumes and assume [2.5.2]
Provides invariants to the implementation that may be used for optimization purposes.

C/C++
```
#pragma omp assumes clause [ [ [,] clause] ... ]
- or -
#pragma omp begin assumes clause [ [ [,] clause] ... ]
    declaration-definition-seq
#pragma omp end assumes
- or -
#pragma omp assume clause [ [ [,] clause] ... ]
    structured-block
```
Fortran
```
!$omp assumes clause [ [ [,] clause] ... ]
- or -
!$omp assume clause [ [ [,] clause] ... ]
    loosely-structured-block
!$ omp end assume
- or -
!$omp assume clause [ [ [,] clause] ... ]
    strictly-structured-block
[ !$ omp end assume]
```
clause:
assumption-clause
ext_*implementation-defined-requirement*
assumption-clause:
absent(*directive-name* [[, *directive-name*] ...])
contains(*directive-name* [[, *directive-name*] ...])
no_openmp
no_openmp_routines
no_parallelism
C/C++ **holds**(*scalar-expression*)
For **holds**(*scalar-logical-expression*)

nothing [2.5.3]
Indicates explicitly that the intent is to have no effect.

C/C++
```
#pragma omp nothing
```
Fortran
```
!$omp nothing
```

error [2.5.4]
Instructs the compiler or runtime to display a message and to perform an error action.

C/C++
```
#pragma omp error [clause [ [,] clause] ... ]
```
Fortran
```
!$omp error [clause [ [,] clause] ... ]
```
clause:
at(**compilation** | **execution**)
severity(**fatal** | **warning**)
message(*msg-string*)

parallel construct

parallel [2.6] [2.6]
Creates a team of OpenMP threads that execute the region.

C/C++
```
#pragma omp parallel [clause[ [,]clause] ... ]
    structured-block
```
Fortran
```
!$omp parallel [clause[ [,]clause] ...]
    loosely-structured-block
!$ omp end parallel
- or -
!$omp parallel [clause[ [,]clause] ...]
    strictly-structured-block
[ !$ omp end parallel]
```
clause:
default (*data-sharing-attribute*) **private** (*list*)
firstprivate (*list*) **shared** (*list*) **copyin** (*list*)
reduction ([*reduction-modifier*,] *reduction-identifier* : *list*)
proc_bind(**primary** | **master** (deprecated) | **close** | **spread**)
allocate ([*allocator* :]*list*)
C/C++ **if** ([**parallel** :] *scalar-expression*)
C/C++ **num_threads** (*integer-expression*)
For **if** ([**parallel** :] *scalar-logical-expression*)
For **num_threads** (*scalar-integer-expression*)

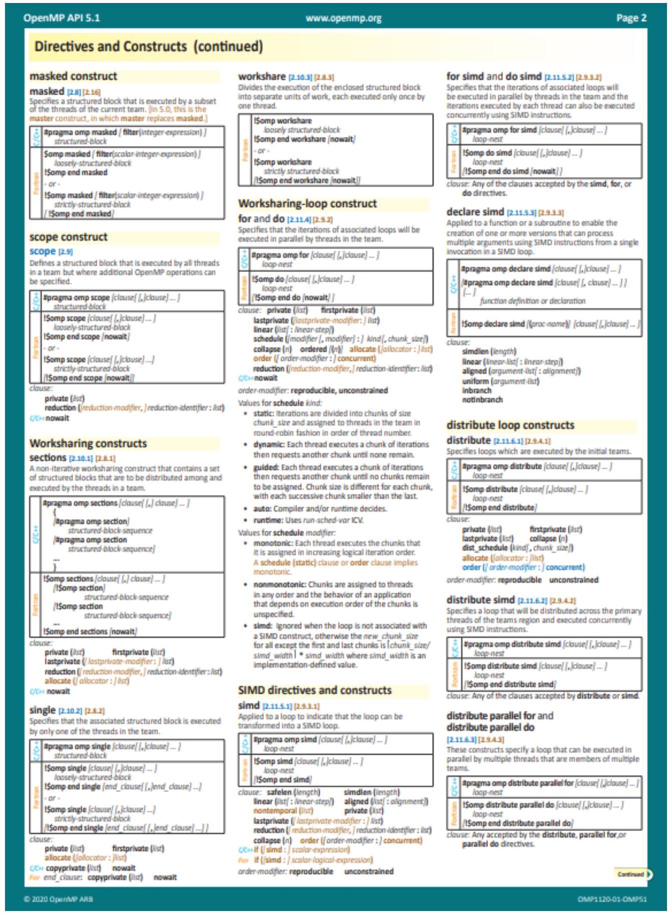
teams construct

teams [2.7] [2.7]
Creates a league of initial teams where the initial thread of each team executes the region.

C/C++
```
#pragma omp teams [clause[ [,]clause] ... ]
    structured-block
```
Fortran
```
!$omp teams [clause[ [,]clause] ... ]
    loosely-structured-block
!$omp end teams
- or -
!$omp teams [clause[ [,]clause] ...]
    strictly-structured-block
[!$omp end teams]
```
clause:
private (*list*)
firstprivate (*list*)
shared (*list*)
reduction ([**default** ,] *reduction-identifier* : *list*)
allocate ([*allocator* :] *list*)
default (*data-sharing-attribute*)
num_teams ([*lower-bound* :] *upper-bound*)
C/C++ **thread_limit** (*integer-expression*)
For **thread_limit** (*scalar-integer-expression*)

Continued

© 2020 OpenMP ARB OMP1120-01-OMP51



OpenMP API 5.1 www.openmp.org Page 2

Directives and Constructs (continued)

masked construct

masked [2.8] [2.16]
Specifies a structured block that is executed by a subset of the threads of the current team. [In 5.0, this is the **master** construct, in which **master** replaces **masked**.]

