Parallel and Distributed Systems

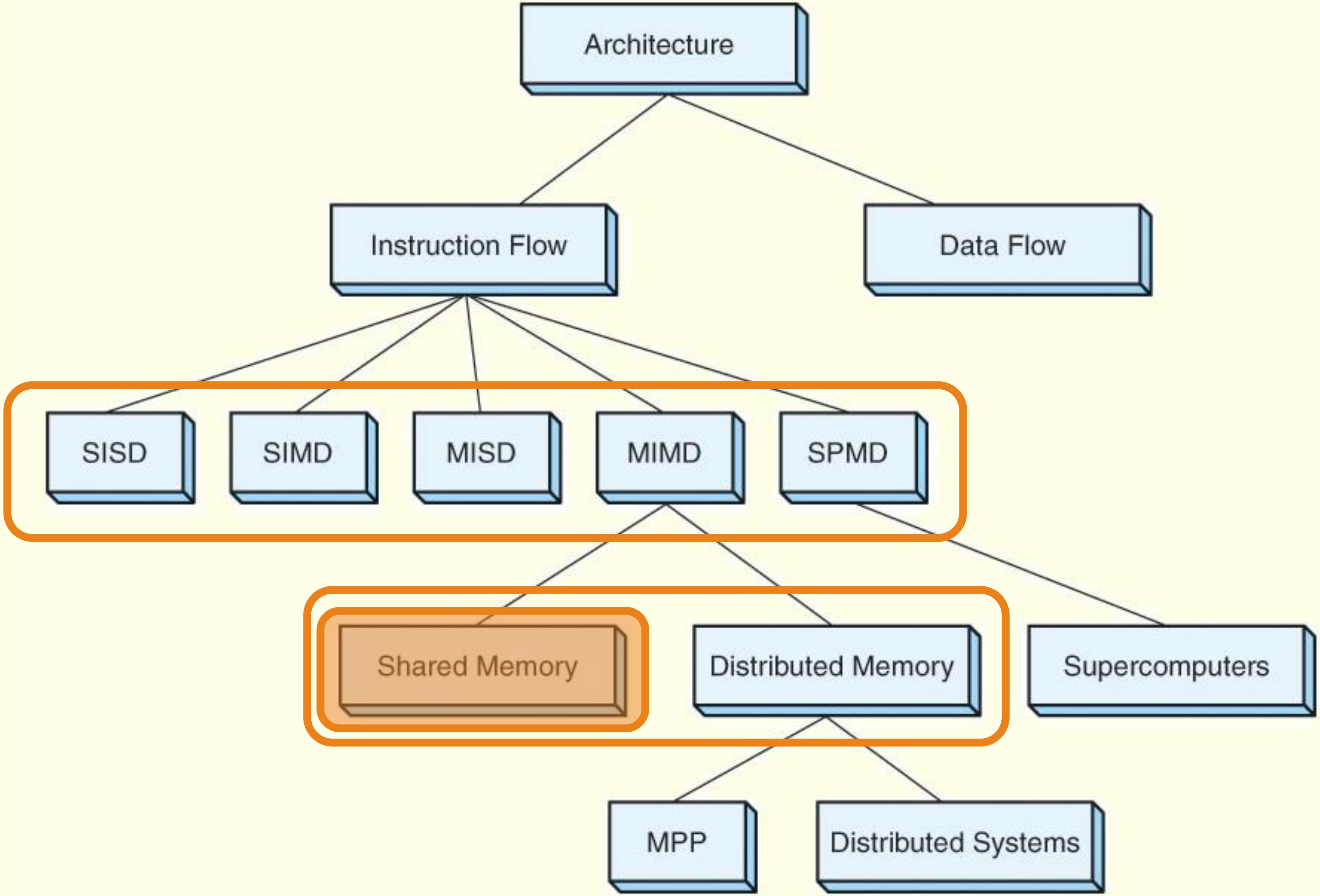
/ 3

Pavel Krömer,
Dept. of Computer Science,
VSB – Technical University of Ostrava

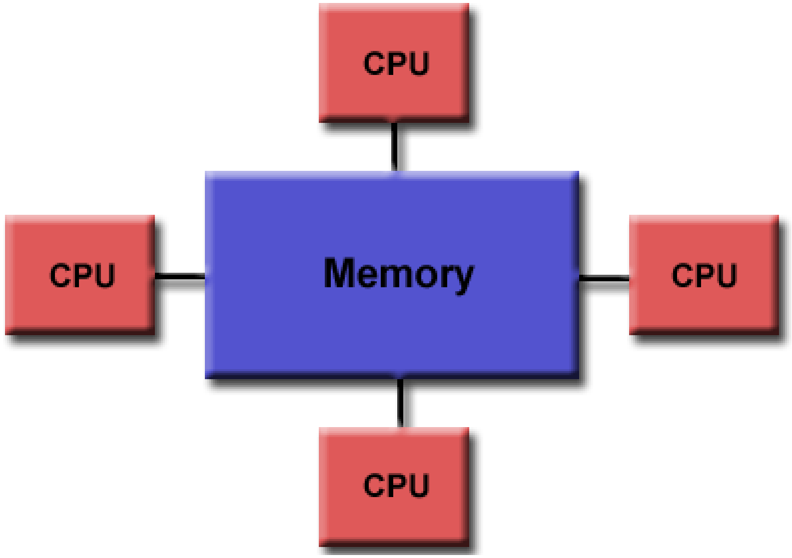
Agenda

- Major topics
  - Shared memory systems, open multiprocessing, OpenMP
- Literature
  - Barbara Chapman, Gabriele Jost, Ruud van der Pas, Using OpenMP Portable Shared Memory Parallel Programming, MIT Press, 2008
  - Peter Pacheco, An Introduction to Parallel Programming, Elsevier, 2011 (Ch. 4, 5)

