Parallel Programming

Part 3:
Programming for Shared Memory Multiprocessor Architectures with OpenMP

foils by M. Gerndt, T. Fahringer, Michael Klemm, Tim Mattson, Yang-Suk Kee, Christian Terboven, Peter Thoman

OpenMP

- Portable programming of shared memory systems.
- It is a quasi-standard.
- API for Fortran and C/C++
  - directives (commands to the compiler, not hints)
  - runtime routines
  - environment variables
- www.openmp.org

Example

Program

```
main(){
    #pragma omp parallel
    {
     printf("Hello world");
    }
}
```

Compilation

```
> gcc –O2 –fopenmp openmp.c
```

Execution

```
> export OMP_NUM_THREADS=2
> ./a.out
Hello world
Hello world
```

```
> export OMP_NUM_THREADS=3
> ./a.out
Hello World
Hello World
Hello World
```

Goals

- Standardization
  - Provide a standard among a variety of shared memory architectures (platforms)
  - High-level interfaces to thread programming
- Lean and Mean
  - An increasingly complex set of directives for shared address space programming
  - Just 20 directives are enough to represent significant parallelism

Hello World Program: Pthread Version

```
#include <pthread.h>
#include <stdio.h>

void* thrfunc(void* arg)
{
   printf("hello from thread %d\n", *(int*)arg);
}
```

```
int main(void)
{
   pthread_t thread[4];
   pthread_attr_t attr;
   int arg[4] = {0,1,2,3};
   int i;

   // setup joinable threads with system scope
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   // create N threads
   for(i=0; i<4; i++)
      pthread_create(&thread[i], &attr, thrfunc, (void*)&arg[i]);
   // wait for the N threads to finish
   for(i=0; i<4; i++)
      pthread_join(thread[i], NULL);
}
```

Hello World: OpenMP Version

```
#include <omp.h>
#include <stdio.h>

int main(void)
{
   #pragma omp parallel
   printf("hello from thread %d\n", omp_get_thread_num());
}
```

Goals (cont.)

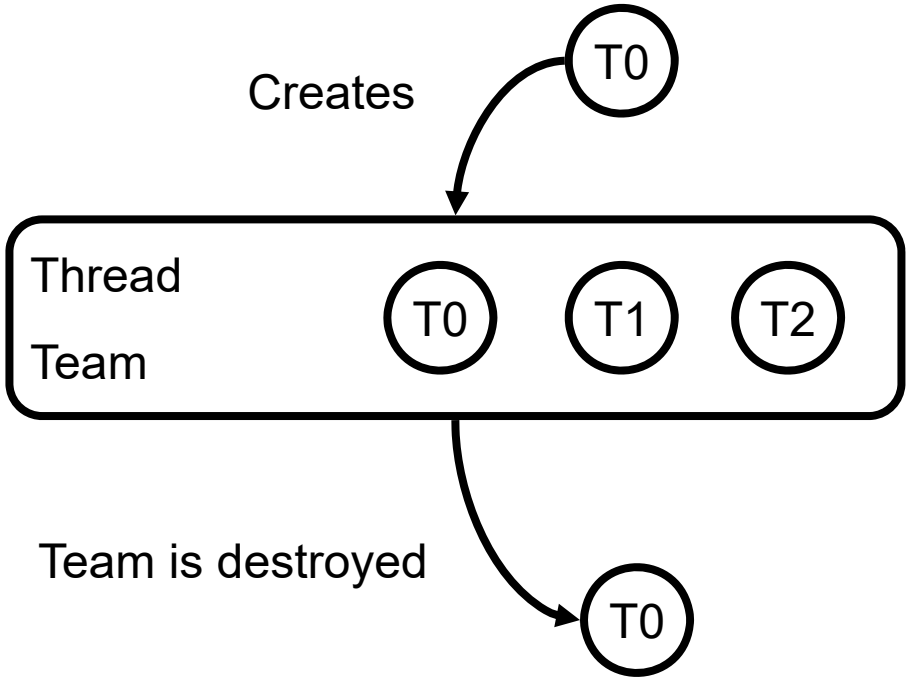
- Ease of use
  - Incrementally parallelize a serial program
    - Unlike all or nothing approach of message-passing
  - Implement both coarse-grain and medium-grain parallelism
- Portability
  - Fortran (77, 90, and 95), C, and C++
  - Public forum for API and membership

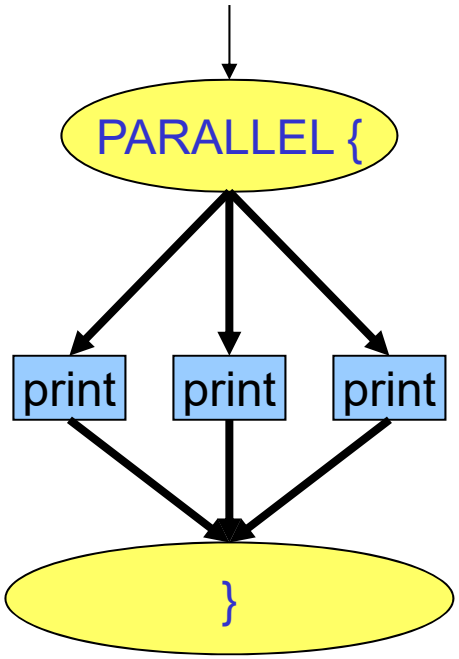
Programming Model

- Thread-based Parallelism
  - A shared memory process with multiple threads
  - Based upon multiple threads in the shared memory programming paradigm
- Explicit Parallelism
  - Explicit (not automatic) programming model
  - Offer the programmer some control over parallelization
    - Less control over locality

Execution Model

```
#pragma omp parallel
{
 printf("Hello world %d\n", omp_get_thread_num());
}
```