C/C++
```
#pragma omp masked [ filter(integer-expression) ]
    structured-block
```
Fortran
```
$omp masked [ filter(scalar-integer-expression) ]
    loosely-structured-block
!$omp end masked
- or -
!$omp masked [ filter(scalar-integer-expression) ]
    strictly-structured-block
[ !$omp end masked]
```

scope construct

scope [2.9]
Defines a structured block that is executed by all threads in a team but where additional OpenMP operations can be specified.

C/C++
```
#pragma omp scope [clause[ [,]clause] ... ]
    structured-block
```
Fortran
```
!$omp scope [clause[ [,]clause] ... ]
    loosely-structured-block
!$omp end scope [nowait]
- or -
!$omp scope [clause[ [,]clause] ...]
    strictly-structured-block
[!$omp end scope [nowait]]
```
clause:
private (*list*)
reduction ([*reduction-modifier*,] *reduction-identifier* : *list*)
C/C++ **nowait**

Worksharing constructs

sections [2.10.1] [2.8.1]
A non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

C/C++
```
#pragma omp sections [clause[ [,] clause] ... ]
    {
    [#pragma omp section]
        structured-block-sequence
    [#pragma omp section
        structured-block-sequence]
    ...
    }
```
Fortran
```
!$omp sections [clause[ [,] clause] ... ]
    [!$omp section]
        structured-block-sequence
    [!$omp section
        structured-block-sequence]
    ...
!$omp end sections [nowait]
```
clause:
private (*list*) **firstprivate** (*list*)
lastprivate ([*lastprivate-modifier* :] *list*)
reduction ([*reduction-modifier*,] *reduction-identifier* : *list*)
allocate ([*allocator* :] *list*)
C/C++ **nowait**

single [2.10.2] [2.8.2]
Specifies that the associated structured block is executed by only one of the threads in the team.

C/C++
```
#pragma omp single [clause[ [,]clause] ... ]
    structured-block
```
Fortran
```
!$omp single [clause[ [,]clause] ... ]
    loosely-structured-block
!$omp end single [end_clause[ [,]end_clause] ...]
- or -
!$omp single [clause[ [,]clause] ...]
    strictly-structured-block
[!$omp end single [end_clause[ [,]end_clause] ...] ]
```
clause:
private (*list*) **firstprivate** (*list*)
allocate ([*allocator* :]*list*)
C/C++ **copyprivate** (*list*) **nowait**
For *end_clause*: **copyprivate** (*list*) **nowait**

workshare [2.10.3] [2.8.3]
Divides the execution of the enclosed structured block into separate units of work, each executed only once by one thread.

Fortran
```
!$omp workshare
    loosely structured-block
!$omp end workshare [nowait]
- or -
!$omp workshare
    strictly structured-block
[!$omp end workshare [nowait]]
```

Worksharing-loop construct

for and **do** [2.11.4] [2.9.2]
Specifies that the iterations of associated loops will be executed in parallel by threads in the team.

C/C++
```
#pragma omp for [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp do [clause[ [,]clause] ... ]
    loop-nest
[!$omp end do [nowait] ]
```
clause: **private** (*list*) **firstprivate** (*list*)
lastprivate ([*lastprivate-modifier*:] *list*)
linear (*list*[: *linear-step*])
schedule ([*modifier* [, *modifier*] :] *kind*[, *chunk_size*])
collapse (*n*) **ordered** [(*n*)] **allocate** ([*allocator* :] *list*)
order ([*order-modifier* :] **concurrent**)
reduction ([*reduction-modifier*,] *reduction-identifier*: *list*)
C/C++ **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.
- **dynamic:** Each thread executes a chunk of iterations then requests another chunk until none remain.
- **guided:** Each thread executes a chunk of iterations 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.
- **auto:** Compiler and/or runtime decides.
- **runtime:** Uses *run-sched-var* ICV.

Values for **schedule** *modifier*:

- **monotonic:** Each thread executes the chunks that it is assigned in increasing logical iteration order. A **schedule** (**static**) clause or **order** clause implies **monotonic**.
- **nonmonotonic:** 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 SIMD construct, otherwise the *new_chunk_size* for all except the first and last chunks is ⌈*chunk_size*/*simd_width*⌉ * *simd_width* where *simd_width* is an implementation-defined value.

SIMD directives and constructs

simd [2.11.5.1] [2.9.3.1]
Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

C/C++
```
#pragma omp simd [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp simd [clause[ [,]clause] ... ]
    loop-nest
[!$omp end simd]
```
clause: **safelen** (*length*) **simdlen** (*length*)
linear (*list*[: *linear-step*]) **aligned** (*list*[: *alignment*])
nontemporal (*list*) **private** (*list*)
lastprivate ([*lastprivate-modifier* :] *list*)
reduction ([*reduction-modifier*,] *reduction-identifier* : *list*)
collapse (*n*) **order** ([*order-modifier* :] **concurrent**)
C/C++ **if** ([**simd** :] *scalar-expression*)
For **if** ([**simd** :] *scalar-logical-expression*)
order-modifier: **reproducible unconstrained**

for simd and **do simd** [2.11.5.2] [2.9.3.2]
Specifies that the iterations of associated loops will be executed in parallel by threads in the team and the iterations executed by each thread can also be executed concurrently using SIMD instructions.

C/C++
```
#pragma omp for simd [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp do simd [clause[ [,]clause] ... ]
    loop-nest
[!$omp end do simd [nowait] ]
```
clause: Any of the clauses accepted by the **simd**, **for**, or **do** directives.

declare simd [2.11.5.3] [2.9.3.3]
Applied to a function or a subroutine to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop.

C/C++
```
#pragma omp declare simd [clause[ [,]clause] ... ]
[#pragma omp declare simd [clause[ [, clause] ... ] ]
[ ... ]
    function definition or declaration
```
Fortran
```
!$omp declare simd [(proc-name)] [clause[ [,]clause] ... ]
```
clause:
simdlen (*length*)
linear (*linear-list*[: *linear-step*])
aligned (*argument-list*[: *alignment*])
uniform (*argument-list*)
inbranch
notinbranch

distribute loop constructs

distribute [2.11.6.1] [2.9.4.1]
Specifies loops which are executed by the initial teams.