Shared memory systems


Architecture
Instruction Flow
Data Flow
SISD
SIMD
MISD
MIMD
SPMD
Shared Memory
Distributed Memory
Supercomputers
MPP
Distributed Systems

Shared memory systems



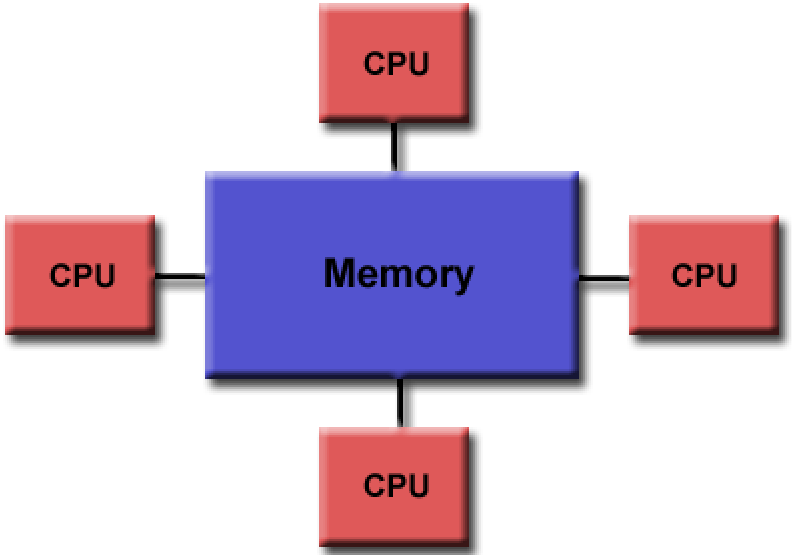
Tightly coupled systems

All processors have access to the complete memory
(as a global [shared] address space)

Processors can operate independently but share memory resources and I/O

Changes in memory caused by one processor are visible to all other

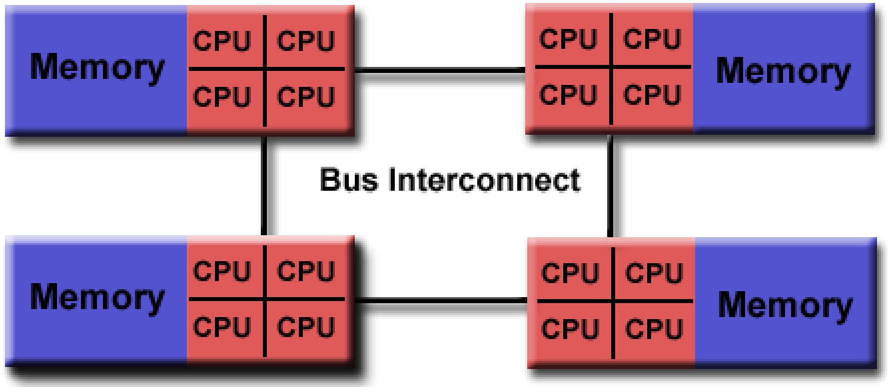
Shared memory systems



Uniform memory access (UMA)

- AKA Symmetric Multiprocessors (SMPs)
- Systems of identical processors with equal access and access times to memory
- Sometimes called Cache Coherent UMA (CC-UMA)
  - if one processor updates a location in shared memory, all the other processors know about the update
  - cache memories that provide access to these variables are kept consistent
  - accomplished at the hardware level (snoopy/sniffy bus protocol)

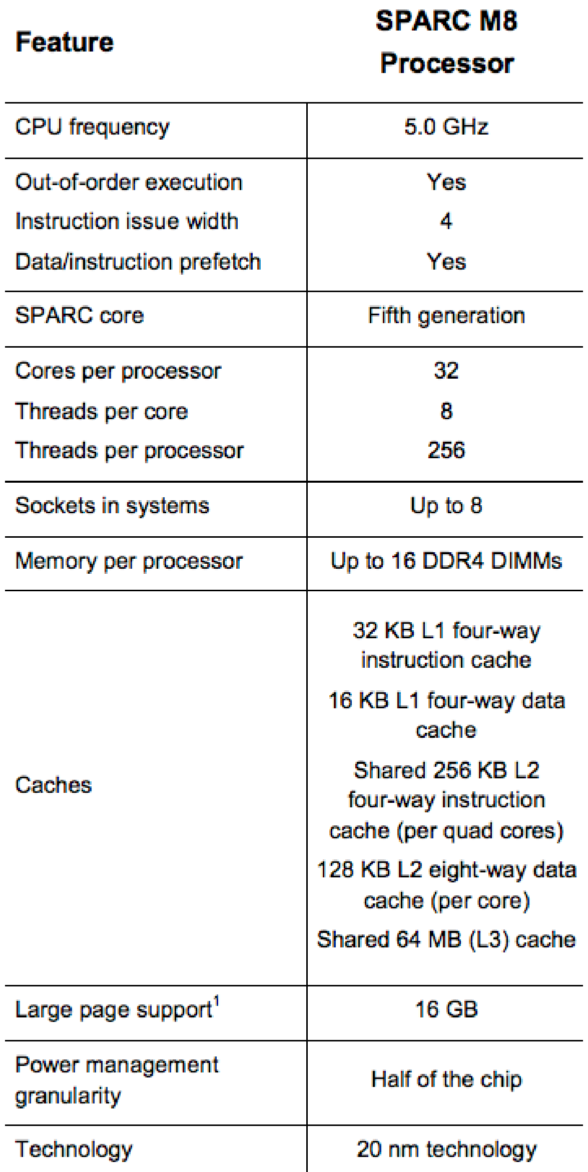
Shared memory systems



Non-uniform memory access (NUMA)

- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- Cache Coherent NUMA (CC-NUMA) if cache coherency is achieved

Shared memory systems



Feature	SPARC M8 Processor
CPU frequency	5.0 GHz
Out-of-order execution	Yes
Instruction issue width	4
Data/instruction prefetch	Yes
SPARC core	Fifth generation
Cores per processor	32
Threads per core	8
Threads per processor	256
Sockets in systems	Up to 8
Memory per processor	Up to 16 DDR4 DIMMs
Caches	32 KB L1 four-way instruction cache 16 KB L1 four-way data cache Shared 256 KB L2 four-way instruction cache (per quad cores) 128 KB L2 eight-way data cache (per core) Shared 64 MB (L3) cache
Large page support[1]	16 GB
Power management granularity	Half of the chip
Technology	20 nm technology

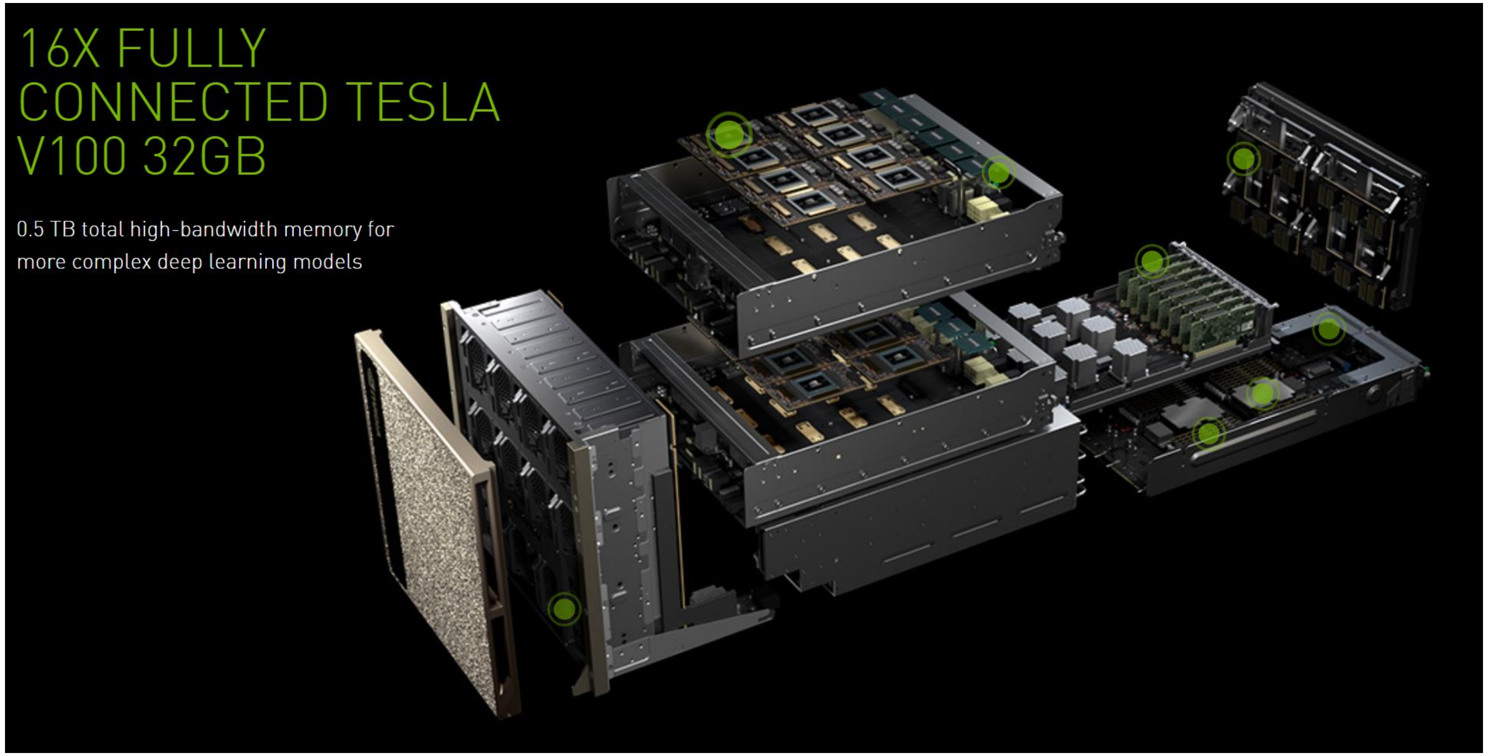
SPARC M8-8 / 2048/ 8TB

SGI UV 3000 SMP system scales up to 256 sockets /x16 cores/ and 64TB of coherent shared memory with industry-standard Intel® Xeon® v3 processors and Linux® O/S