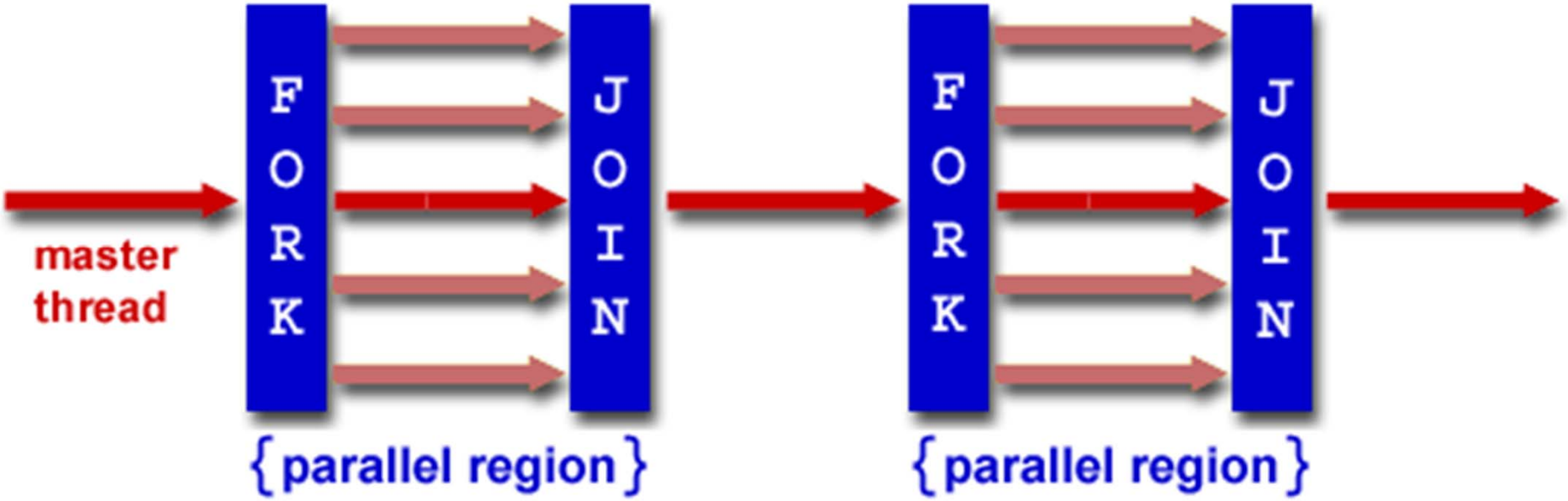




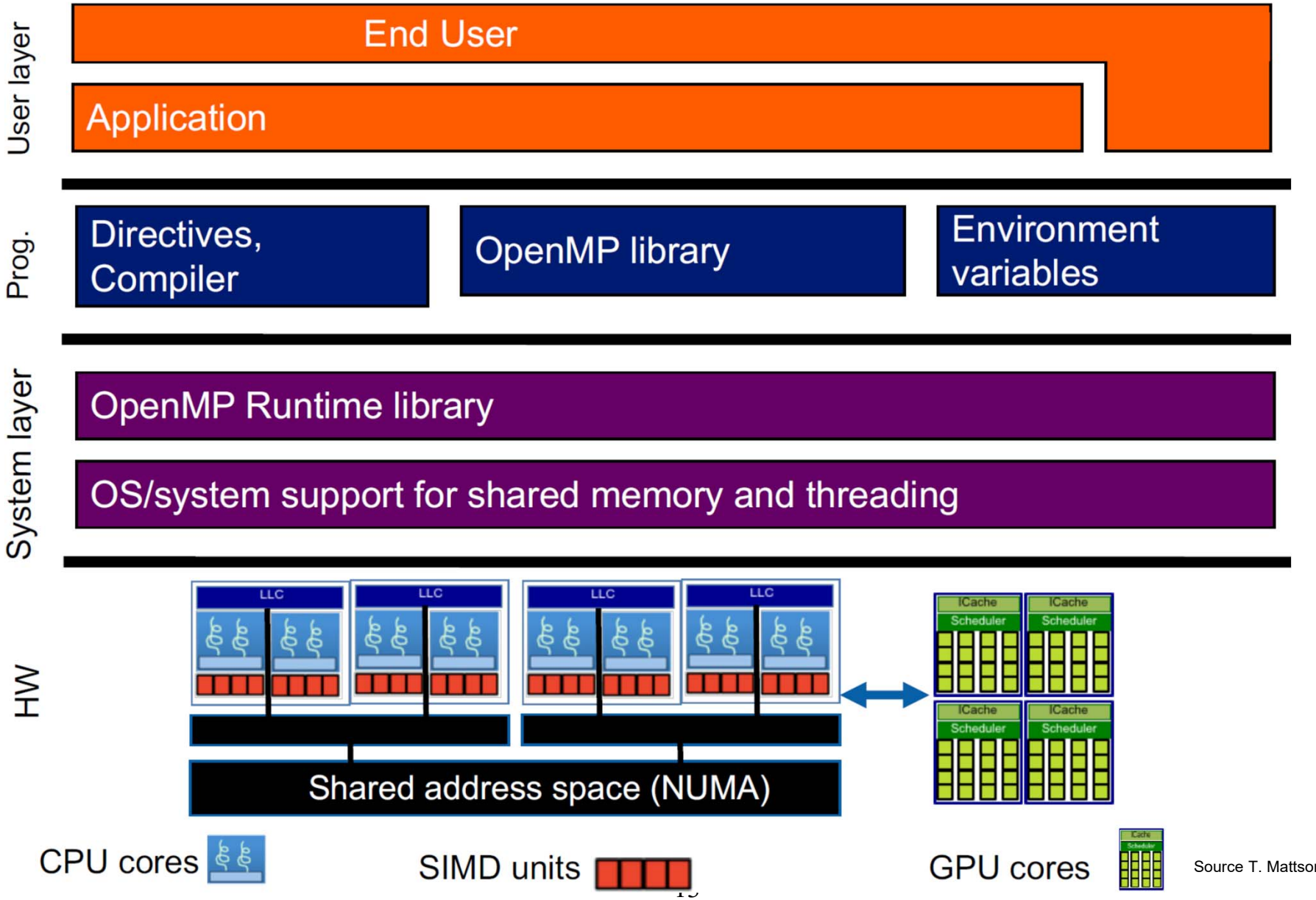
Fork/Join Execution Model

- An OpenMP-program starts as a single thread (master thread).
- Additional threads (Team) are created when the master hits a parallel region.
- When all threads finished the parallel region, the new threads are given back to the runtime or operating system.
- A team consists of a fixed set of threads executing the parallel region simultaneously.
- All threads in the team are synchronized at the end of a parallel region via a barrier.
- The master continues after the parallel region.

Fork-Join Model



OpenMP HW/SW Stack



Source T. Mattson, Intel

Syntax of Directives and Pragmas

C / C++

#pragma omp *directive name* [*clause list*]

```
int main()  {
  #pragma omp parallel default(shared)
  {
    printf("hello world\n");
  }
}
```

Fortran

!$OMP *directive name* [*parameters*]

```
!$OMP PARALLEL DEFAULT(SHARED)
        write(*,*) 'Hello world'
!$OMP END PARALLEL
```

Directives

Directives can have continuation lines

- C

```
#pragma omp parallel private(i) \
    private(j)
```

- Fortran

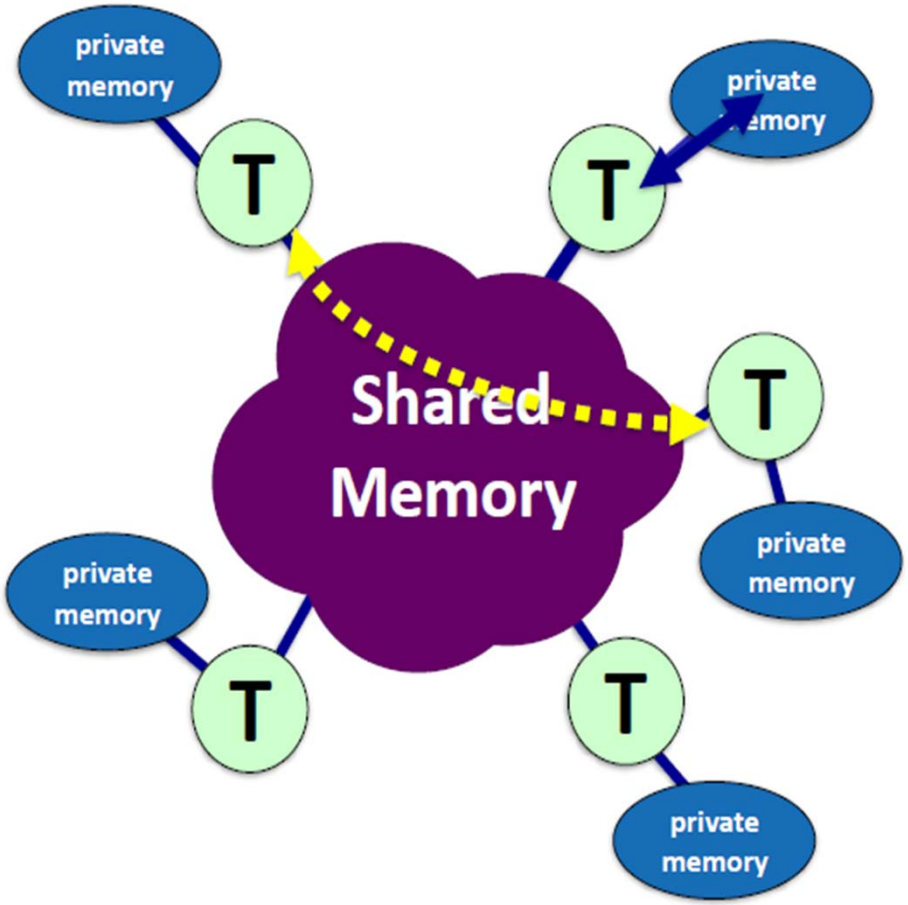
```
!$OMP directive name first_part &
!$OMP continuation_part
```

OpenMP Components

Directives	Runtime functions	Environment variables
◆ Worksharing	◆ Thread Management	◆ Thread Settings
◆ Tasking	◆ Work Scheduling	◆ Thread Controls
◆ Affinity	◆ Tasking	◆ Work Scheduling
◆ Accelerators	◆ Affinity	◆ Affinity
◆ Cancellation	◆ Accelerators	◆ Accelerators
◆ Synchronization	◆ Cancellation	◆ Cancellation
	◆ Locking	◆ Operational

Source: Ruud van der Pas, Oracle

OpenMP Memory Model



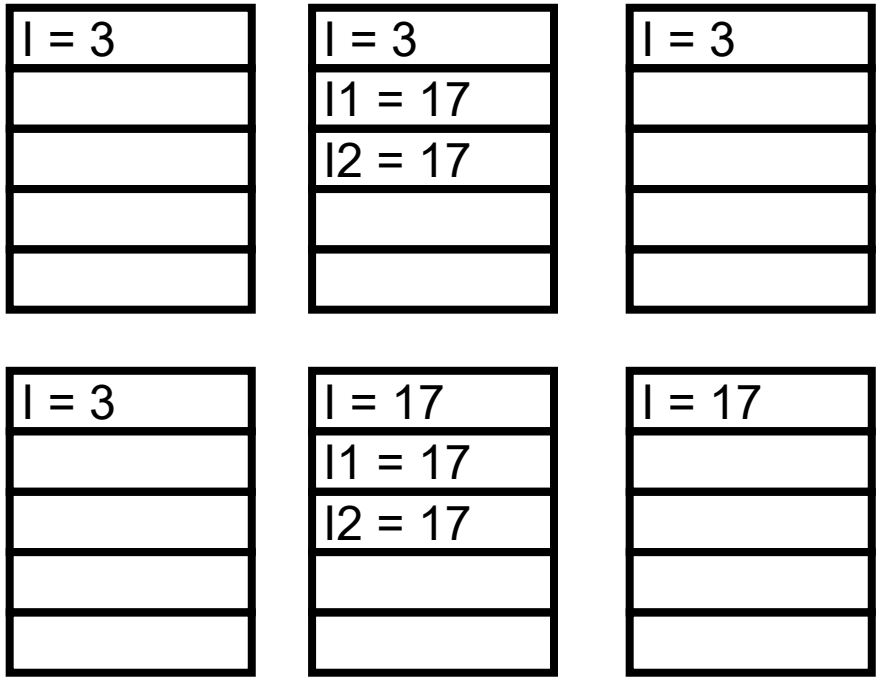
- All threads have access to the same, globally shared memory.
- Data in private memory is only accessible by the thread owning this memory.
- No other thread sees the change(s) in private memory.
- By default variables are shared.
- Data transfer is through shared memory and is 100 % transparent to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

Source: C. Terboven, RWTH Aachen

Example: Private Data

```
I=3;
#pragma omp parallel private(I)
{
 I=17;
}
Printf("Value of I=%d\n", I);
```

Do not code like this!

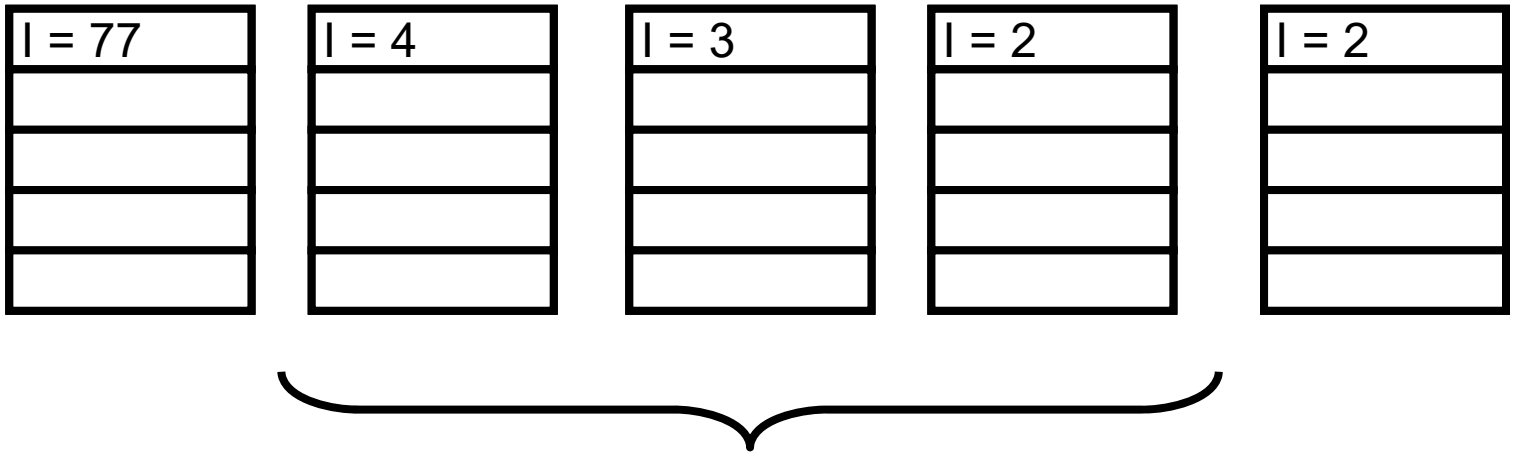


Private Data

- A new copy is created for each thread.
- One thread may reuse the global shared copy.
- The private copies are destroyed after the parallel region.
- The value of the global shared copy is undefined after the parallel region. It is undefined whether it is modified inside of the parallel region or not.