C/C++
```
#pragma omp distribute [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp distribute [clause[ [,]clause] ... ]
    loop-nest
[!$omp end distribute]
```
clause:
private (*list*) **firstprivate** (*list*)
lastprivate (*list*) **collapse** (*n*)
dist_schedule (*kind*[, *chunk_size*])
allocate ([*allocator* :]*list*)
order ([*order-modifier* :] **concurrent**)
order-modifier: **reproducible unconstrained**

distribute simd [2.11.6.2] [2.9.4.2]
Specifies a loop that will be distributed across the primary threads of the teams region and executed concurrently using SIMD instructions.

C/C++
```
#pragma omp distribute simd [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp distribute simd [clause[ [,]clause] ... ]
    loop-nest
[!$omp end distribute simd]
```
clause: Any of the clauses accepted by **distribute** or **simd**.

distribute parallel for and **distribute parallel do**
[2.11.6.3] [2.9.4.3]
These constructs specify a loop that can be executed in parallel by multiple threads that are members of multiple teams.

C/C++
```
#pragma omp distribute parallel for [clause[ [,]clause] ... ]
    loop-nest
```
Fortran
```
!$omp distribute parallel do [clause[ [,]clause] ... ]
    loop-nest
[!$omp end distribute parallel do]
```
clause: Any accepted by the **distribute**, **parallel for**, or **parallel do** directives.

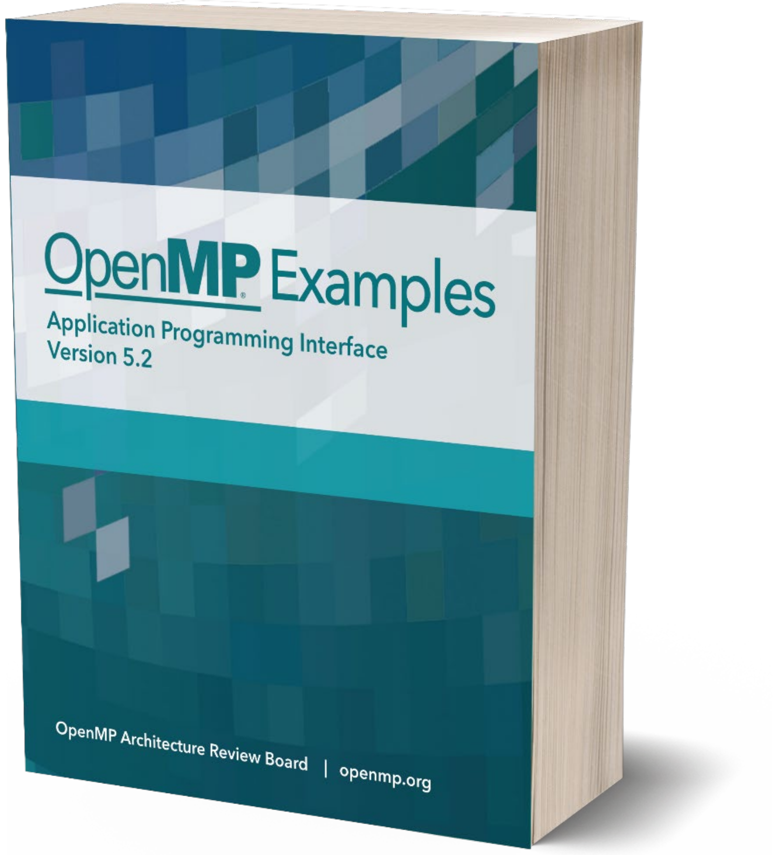
Continued

© 2020 OpenMP ARB OMP1120-01-OMP51

OpenMP API Specification & Examples

OpenMP
Application Programming Interface
Specification Version 5.2
OpenMP Architecture Review Board | openmp.org

https://link.openmp.org/book52
https://link.openmp.org/tr11

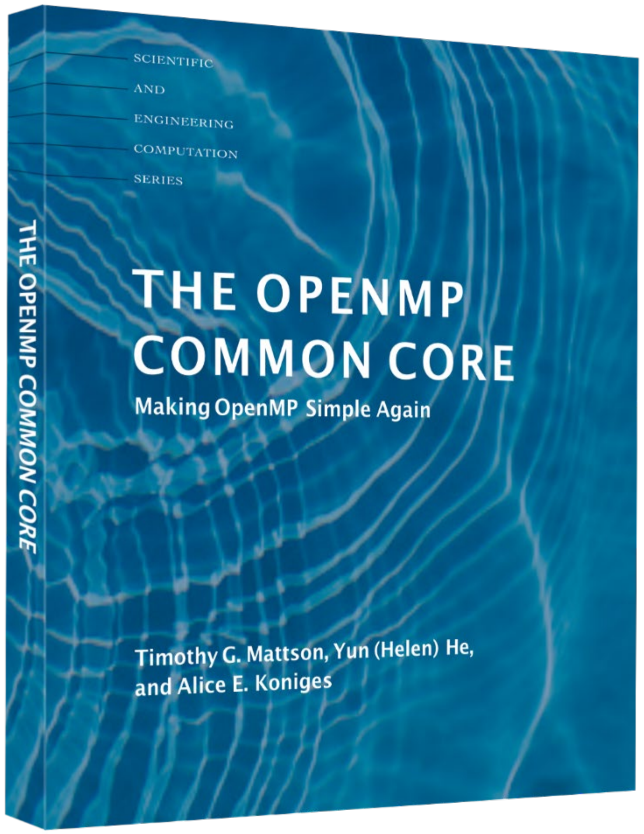

OpenMP Examples
Application Programming Interface
Version 5.2
OpenMP Architecture Review Board | openmp.org

https://link.openmp.org/examples521

Recent Books About OpenMP

SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven
USING OPENMP – THE NEXT STEP