Shared memory systems

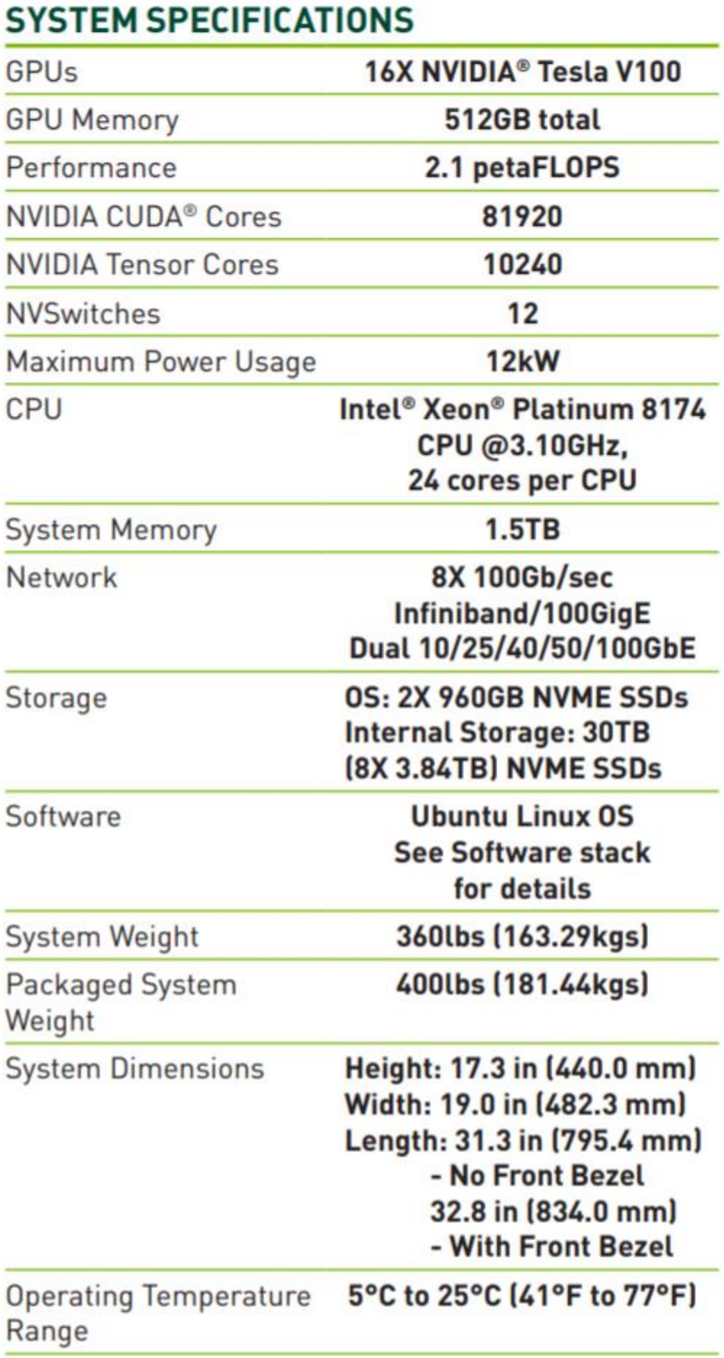
New architectures in town

- Nvidia DGX-2(H)



16X FULLY CONNECTED TESLA V100 32GB

0.5 TB total high-bandwidth memory for more complex deep learning models



SYSTEM SPECIFICATIONS	
GPUs	16X NVIDIA® Tesla V100
GPU Memory	512GB total
Performance	2.1 petaFLOPS
NVIDIA CUDA® Cores	81920
NVIDIA Tensor Cores	10240
NVSwitches	12
Maximum Power Usage	12kW
CPU	Intel® Xeon® Platinum 8174 CPU @3.10GHz, 24 cores per CPU
System Memory	1.5TB
Network	8X 100Gb/sec Infiniband/100GigE Dual 10/25/40/50/100GbE
Storage	OS: 2X 960GB NVME SSDs Internal Storage: 30TB (8X 3.84TB) NVME SSDs
Software	Ubuntu Linux OS See Software stack for details
System Weight	360lbs (163.29kgs)
Packaged System Weight	400lbs (181.44kgs)
System Dimensions	Height: 17.3 in (440.0 mm) Width: 19.0 in (482.3 mm) Length: 31.3 in (795.4 mm) - No Front Bezel 32.8 in (834.0 mm) - With Front Bezel
Operating Temperature Range	5°C to 25°C (41°F to 77°F)

Shared memory systems

Pros and cons

- Advantages
    - Global address space provides a user-friendly programming access to memory
    - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages
    - Lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management
    - Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Programming shared memory systems

Shared memory systems

OpenMP®
Enabling HPC since 1997

Basic considerations

- ability to execute data vs. task parallel programs
- SIMD vs. MIMD (wrt. SPMD)

High-level strategies

- use of some low or high-level mechanism for multi-process and/or multi-thread parallel computation
- communication usually (but not exclusively) through shared memory
- open multiprocessing (OpenMP)