Example: Shared Data

```
I=77
#pragma omp parallel shared(I)
{
 I=omp_get_thread_num();
}
Printf("Value of I=%d\n", I);
```



Default Storage Attributes

- Shared memory programming model
  - Most variables are shared by default
- Global variables are shared among threads
  - C: file scope variables, static variables
  - Dynamically allocated memory (allocate, malloc, new)
- But not everything is shared
  - Stack variables in C functions called from parallel regions are PRIVATE
- Variables declared inside of OMP region are by default private if not explicitly declared as shared.
- Variables declared outside of OMP region are by default shared if not explicitly declared as private.

Source T. Mattson, Intel

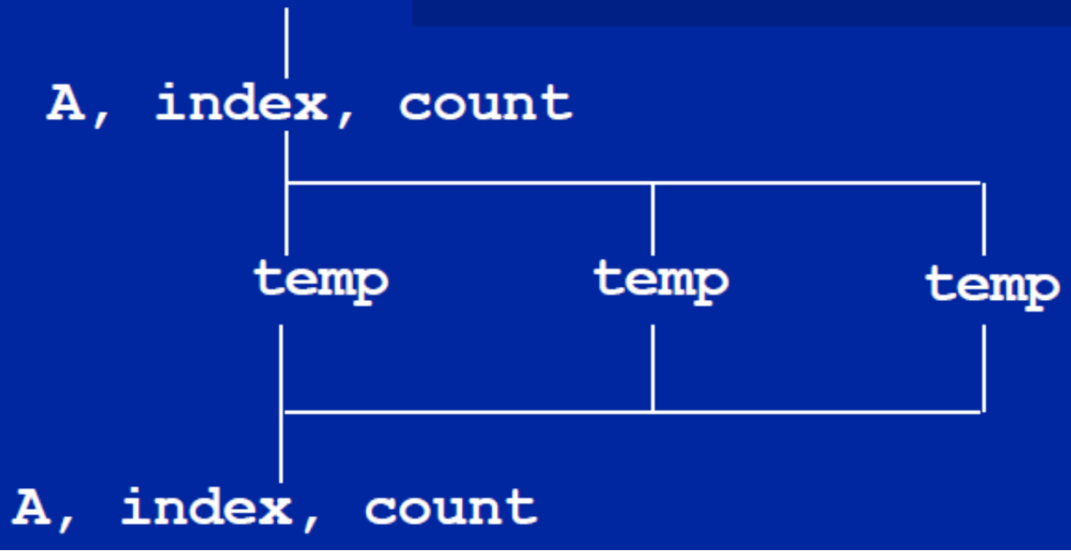
Data sharing examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
      work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Source T. Mattson, Intel

OpenMP Memory Model

- OpenMP supports relaxed-consistency shared memory model.
  - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
  - Temporary views are made consistent only at certain points in the program.
  - #pragma omp flush operation enforces consistency.

OpenMP Flush Operation

- #pragma omp flush [(list)]
- Flush set:
  - all thread visible variables for a flush construct without an argument list.
  - a list of variables when the „flush(list)“ construct is used.
- Flush defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the flush set.
  - all previous read/writes by this thread have completed and writes are written back to main memory
  - Flush only guarantees consistency between operations within the executing thread and main memory.
    - Other threads may have to include flush as well to see writes of other threads.
  - no subsequent read/writes (after flush) by this thread have occurred
  - flushes with overlapping flush sets cannot be reordered.

```
double A;
A=compute();

#pragma omp flush(A)
```

A can be committed to memory as soon as here.

or as late as here

Implicit Flush Operations

among others:

- in barriers or whenever a lock is set or unset
- at entry to and exit from
  - parallel, parallel worksharing, critical, ordered regions
  - critical regions
- at exit of worksharing regions (unless nowait is specified)
- at entry to and exit from atomic - flush set is the address of the variable atomically updated

Moving Data Between Threads

- To move the value of a shared variable V from thread 0 to thread 1, do the following in exactly this chronological order:
  1. Write V on thread 0
  2. Flush V on thread 0
  3. Flush V on thread 1
  4. Read V on thread 1

- Write of V by thread 0 is guaranteed visible and valid on thread 1.

Flush Example: producer consumer

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

```
while (!flag);
b=a;
```

- This is incorrect code.
- Compiler and/or HW may re-order the reads/writes to *a* and *flag*, or *flag* may be held in a register.
- Use flush directive to make this code work.

Thread 0

```
a = foo();
#pragma omp flush(a)
flag = 1;
#pragma omp flush(flag)
```

- First flush ensures *flag* is written after *a*.
- Second flush ensures *flag* is written to memory.

Thread 1

```
#pragma omp flush(flag)
while (!flag){
#pragma omp flush(flag)
}
#pragma omp flush(a)
b=a;
```

- First and second flushes ensure *flag* is read from memory.
- Third flush ensures a is read from memory.

Thread 0

```
a = foo();
#pragma omp flush(a)
flag = 1;
#pragma omp flush(flag)
```

Thread 1

```
#pragma omp flush(flag)
while (!flag)
```

```
#pragma omp flush(a)
b=a;
```

```
a = foo();
#pragma omp flush(a)
flag = 1;
#pragma omp flush(flag)
```

```
#pragma omp flush(flag)
while (!flag){
#pragma omp flush(flag)
}
b=a;
```

```
a = foo();
flag = 1;
#pragma omp flush(a,flag)
```

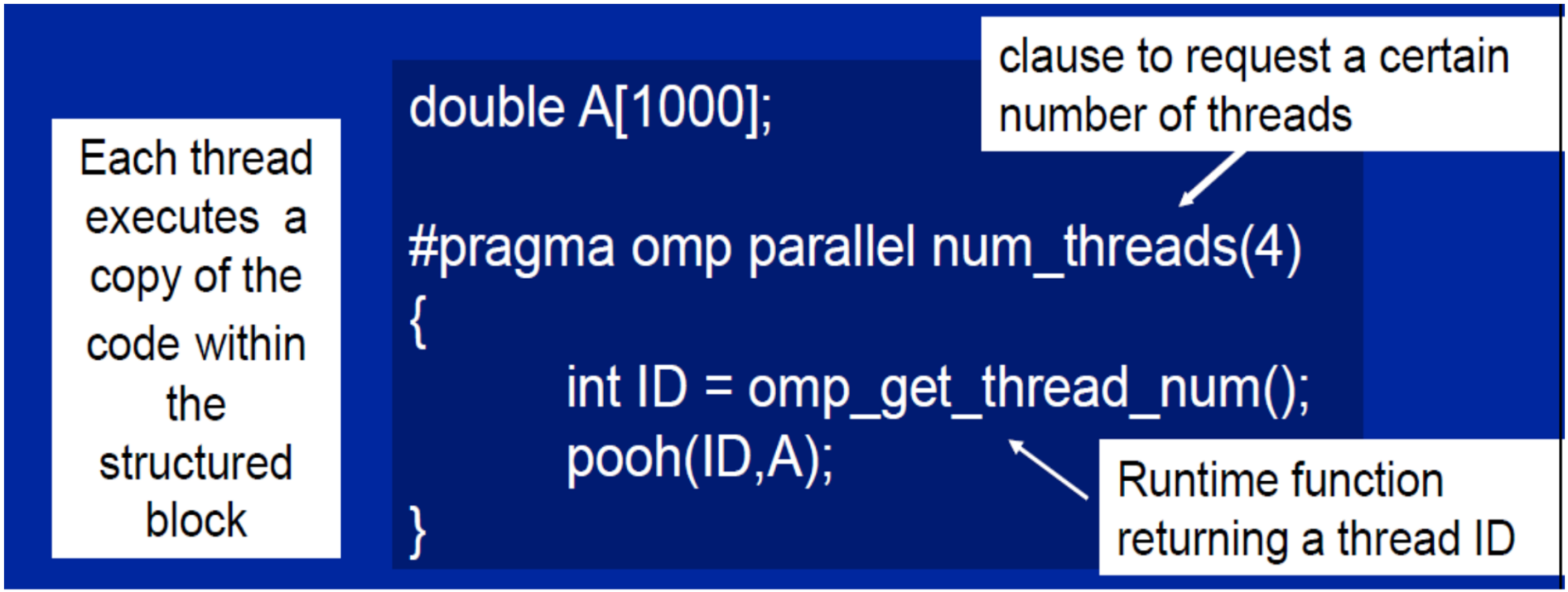
```
#pragma omp flush(flag)
while (!flag){
#pragma omp flush(flag)
}
#pragma omp flush(a)
b=a;
```

Flush

- A flush operation does not synchronize different threads. It only ensures that a thread‘s values are made consistent with main memory.
- Keeping track of consistency when flushes are used can be confusing in particular with the „flush(list)“.