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

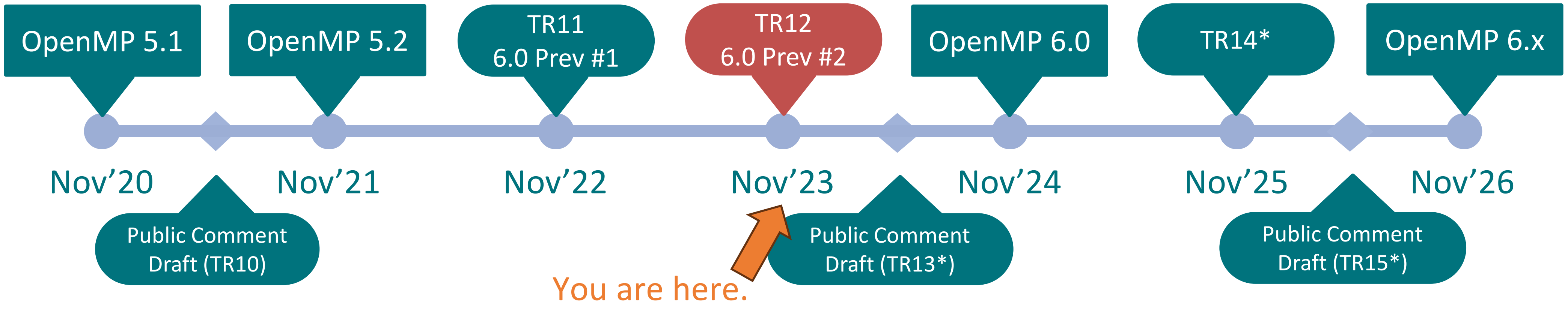


A new book about the OpenMP Common Core, 2019

OpenMP Roadmap

- Roadmap for the releases of the OpenMP API
  - 5-year cadence for major releases, one minor release in between
  - OpenMP 5.2 was an additional release before OpenMP version 6.0
  - (At least) one Technical Report (TR) with feature previews in every year

Version	Year
4.0	2013
5.0	2018
6.0	2024
7.0	2029

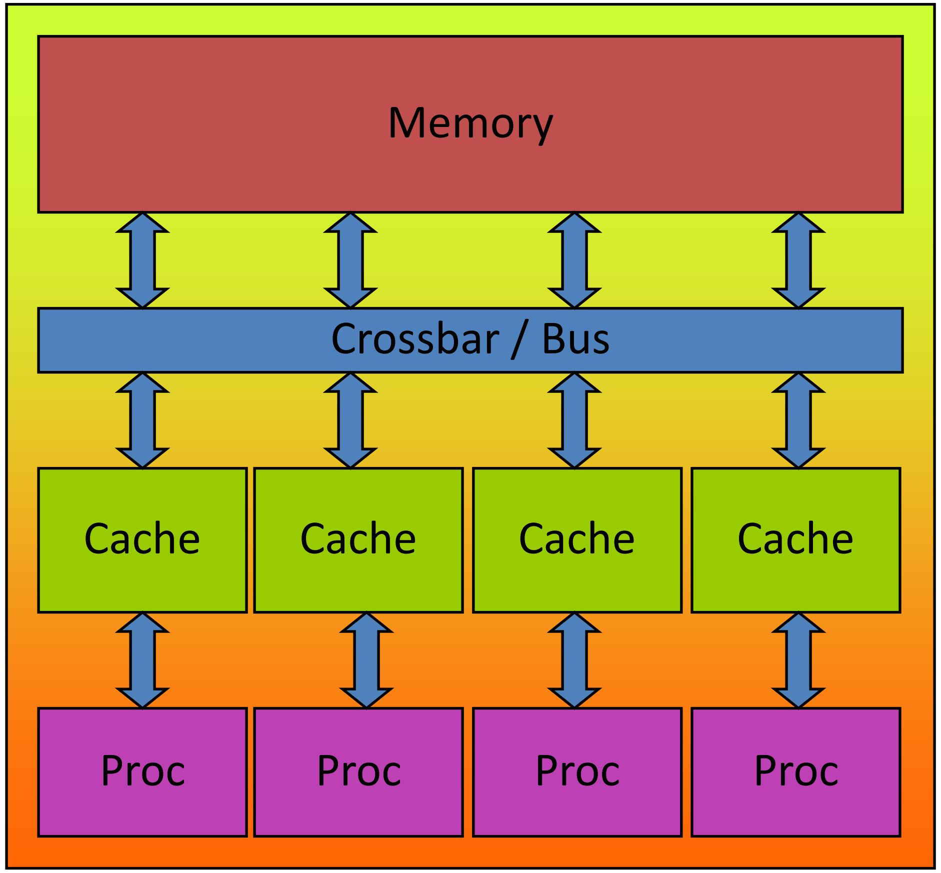


Programming the OpenMP API

Parallel Region

OpenMP‘s machine model

- OpenMP: Shared-Memory Parallel Programming Model.