platform/language/d s

Charm++
parallel programming framework

Numba RAY Golang C++ 11

Intel® Threading Building Blocks

Open Multiprocessing

Industry standard API for C/C++ and Fortran shared memory parallel programming

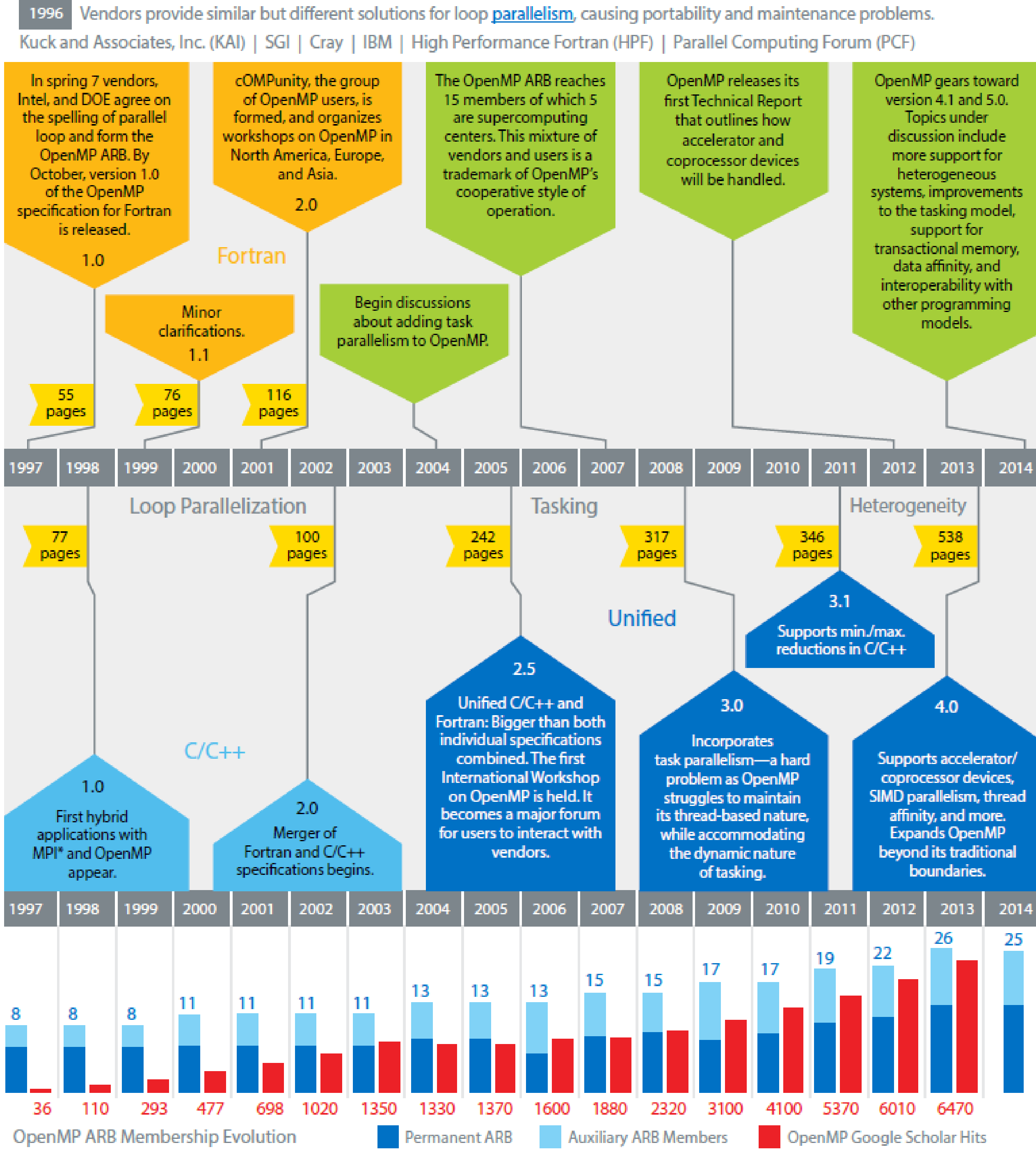
- governed by OpenMP Architecture Review Board
- major HW/SW and compiler vendors (Intel, ~~PGI~~ NVidia, IBM, AMD, Cray, Oracle, . . .)

Multiple versions

- 1.0 (Fortran ’97, C ’98) - 3.1 (2011) shared memory
- 4.0 (2013) accelerators, NUMA
- 4.5 (2015) improved memory mapping, SIMD
- 5.0 (2018) improved accelerator support

Issues dealt with

- Fortran/C++
- accelerators, offloading to device



1996 Vendors provide similar but different solutions for loop parallelism, causing portability and maintenance problems.

Kuck and Associates, Inc. (KAI) | SGI | Cray | IBM | High Performance Fortran (HPF) | Parallel Computing Forum (PCF)

In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released. 1.0

Minor clarifications. 1.1

cOMPunity, the group of OpenMP users, is formed, and organizes workshops on OpenMP in North America, Europe, and Asia. 2.0

Begin discussions about adding task parallelism to OpenMP.

The OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP’s cooperative style of operation.

OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.

OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.