Thread Creation: Parallel Regions

- Parallel threads are created in OpenMP with the parallel construct.
- For example to create a 4 thread parallel region:



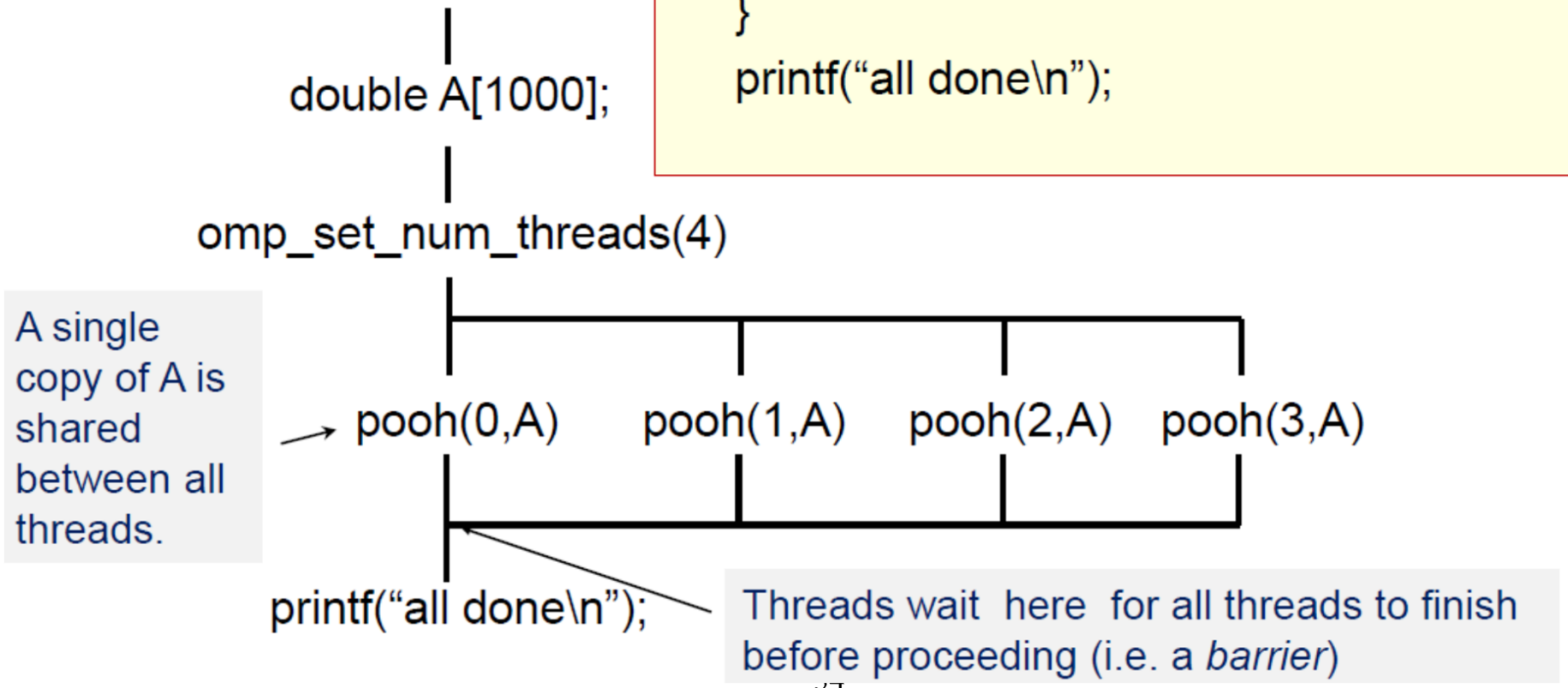
- Every thread calls pooh(ID,A) for ID = 0 to 3

Source T. Mattson, Intel

Thread Creation: Parallel Regions, cont.

- Each thread executes the same code

```
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



Source T. Mattson, Intel

Parallel Region

- The statements enclosed lexically within a parallel region define the lexical extent of the region.
- The dynamic extent further includes the routines called from within the construct.

```
#pragma omp parallel [clause list]
{
  parallel region
}
```

Parallel Region Example

```
main (){
int iam, nthreads;

#pragma omp parallel private(iam,nthreads)
 {
  iam = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  printf("ThreadID %d, out of %d threads\n", iam, nthreads);

  if (iam == 0) ! Different control flow
    printf("Here is the Master Thread.\n");
  else
    printf("Here is another thread.\n");
 }
}
```

Lexical and Dynamic Extend

```
main (){
int a[100];
#pragma omp parallel
 {
  Ex(a);
 }
}
```

```
sub Ex(int a[])
{
#pragma omp for
  for (int i= 1; i<n;i++)
     a[i] = i;
}
```

- Local (stack) variables of a function called in a parallel region are by default private.
- **"#pragma omp for"** must be in the dynamic extend of a parallel region.

Work-Sharing Constructs

- Work-sharing constructs distribute the specified work to threads within the current team.

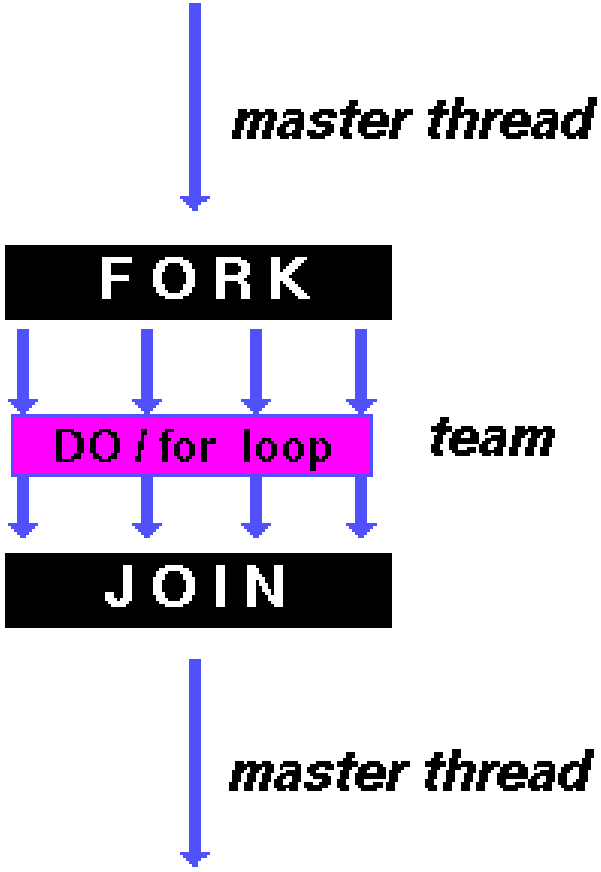
- Types
  - Parallel loop
  - Parallel section
  - Master region
  - Single region

Parallel Loop

```
#pragma omp for [clause list]
    for ...
```

- The iterations of the for-loop are distributed to the threads.
- The scheduling of loop iterations is determined by one of the scheduling strategies *static*, *dynamic*, *guided*, and *runtime*.
- There is no synchronization at the beginning.
- All threads of the team synchronize at an implicit barrier if the parameter *nowait* is not specified.
- The loop variable is by default private. It must not be modified in the loop body.
- The expressions in the for-statement are restricted.

#pragma omp for (Cont'd)



Scheduling Strategies

- Schedule clause: schedule (type [size])
- Scheduling types:
  - static: Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.
  - dynamic: The iterations are broken into chunks of the specified size. When a thread finishes the execution of a chunk, the next chunk is assigned to that thread.
  - guided: Similar to dynamic, but the size of the chunks is exponentially decreasing. The size parameter specifies the smallest chunk. The initial chunk is implementation dependent.
  - runtime: The scheduling type and the chunk size is taken from OMP_SCHEDULE environment variable.
  - auto: Schedule is left up to the runtime system to choose (does not have to be any of the above)

Which Schedule Clause to Use

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Source T. Mattson, Intel

Example 1: Dynamic Scheduling

```
main(){
int i, a[1000];

#pragma omp parallel
 {
  #pragma omp for schedule(dynamic, 4)
  for (i=0; i<1000;i++)
         a[i] = omp_get_thread_num();

  #pragma omp for schedule(guided)
  for (i=0; i<1000;i++)
         a[i] = omp_get_thread_num();

 }
}
```

Example 2: Dynamic Scheduling

```
main(){
int i, j, a[1000][1000];

#pragma omp parallel private(j)
 {
  #pragma omp for schedule(dynamic, 4)
  for (i=0; i<1000;i++)
     for (j=0; j<1000;j++)
           a[i][j] = …;

 }
```

- Note that only the loop variable of the parallel for loop is private. All other loop variables may be shared.
- Make sure to make all loop variables of parallel loops private - also the ones in loop nest.

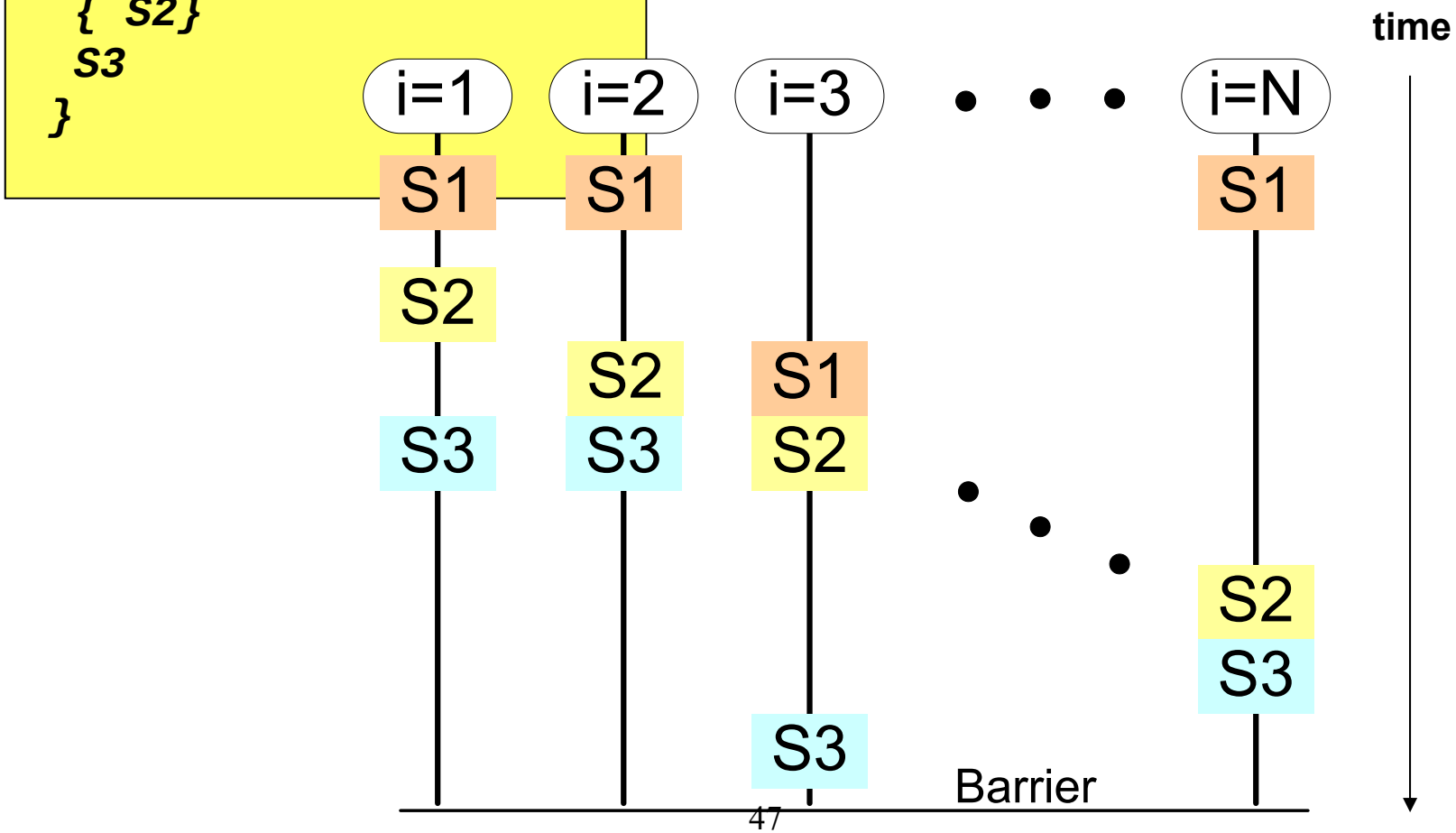
Ordered Construct