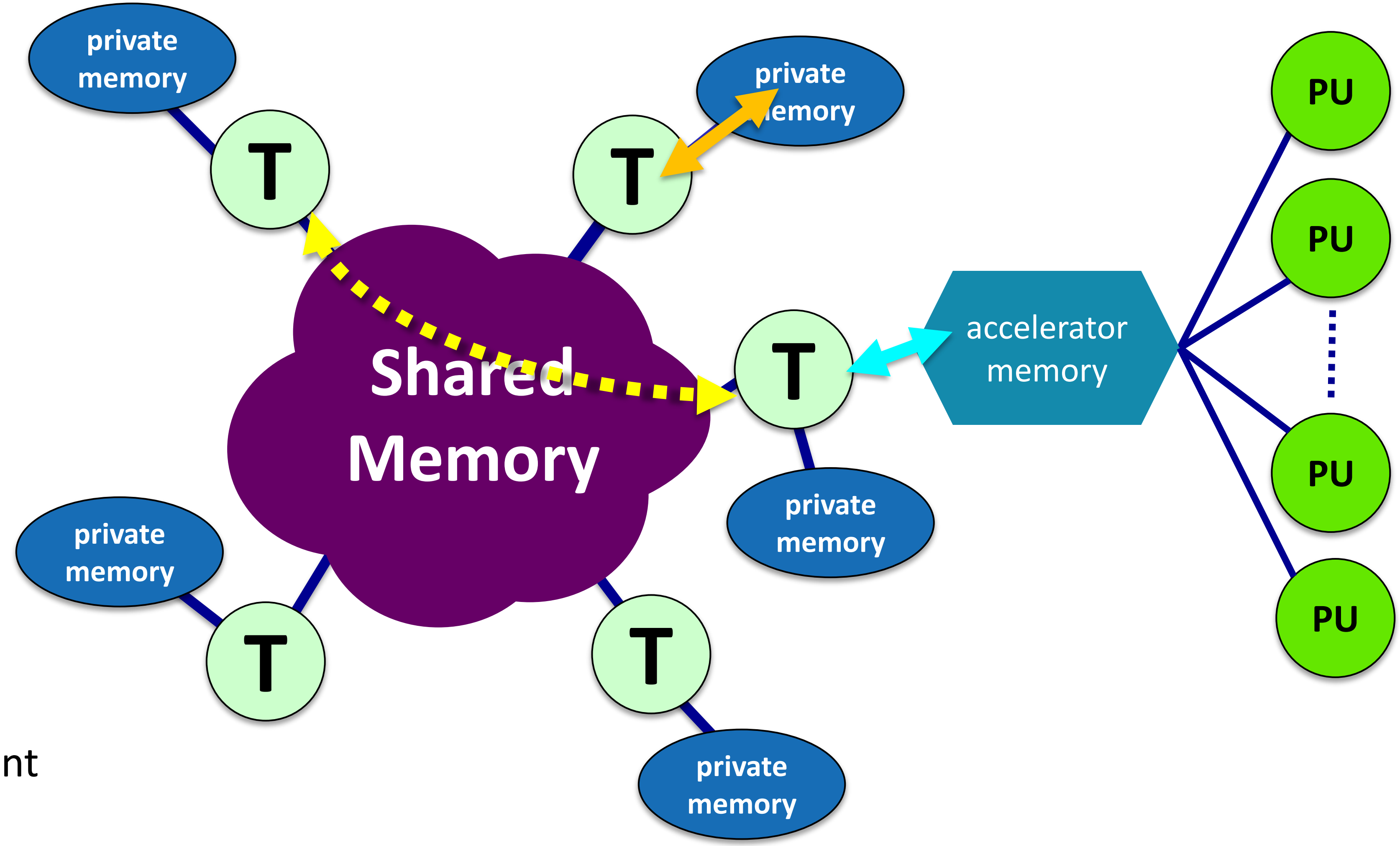
All processors/cores access a shared main memory.

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

Parallelization in OpenMP employs multiple threads.

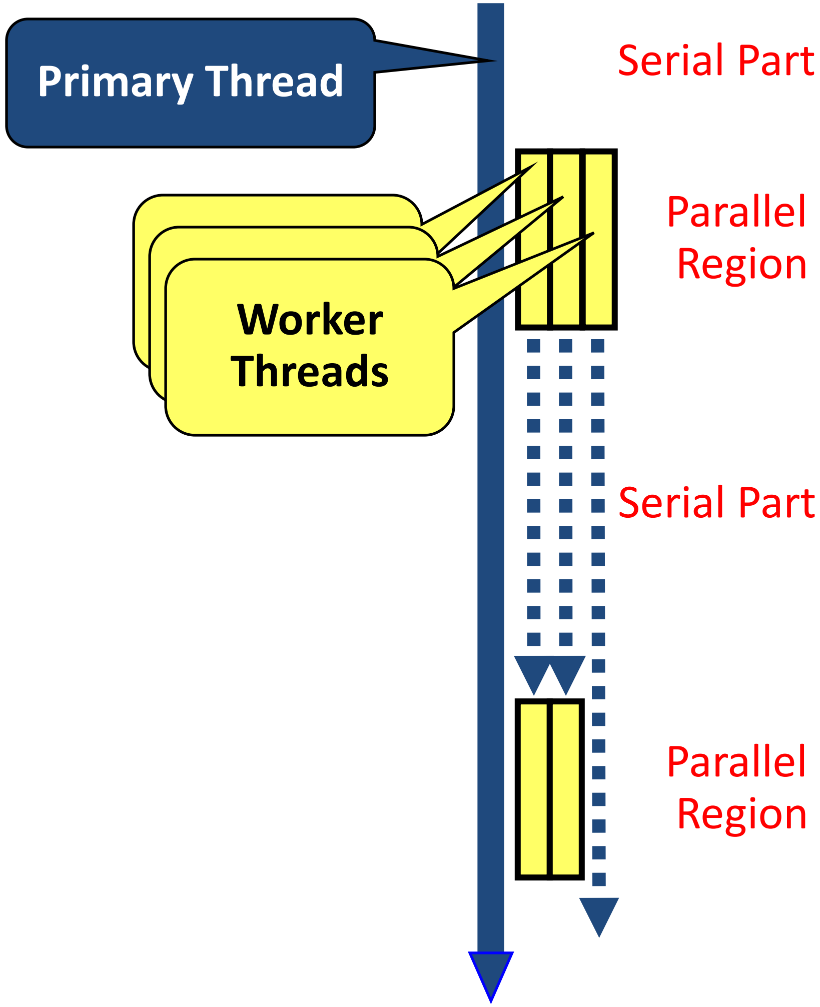
The OpenMP Memory Model

- 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 *Primary Thread*.
- *Worker* threads are spawned at *Parallel Regions*, together with the primary thread 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++

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

Fortran

```
!$omp parallel
    ...
    structured block
    ...
!$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

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
```