1.0 First hybrid applications with MPI* and OpenMP appear.

2.0 Merger of Fortran and C/C++ specifications begins.

2.5 Unified C/C++ and Fortran: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.

3.0 Incorporates task parallelism—a hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.

3.1 Supports min./max. reductions in C/C++

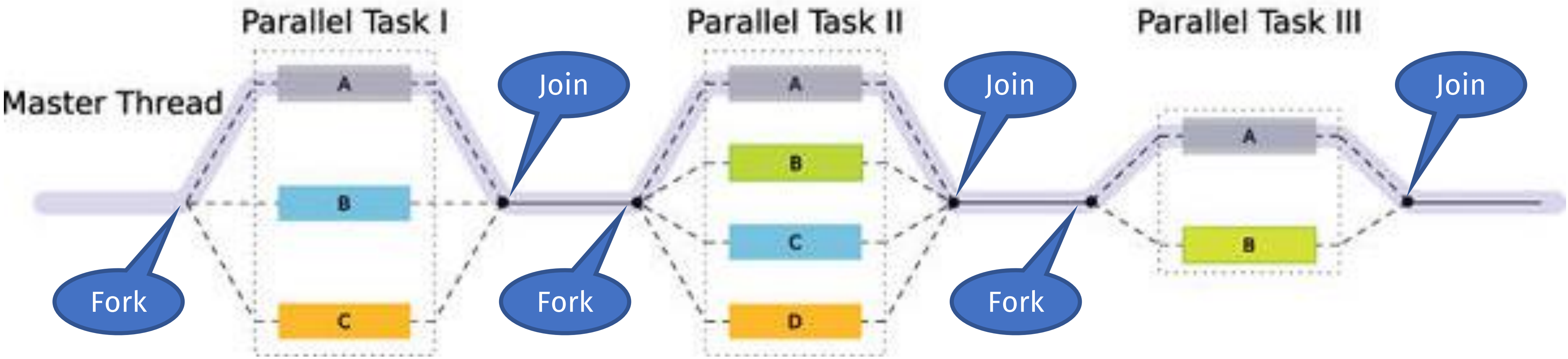
4.0 Supports accelerator/coprocessor devices, SIMD parallelism, thread affinity, and more. Expands OpenMP beyond its traditional boundaries.

OpenMP ARB Membership Evolution

Execution model: fork-join

Fork-join programming model

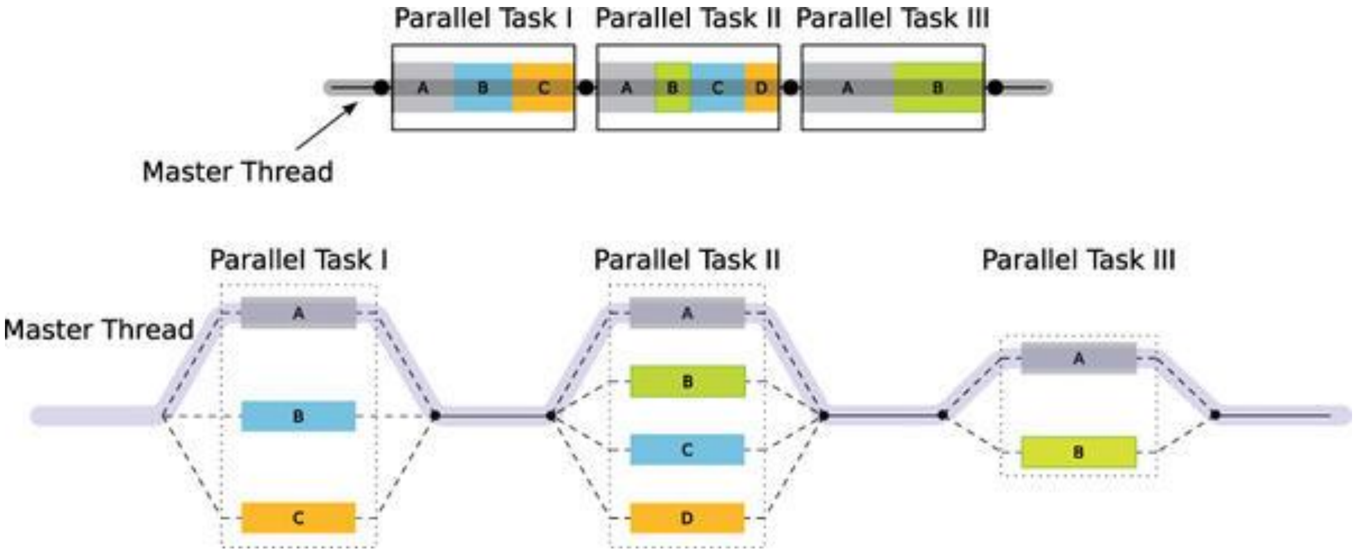
- one master thread that executes all serial regions
- master forks new worker threads at the beginning of parallel regions
- parallel threads share the work and sync at the end parallel regions
- each thread works with shared and private variables (for convenience, all in the global shared address space)



Implementation: directive-based

The programmer specifies what, the compiler decides how

- best practices and patterns
- automation and optimization
- portability
- single source for sequential/parallel code