```
#pragma omp for ordered
    for (...)
     { ...
       #pragma omp ordered
       { ... }
       ...
     }
```

- Ordered construct must be within the dynamic extent of an *omp for* construct with an ordered clause.
- Ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.
- Code outside the ordered construct can run in parallel

Example with ordered clause

```
#pragma omp for ordered
for (...)
 { S1
  #pragma omp ordered
  { S2}
  S3
 }
```

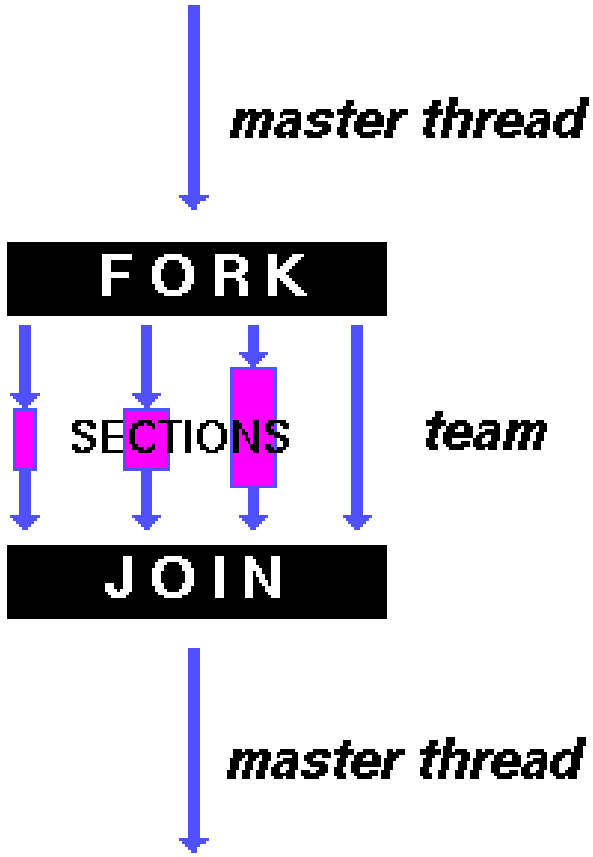


Parallel Section

```
#pragma omp sections [parameters]
{
 [#pragma omp section]{
      block }
 [#pragma omp section]{
      block }
}
```

- Each section of a parallel section is executed once by one thread of the team.
- Threads that finished their section wait at the implicit barrier at the end of the section construct.

Parallel Sections (Cont'd)



Example: Parallel Section

```
main(){
int i, a[1000], b[1000]

#pragma omp parallel private(i)
 {
 #pragma omp sections
  {
  #pragma omp section
  for (i=0; i<1000; i++)
    a[i] = 100;
  #pragma omp section
  for (i=0; i<1000; i++)
    b[i] = 200;
  }

 }
}
```

Overlap Computation with I/O Operations

```
#pragma omp parallel sections
 {
 #pragma omp section {                          // input thread
    for (int i=0; i<N; i++){
      (void) read_input(i);
      (void) signal_read(i);
    }
 }
  #pragma omp section {                         //processing thread
     for (int i=0; i<N; i++) {
        (void) wait_read(i);
         (void) process_data(i);
         (void) signal_processed(i);
     }
  }
  #pragma omp section{                           // output thread
      for (int i=0; i<N; i++) {
          (void) wait_processed(i);
           (void) write_output(i);
       }
  }
 }  // end of parallel sections
```


Source: C. Terboven, RWTH Aachen

Master / Single Region

```
#pragma omp master
  block

#pragma omp single [parameters]
  block
```

- A master or single region enforces that only a single thread executes the enclosed code within a parallel region.
- Common
  - No synchronization at the beginning of the region.
- Different
  - Master region is executed by master thread while the single region can be executed by any thread. No barrier at the end of master region.
  - Master region is skipped by other threads while all threads are synchronized at the end of a single region.

Combined Work-Sharing and Parallel Constructs

- #pragma omp parallel sections
- #pragma omp parallel for

separate parallel and work-sharing construct

```
#pragma omp parallel
 {
  #pragma omp for schedule(dynamic, 4)
  for (int i=0; i<1000;i++)
    a[i] = omp_get_thread_num();
 }
```

combined parallel and work-sharing construct

```
#pragma omp parallel for schedule(dynamic, 4)
  for (int i=0; i<1000;i++)
    a[i] = omp_get_thread_num();
```

Critical Section

```
#pragma omp critical [(Name)]
{ ... }
```

- Mutual exclusion
  - A critical section is a block of code that can be executed by only one thread at a time.
- Critical section name
  - A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name.
  - All unnamed critical directives map to the same name.
  - Critical section names are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Barrier

```
#pragma omp barrier
```

- The barrier synchronizes all the threads in a team.
- When encountered, each thread waits until all of the other threads in that team have reached this point.

Example: Critical Section

```
main(){
int ia = 0
int ib = 0
int itotal = 0

for (int i=0;i<N;i++)
 {
  a[i] = i;
  b[i] = N-i;
 }
```

```
#pragma omp parallel
{
#pragma omp sections
 {
  #pragma omp section
  {
  for (int i=0;i<N;i++)
    ia = ia + a[i];
  #pragma omp critical (c1)
   {
    itotal = itotal + ia;
   }}
  #pragma omp section
  {
  for (int i=0;i<N;i++)
    ib = ib + b[i]
  #pragma omp critical (c1)
   {
    itotal = itotal + ib;
   }}
}} }
```

Atomic Statements

```
#pragma ATOMIC
      expression-stmt
```

- The ATOMIC directive ensures that a specific memory location x is updated atomically
- Must have the following form:
  - *x binop= expr*
  - *x++ or ++x*
  - *x-- or --x*
  - where x is a scalar variable and expr does not reference the object designated by x.
- All parallel assignments to the location must be protected with the atomic directive.

Translation of Atomic

Only the load and store of x are protected, not the evaluation of expressions.

```
#pragma omp atomic
   x += expr
```

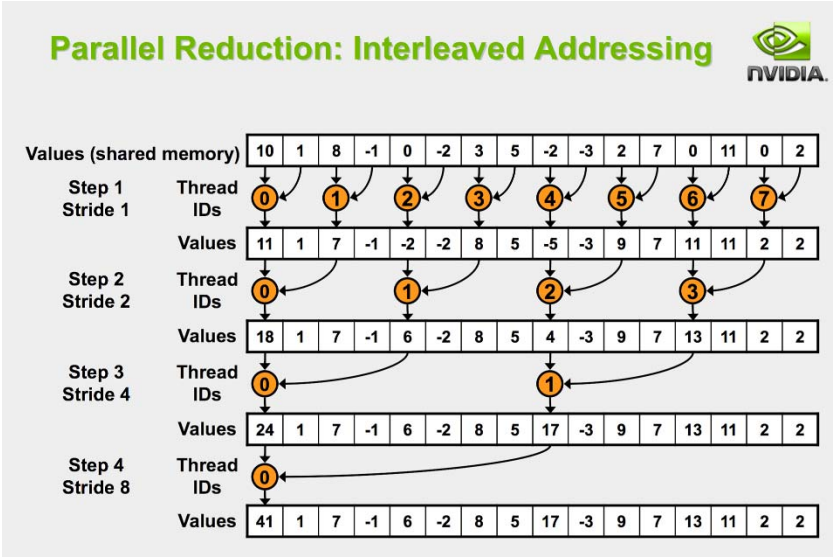
can be rewritten as

```
xtmp = expr
#pragma omp critical(name){
   x = x + xtmp
}
```

Reductions (1)

```
reduction(operator: list)
```

- This clause must be inside a parallel or a work-sharing construct and performs a reduction on the variables that appear in *list*, with the operator *operator*.
- Variables must be shared in enclosing parallel region and can be scalar or arrays.
- *operator* is one of the following:
  - ,+, *, -, &, ^, |, &&, ||
- A reduction is typically specified for statements of the form:
  - *x = x operator expr*
  - *x binop= expr*
  - *x++, ++x, x--, --x*



Reductions (2)

- Inside a parallel or a work-sharing construct:
  - a local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for a "+").
  - updates occur on the local copy.
  - local copies are reduced into a single value and combined with the original global value
- reduction operands intial values

operator	initial value
+	0
-	0
*	1
min	largest positive number
max	smallest negative number

source: The OpenMP Common Core Tutorial, SC19, Tim Mattson et al.

Example: Reduction