Using OpenMP Compilers

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)
  - → AOCC, AOCL, ROCmCC: `-fopenmp`
  - → HPE/Cray CPE: `-homp`
  - → IBM XL: -qsmp=omp
- Switches have to be passed to both compiler and linker:

```
$ gcc [...] -fopenmp -o matmul.o -c matmul.c
$ gcc [...] -fopenmp -o matmul matmul.o
$./matmul 1024
Sum of matrix (serial):   134217728.000000, wall time 0.413975, speed-up 1.00
Sum of matrix (parallel): 134217728.000000, wall time 0.092162, speed-up 4.49
```

Demo

Hello OpenMP World

Worksharing

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++

```
int i;
#pragma omp for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

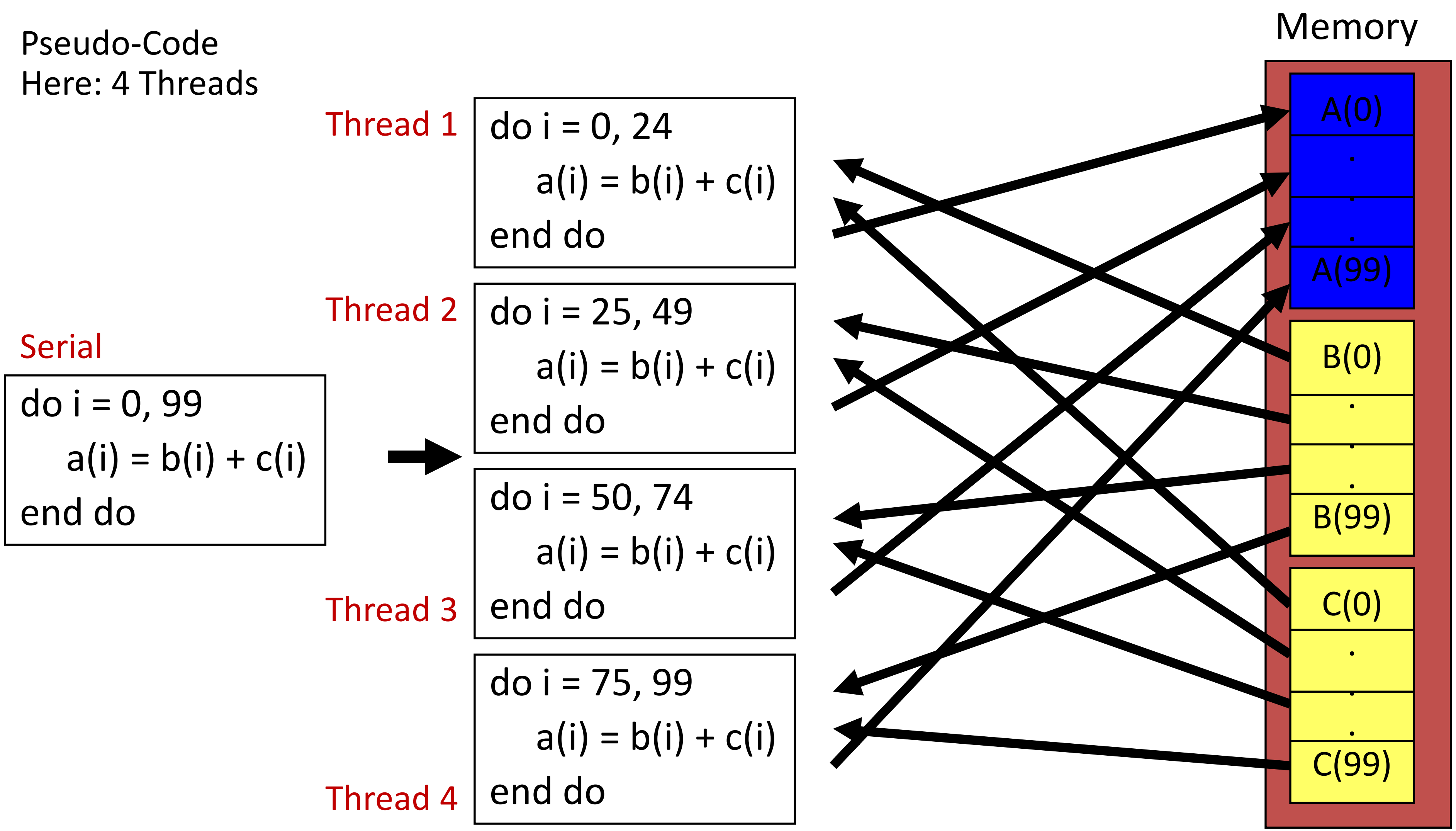
Fortran

```
INTEGER :: i
!$omp do
DO i = 0, 99
   a[i] = b[i] + c[i]
END DO
```

  - 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



Pseudo-Code
Here: 4 Threads

Serial

```
do i = 0, 99
    a(i) = b(i) + c(i)
end do
```

Thread 1

```
do i = 0, 24
    a(i) = b(i) + c(i)
end do
```

Thread 2

```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

Thread 3

```
do i = 50, 74
    a(i) = b(i) + c(i)
end do
```

Thread 4

```
do i = 75, 99
    a(i) = b(i) + c(i)
end do
```

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++

```
#pragma omp single [clause]
... structured block ...
```

Fortran

```
!$omp single [clause]
... structured block ...
!$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 is going to be removed with OpenMP 6.0 (2025)

C/C++