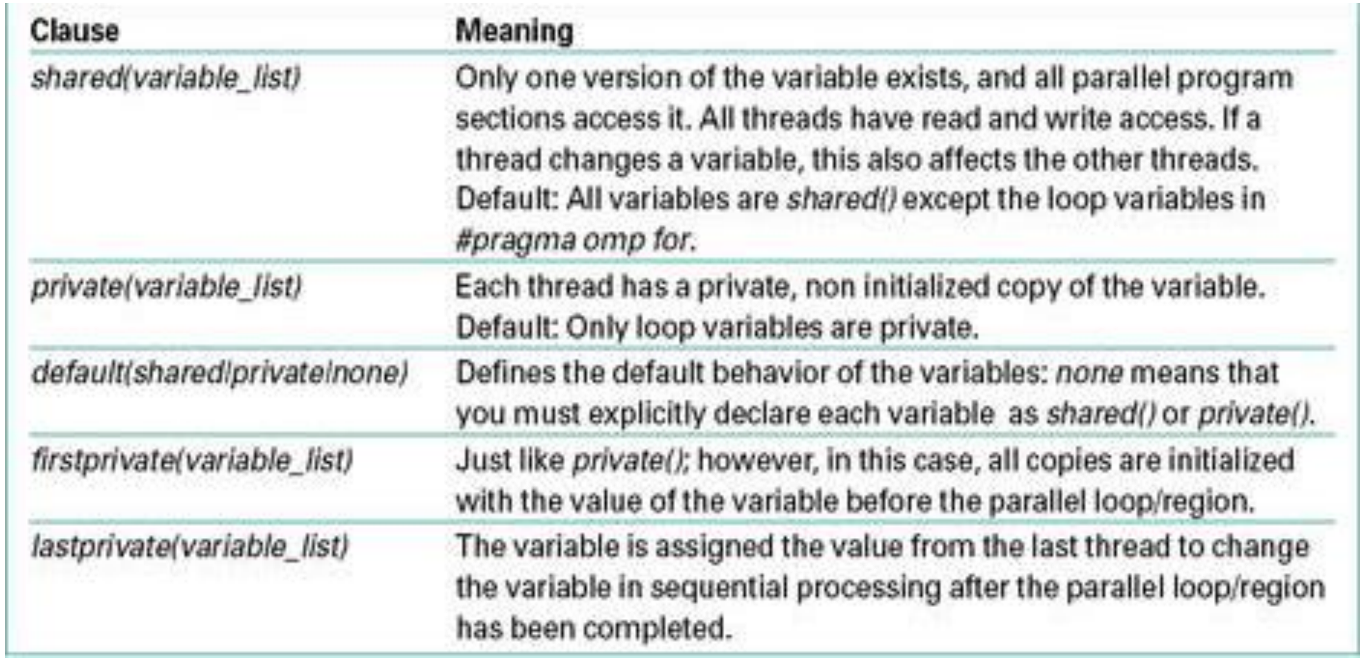
An OpenMP program consists of

- compiler directives and clauses (#pragma omp parallel)
- library functions (omp_get_num_threads())
- environment variables (OMP_NUM_THREADS)

Implementation: directive-based

Clauses of parallel directives specify

- conditional parallelization
  - to determine if the parallel construct results in creation/use of threads if (scalar-expression)
- degree of concurrency
  - to explicitly specify the number of threads created/used num_threads(integer-expression)
- data handling
  - to indicate variable scope (local, global, or 'special')
    private(variable-list)
    shared(variable-list)
    firstprivate(variable-list)
    default(shared j none)



Clause	Meaning
shared(variable_list)	Only one version of the variable exists, and all parallel program sections access it. All threads have read and write access. If a thread changes a variable, this also affects the other threads. Default: All variables are *shared()* except the loop variables in *#pragma omp for*.
private(variable_list)	Each thread has a private, non initialized copy of the variable. Default: Only loop variables are private.
default(shared|private|none)	Defines the default behavior of the variables: *none* means that you must explicitly declare each variable as *shared()* or *private()*.
firstprivate(variable_list)	Just like *private()*; however, in this case, all copies are initialized with the value of the variable before the parallel loop/region.
lastprivate(variable_list)	The variable is assigned the value from the last thread to change the variable in sequential processing after the parallel loop/region has been completed.

The parallel directive

Indicates a parallel region

- creates a group of threads (OMP_NUM_THREADS, omp_set_num_threads(nthreads))
- each thread executes the structured block of code (possibly the same code!)

Examples

```
# pragma omp parallel if (is_parallel == 1) num_threads(8)  private(a) shared(b) firstprivate(c)
```

- if the value of variable is_parallel is one, eight threads are used • each thread has private copy of a and c, but all share one copy of b
- the value of each private copy of c is initialized to value of c before the parallel region

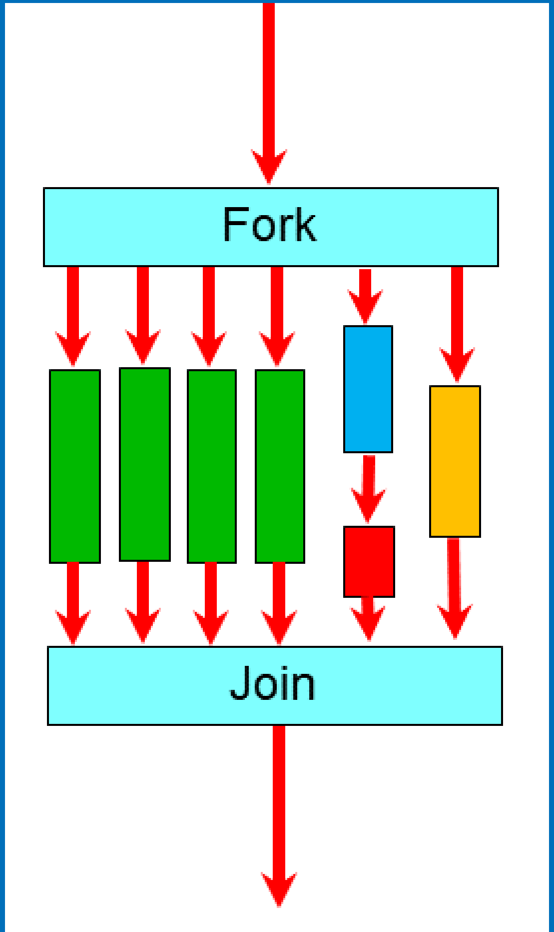
```
# pragma omp parallel reduction(+ : sum) num_threads(8) default(private)
```

- eight threads get a copy of the variable sum
- when threads exit, the values of these local copies are accumulated into the sum variable on the master thread – other reduction operations include *, -, &, |, ^, &&, || • all variables are private unless otherwise specified

The parallel directive

Examples (cont.)

```
#include <omp.h>
…

int main()
{
    int tid, nthreads;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf(”Hello World from thread %d\n", tid);
        #pragma omp barrier
        if ( tid == 0 )
        {
            nthreads = omp_get_num_threads();
            printf(”Total threads= %d\n",nthreads);
        }
    }
}
```