```
#pragma omp parallel for reduction(+: a,b)
for (i=0; i<n; i++) {
  a = a + c[i];
  b = b + d[i];
}
```

```
double sum = 0, m = -1;
#pragma omp parallel for reduction(+: sum) reduction (max: m)
for (i=0; i<n; i++) {
  sum += i;
  if (i > m) m = i;
}
```

```
for (i=0; i<n; i++) b[i] = 0;
#pragma omp parallel for reduction(+: b[0:n-1]) private (j)
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
        b[j] += c[i][j];
```


source: The OpenMP Common Core Tutorial, SC19, Tim Mattson et al.

Classification of Variables:
Data Scope Attribute Clauses

- private(var-list)
  - variables in var-list are private.
- shared(var-list)
  - variables in var-list are shared.
- default(private | shared | none)
  - sets the default for all variables in this region.
- firstprivate(var-list)
  - variables are private and are initialized with the value of the shared or private copy before the region.
  - applicable to: parallel, for, sections, single
- lastprivate(var-list)
  - variables are private and the value of the thread executing the last iteration of a parallel loop in sequential order is copied to the variable outside of the region.
  - applicable to: for, sections

Scoping Variables with Private Clause

```
int i, j;
i = 1;
j = 2;

#pragma omp parallel private(i) firstprivate(j)
 {
  i = 3;
  j = j + 2;
  printf("%d %d\n", i, j);
 }
```

- Note that private data is undefined on entry and exit of parallel regions.
  - Can use firstprivate and lastprivate to deal with this problem.
- The values of the shared copies of *i* and *j* are undefined on exit from the parallel region.
- The private copies of *j* are initialized in the parallel region to 2.

Lastprivate example

```
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
  for (i=0; i<n; i++)
     a[i] = b[i] + b[i+1];
}
// The value of i is n-1
a[i]=b[i];
```

- i after the parallel for loop has the value it held for the last sequential iteration (for i=n-1).

Copyprivate

```
#pragma omp parallel private(x)
{
  #pragma omp single copyprivate(x)
  {
    getValue(x);
  }
  useValue(x);
}
```

- Copyprivate
  - Clause only for single region.
- Value of x is copied to all private variables of all threads after single region.
- Do not use value of x after parallel region.

Runtime Routines for Threads (1)

Used to control and query the parallel execution environment.

- Sets number of threads in subsequent parallel regions unless overwritten by a num_threads clause.
  - omp_set_num_threads(count)
- Query the maximum number of threads available for team creation
  - numthreads = omp_get_max_threads()
- Query number of threads in the current team
  - numthreads = omp_get_num_threads()
- Query own thread number (0..n-1)
  - iam = omp_get_thread_num()
- Query number of processors available to program
  - numprocs = omp_get_num_procs()

Runtime Routines for Threads (2)

- Allow runtime system to adjust the number of threads for team creation

  omp_set_dynamic(logicalexpr)
- Query whether runtime system can adjust the number of threads

  logicalvar= omp_get_dynamic()
- Allow nesting of parallel regions

  omp_set_nested(logicalexpr)
- Query nesting of parallel regions. True if nested parallelism is activated.

  logicalvar= omp_get_nested()

Simple Locks

- Used to encode mutual exclusion with more flexibility than critical and atomic.
- Locks can be held by only one thread at a time.
- A lock is represented by a lock variable of type omp_lock_t.
- The thread that obtained a simple lock cannot set it again.

- Operations
  - omp_init_lock(&lockvar): initialize a lock
  - omp_destroy_lock(&lockvar): destroy a lock
  - omp_set_lock(&lockvar): set lock
  - omp_unset_lock(&lockvar): free lock
  - logicalvar = omp_test_lock(&lockvar): check lock and possibly set lock, returns true if lock was set by the executing thread.

Example: Simple Lock

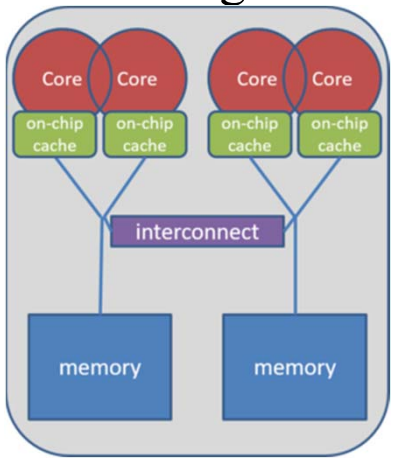
```
#include <omp.h>
int id;
omp_lock_t lock;

omp_init_lock(lock);
#pragma omp parallel shared(lock) private(id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lock); //Only a single thread writes
    printf("My Thread num is: %d", id);
  omp_unset_lock(&lock);