```
~~#pragma omp master[clause]~~
... structured block ...
```

Fortran

```
~~!$omp master[clause]~~
... structured block ...
~~!$omp end master~~
```

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

- Replacement: see the `masked` construct later on.

Demo

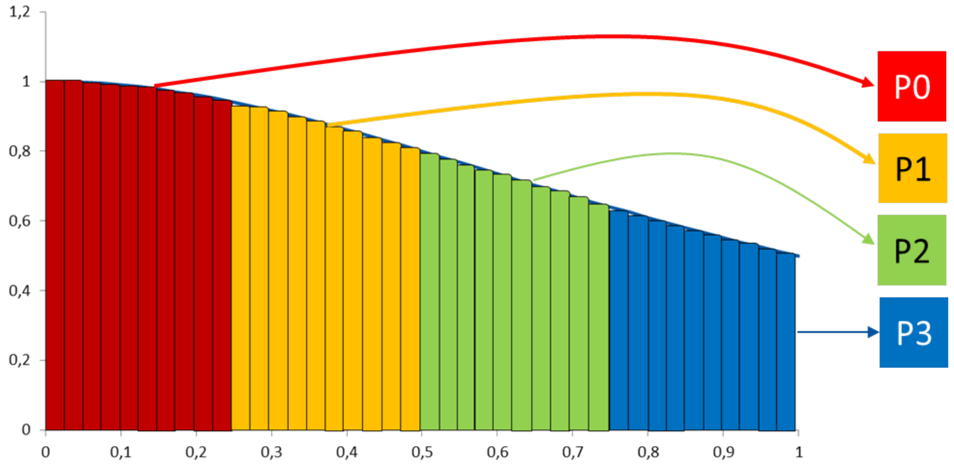
Vector Addition

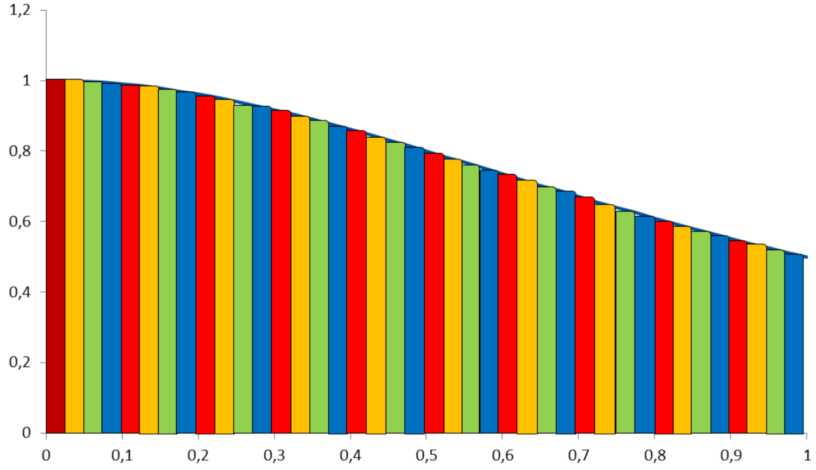
Influencing the For Loop Scheduling / 1

- *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?**
  - → No/low runtime overhead
- **Cons?**
  - → 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];
}
```

- *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++

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

Scoping

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 the variable in the primary thread
  - 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)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

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)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Fortran

```
SAVE INTEGER :: i
!$omp threadprivate(i)
```

Really: try to avoid the use of threadprivate and static variables!

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!

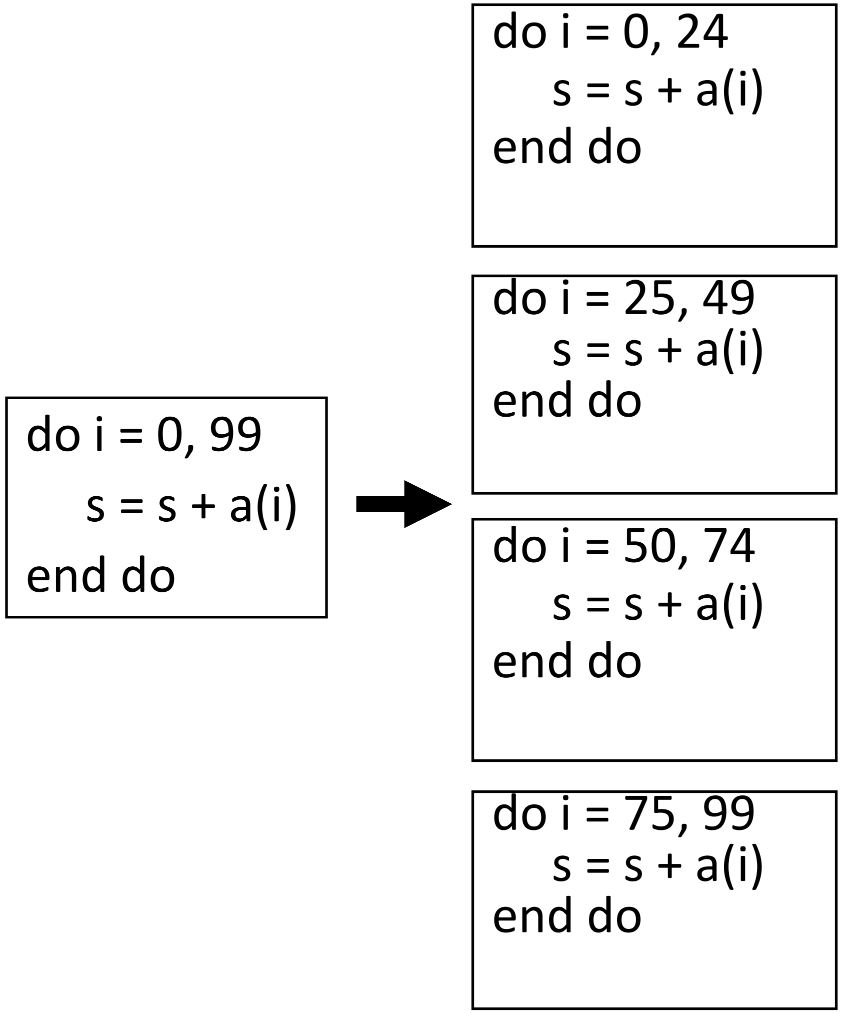
```
#pragma omp parallel
{


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


         s  = s   + a[i];


   }


} // end parallel
```



```
do i = 0, 99
   s = s + a(i)
end do
```

```
do i = 0, 24
   s = s + a(i)
end do
```