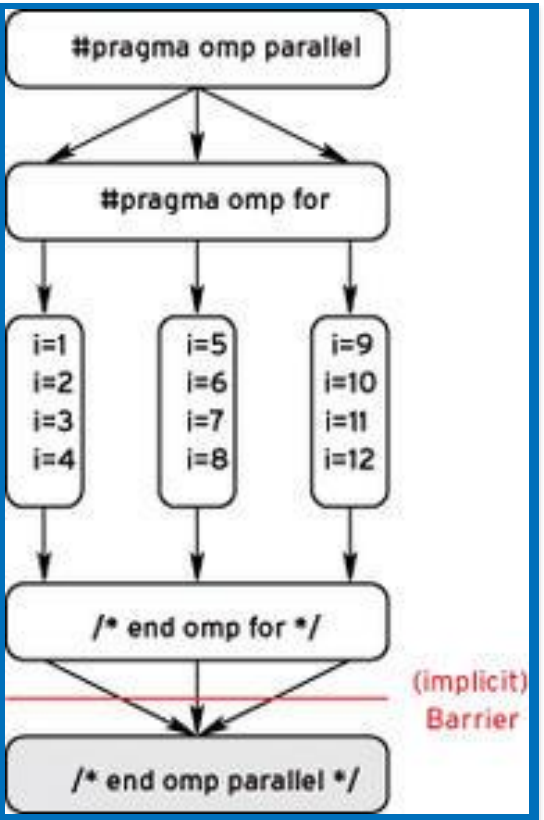
The for work-sharing directive

Solves a typical problem – parallelization of a for loop

- requirement: independent iterations
- the loop index automatically assumed private
- extra code reduced to only two directives plus sequential code (code is easy to read/maintain)
- implicit synchronization at the end of the loop (can be overriden by the nowait clause)
- Very often merged together with parallel
  - #pragma omp parallel for

Example

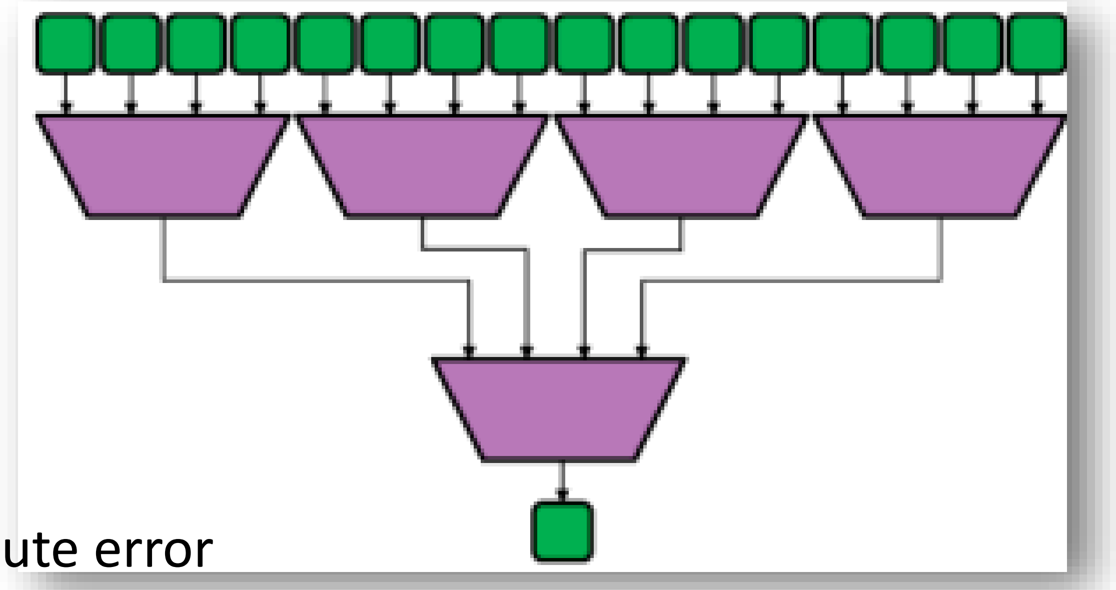
```
#include <cmath>
…
int main()
{
   const int size = 256;
   double sinTable[size];

   #pragma omp parallel
   {
            …
            #pragma omp for
            for(int n=0; n<size; ++n)
               sinTable[n] = std::sin(2 * M_PI * n / size);
            …
   }
}
```



The reduce clause

Solves another typical problem – reduction

- a well-known parallel pattern in shared memory programming



- combines all the elements in a collection into one using an associative two-input, one-output operator
- reductions are used in many algorithms to compute error metrics and termination conditions for iterative algorithm

A more complex example

Parallel, for and reduction working together

Examples (cont.)

```
#include <omp.h>
…
main ()
{
        int i, n, chunk;
        float a[100], b[100], result;
        n = 100; chunk = 10; result = 0.0;

        for (i=0; i < n; i++)
        {
                a[i] = i * 1.0;
                b[i] = i * 2.0;
        }
        #pragma omp parallel for default(shared) private(i) schedule(static,chunk)
        reduction(+:result)
        for (i=0; i < n; i++)
                result = result + (a[i] * b[i]);
}
```

Pros and cons of OpenMP

Advantages

- simple programming model
- single source code for serial and parallel version
- portable and well-supported (gcc)
- code works in serial without adjustments

Disadvantages

- can only be run in shared memory computers
- requires a compiler that supports OpenMP
- high risk of race conditions