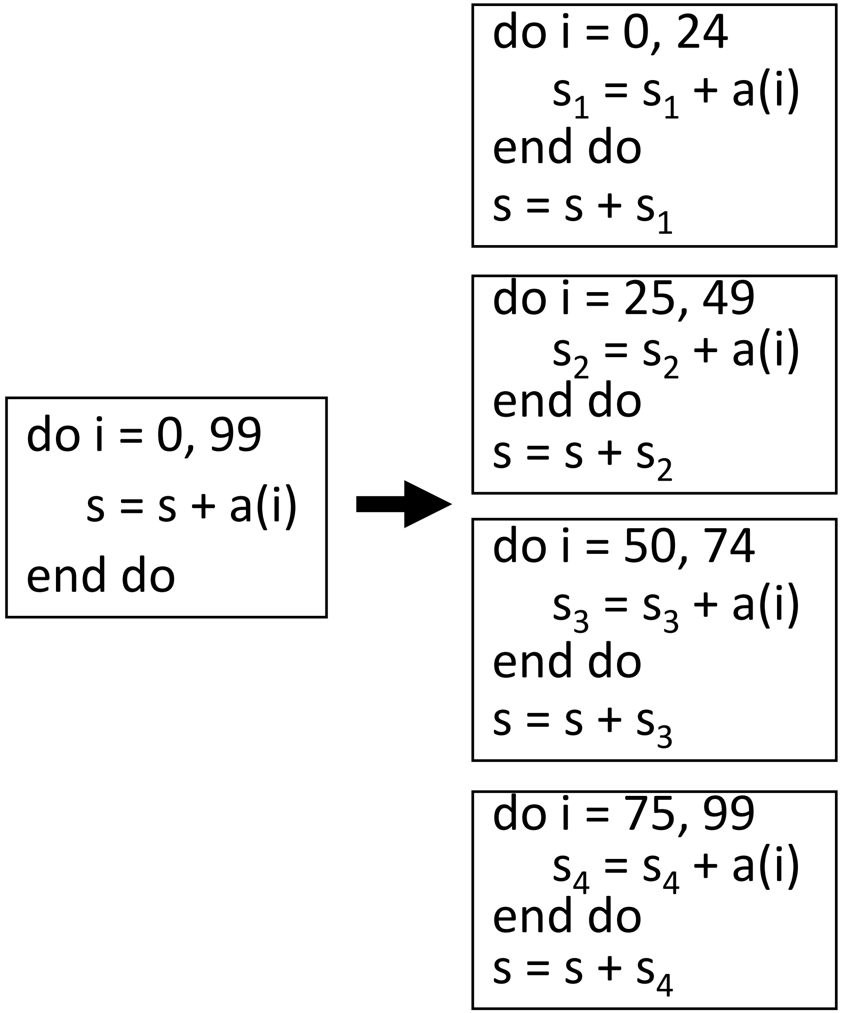
```
do i = 25, 49
   s = s + a(i)
end do
```

```
do i = 50, 74
   s = s + a(i)
end do
```

```
do i = 75, 99
   s = s + a(i)
end do
```

(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
```



```
do i = 0, 99
    $s = s + a(i)$
end do
```

→

```
do i = 0, 24
    $s_1 = s_1 + a(i)$
end do
$s = s + s_1$
```

```
do i = 25, 49
    $s_2 = s_2 + a(i)$
end do
$s = s + s_2$
```

```
do i = 50, 74
    $s_3 = s_3 + a(i)$
end do
$s = s + s_3$
```

```
do i = 75, 99
    $s_4 = s_4 + a(i)$
end do
$s = s + s_4$
```

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];
}
```

  - 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

PI

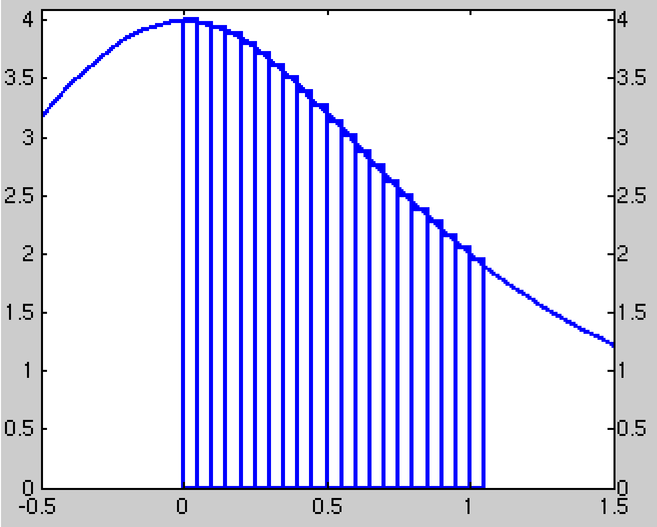
Example: Pi (1/2)

```
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;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



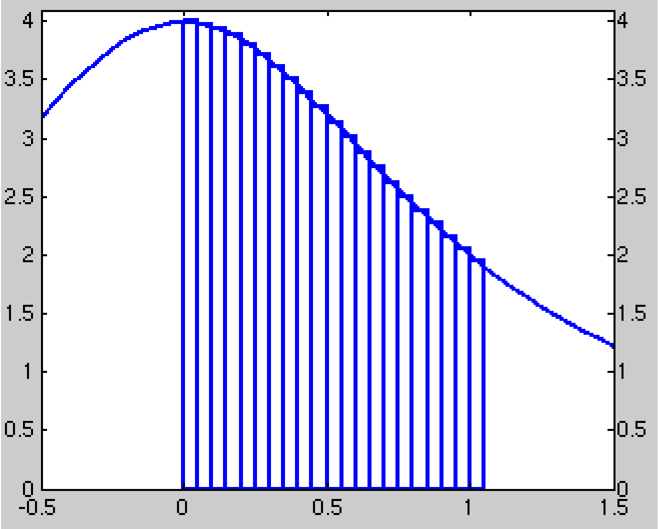
Example: Pi (2/2)

```
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;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



Demo

PI