  WHILE (!omp_test_lock(&lock))
    other_work(id);    //Lock not obtained
  real_work(id);       //Lock obtained
  omp_unset_lock(&lock);//Lock freed
}
omp_destroy_lock(&lock);
```

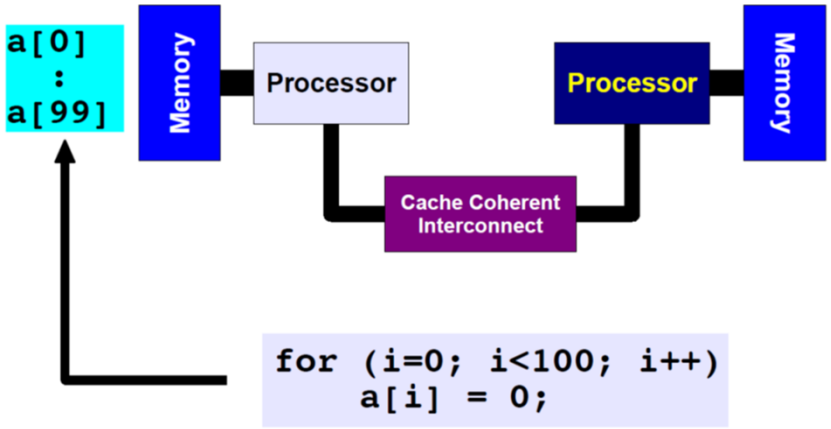
locked

locked

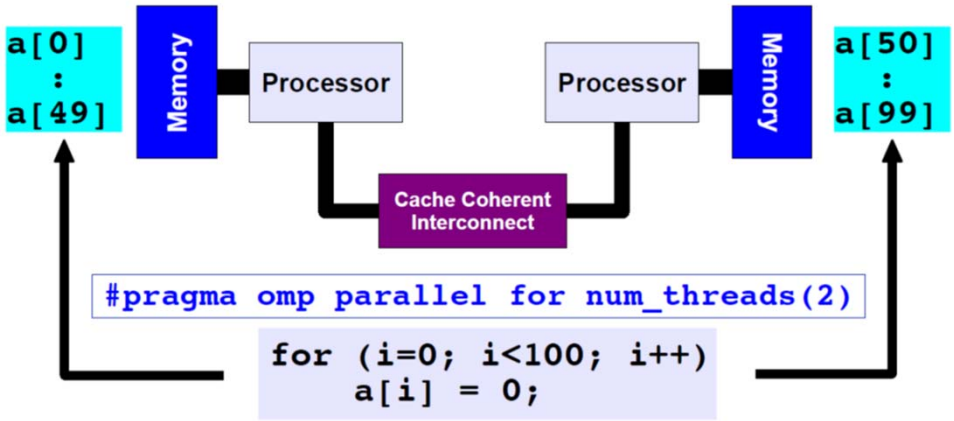
OpenMP Data Placement: First Touch

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer code. This is called first touch.
- Placement is OS-dependent.
- Windows and Linux use first touch placement policy by default.
- Placement is static.





All array elements are in the memory of the processor executing this thread.

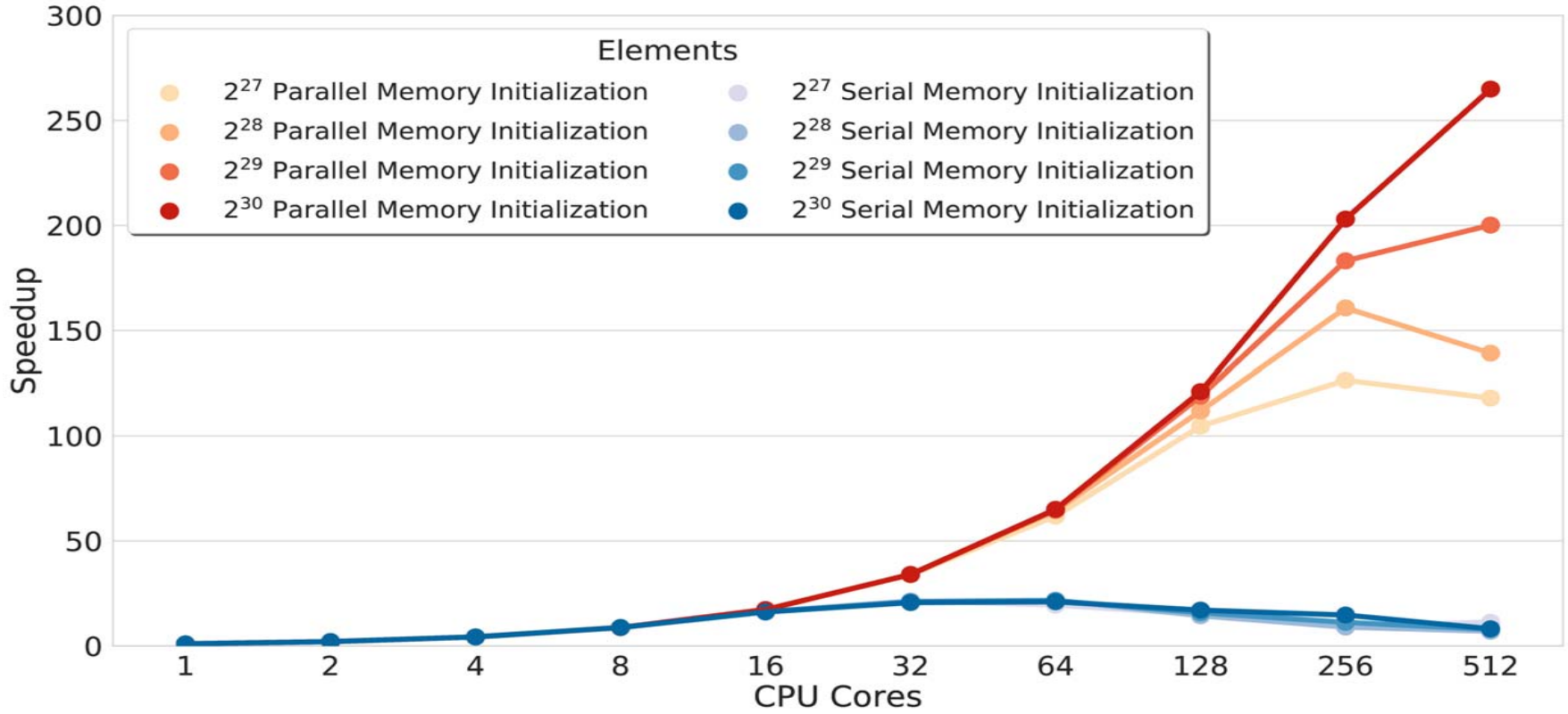


Both memories each have their half of the array.

source: Yonghon Yan, University of Houston

Performance Effect of First Touch

- Parallel Bitonic Sort with serial versus parallel memory initialization. (Part 8 of lecture)
- Mach-2



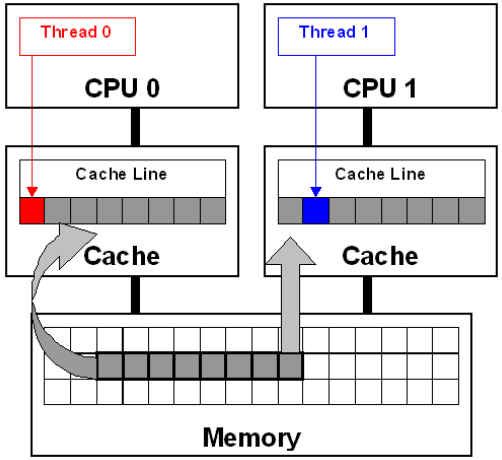
Advanced OpenMP Features

- Thread Affinity
- OpenMP tasks

Thread Affinity

Binding threads to HW topology can be crucial.

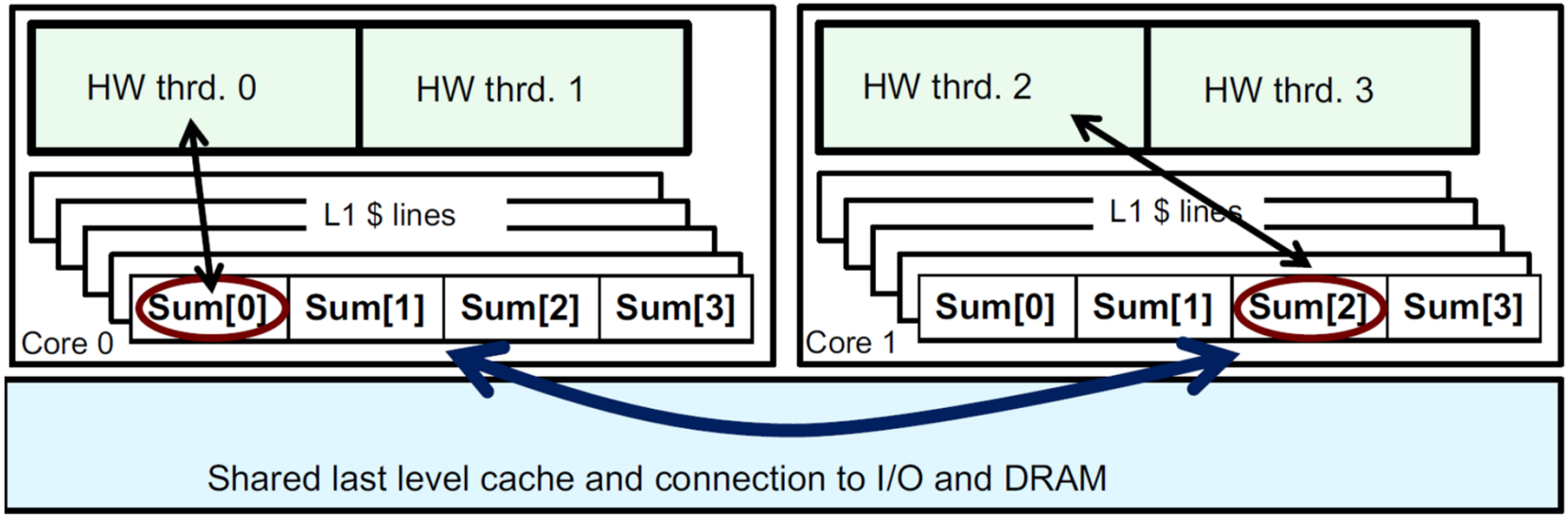
- Finer control for programm
  - improve locality between OpenMP threads
  - less false sharing
- Placing threads far apart, i.e. on different sockets may
  - improve aggregated memory bandwidth available
  - improve combined cache size available
  - increase synchronization
- Placing threads close together, ie. on adjacent cores with possible shared caches may
  - decrease synchronization
  - decrease available memory bandwidth and cache size
- Very HW and application specific
- Try a few variations to explore performance effects



source: Intel developer zone

False Sharing

- Independent data elements are placed in the same cache line.
- Each update will cause the cache lines to slosh back and forth between threads.



- Example for poor scalability:
  - Promote scalars to an array to support creation of an SPMD program.
  - Array elements are contiguous in memory and hence share cache lines.
- Solution:
  - Pad arrays thus elements are on distinct cache lines.

source: The OpenMP Common Core Tutorial, SC19, Tim Mattson et al.

Example
Eliminate False Sharing by Padding

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{             int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
              step = 1.0/(double) num_steps;
              omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {          int i, id,nthrds;
              double x;
              id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)   nthreads = nthrds;
               for (i=id, sum[id][0]=0.0;i< num_steps; i=i+nthrds) {
                         x = (i+0.5)*step;
                         sum[id][0] += 4.0/(1.0+x*x);
               }
    }
               for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

source: The OpenMP Common Core Tutorial, SC19, Tim Mattson et al.

Results
pi Program Padded Accumulator

- original serial pi program with 100.000.000 steps ran in 1.83 seconds.

Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define   PAD     8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS][PAD];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {         int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id][0]=0.0;i< num_steps; i=i+nthrds) {
                        x = (i+0.5)*step;
                        sum[id][0] += 4.0/(1.0+x*x);
            }
   }
            for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

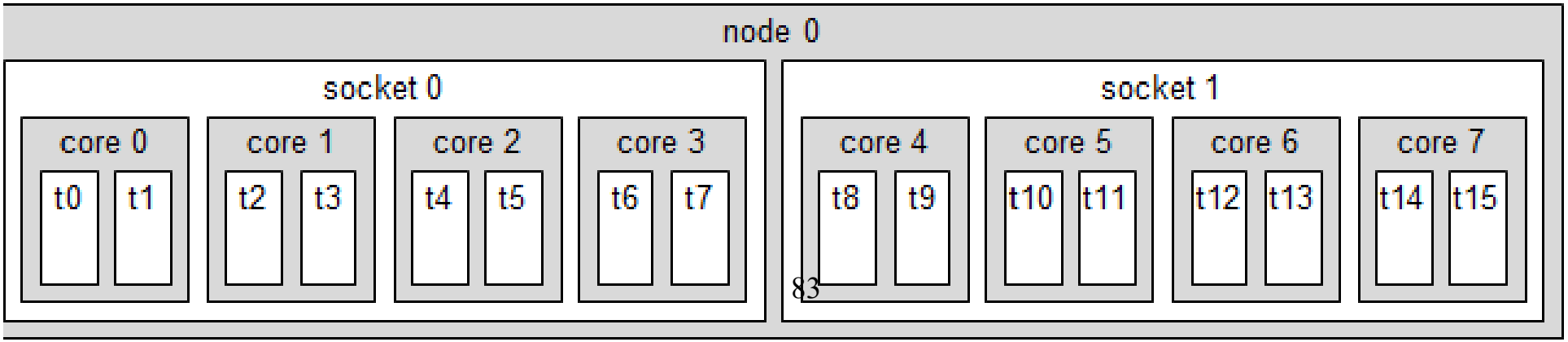
threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

- Intel compiler (icpc) with default optimization level O2 on Apple OS X 10.7.3 with dual core (4 HW threads) Intel Core i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

source: The OpenMP Common Core Tutorial, SC19, Tim Mattson et al.

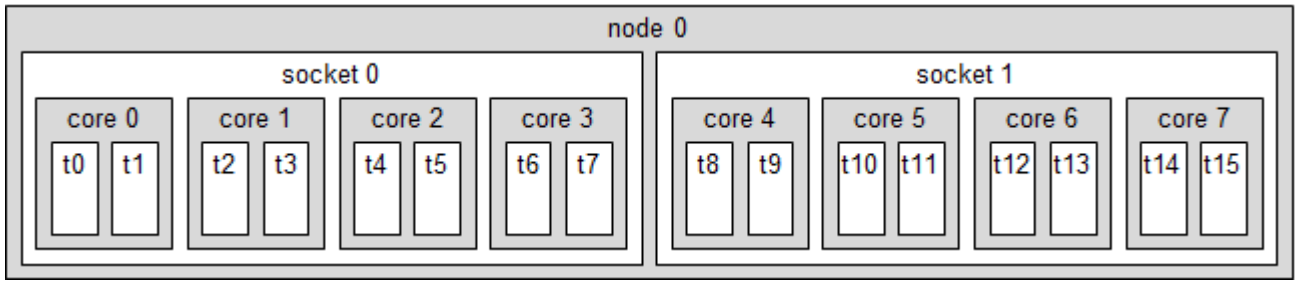
OpenMP 4.0 Places

- Place
  - HW resource that can execute an OpenMP thread.
  - Place-list is an ordered list of places.
  - Place-list is a static construct that should not change during program execution however its partition can change. It can be partitioned in different sets of places during runtime.
  - new environment variable **OMP_PLACES**
- Example: 8 places with 2 HW threads each
  - **OMP_PLACES="(0,1),(2,3),(4,5),(6,7),(8,9),(10,11),(12,13),(14,15)"**
  - **OMP_PLACES="(0:2):8:2"**



OpenMP 4.0 Places

- **OMP_PLACES=threads | cores | sockets**
  - threads: every HW thread defines a place
  - cores: every core defines a place
  - socket: every socket defines a place
- Examples
  - **OMP_PLACES=cores**
  - **OMP_PLACES=threads(4)**

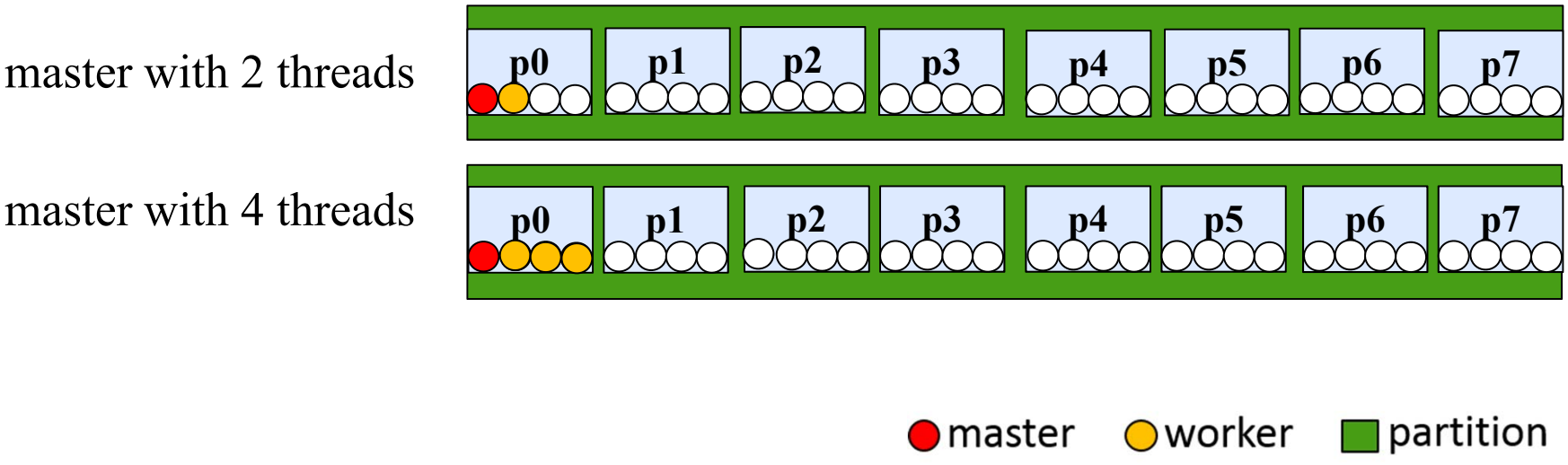


OpenMP Affinity Policies

- **proc_bind** clause
  - define affinity policy between OpenMP threads and places.
  - can dynamically change
- 3 policies
  - master: map OpenMP threads in the same place as the master thread
  - close: bind threads close to the master thread while still distributing threads for load balancing
  - spread: spread OpenMP threads evenly among places

Master Affinity

- assign OpenMP threads in the same place as the master
- for best data locality to the master thread
- **#pragma omp parallel proc_bind(master)**



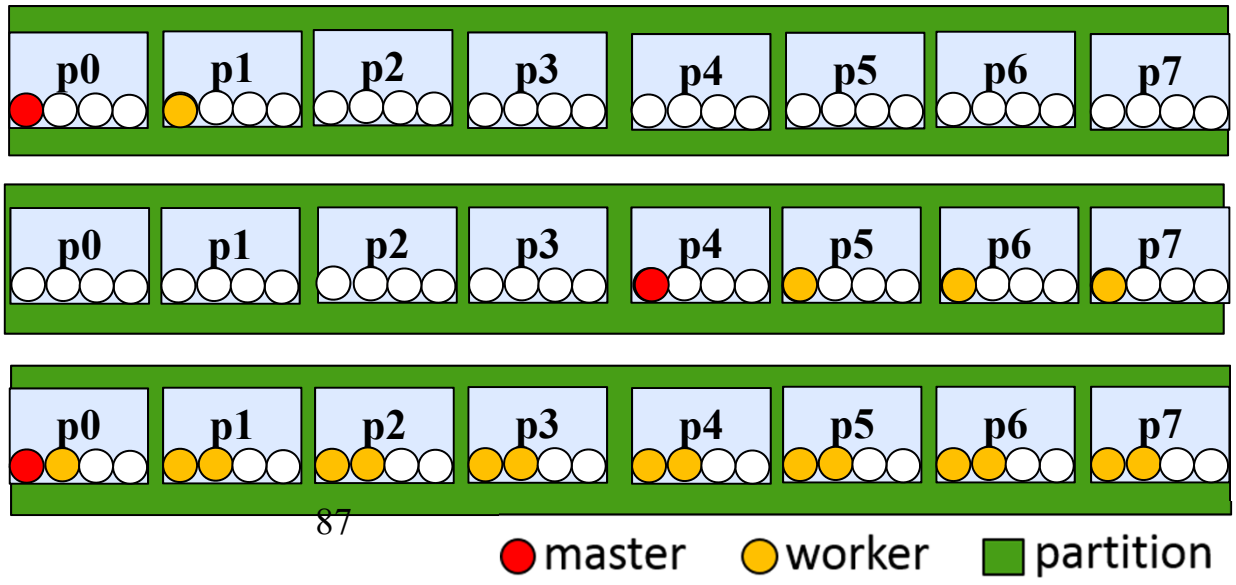
Close Affinity

- assign OpenMP threads near the place of the master
- place partitioning is not changed
- wrap around once each place gets one thread
- for data locality, load-balancing, and more dedicated resources
- **#pragma omp parallel proc_bind(close)**

close with 2 threads

close with 4 threads

close with 16 threads

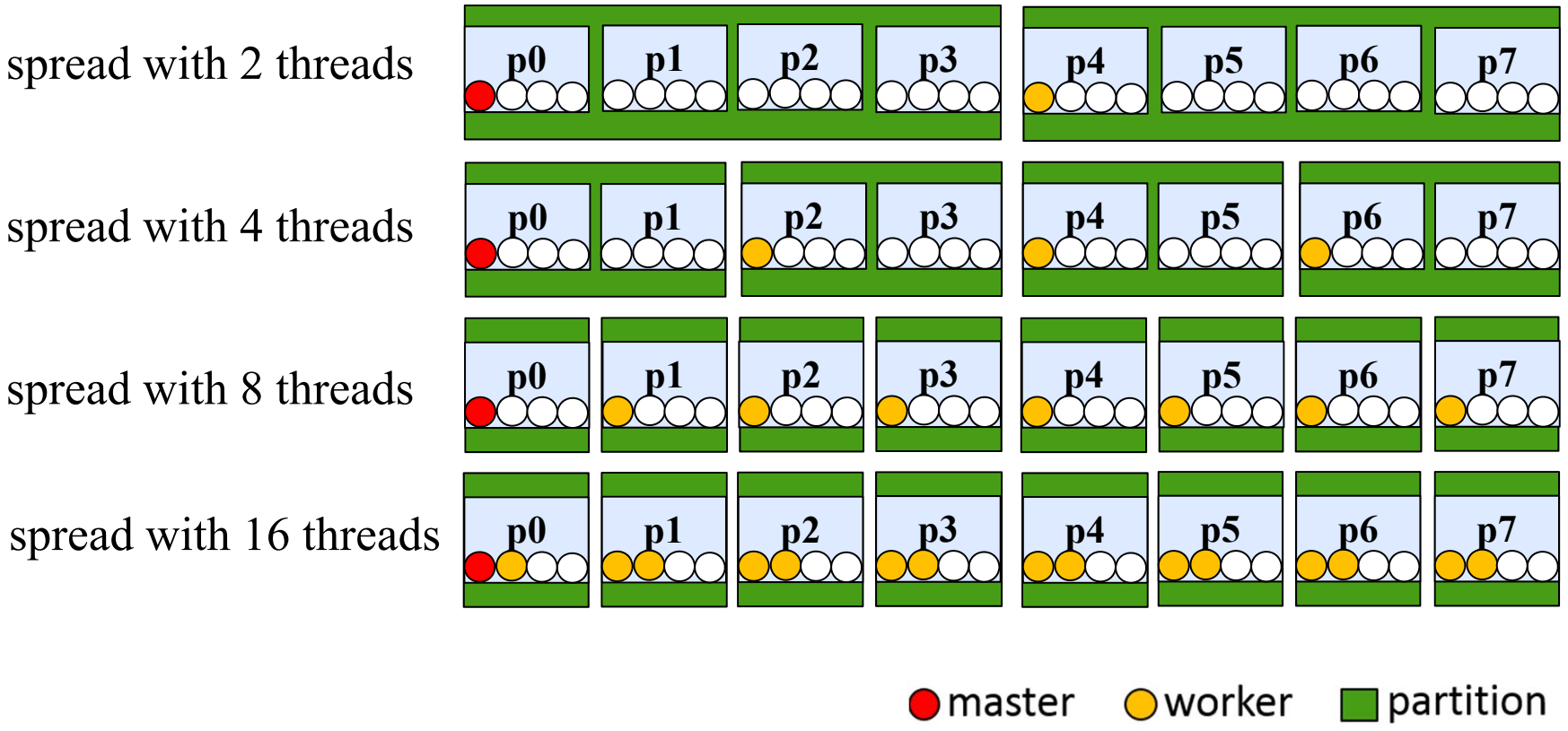


Spread Affinity

- for load balancing, most dedicated hardware resources
- spread OpenMP threads as evenly as possible among places
- partition the place list.
  - partition is a notation to express location of dynamically created threads - always in the same partition.
- create sub-partition of the place list
  - subsequent threads will only be allocated within sub-partition

Example: Spread Affinity

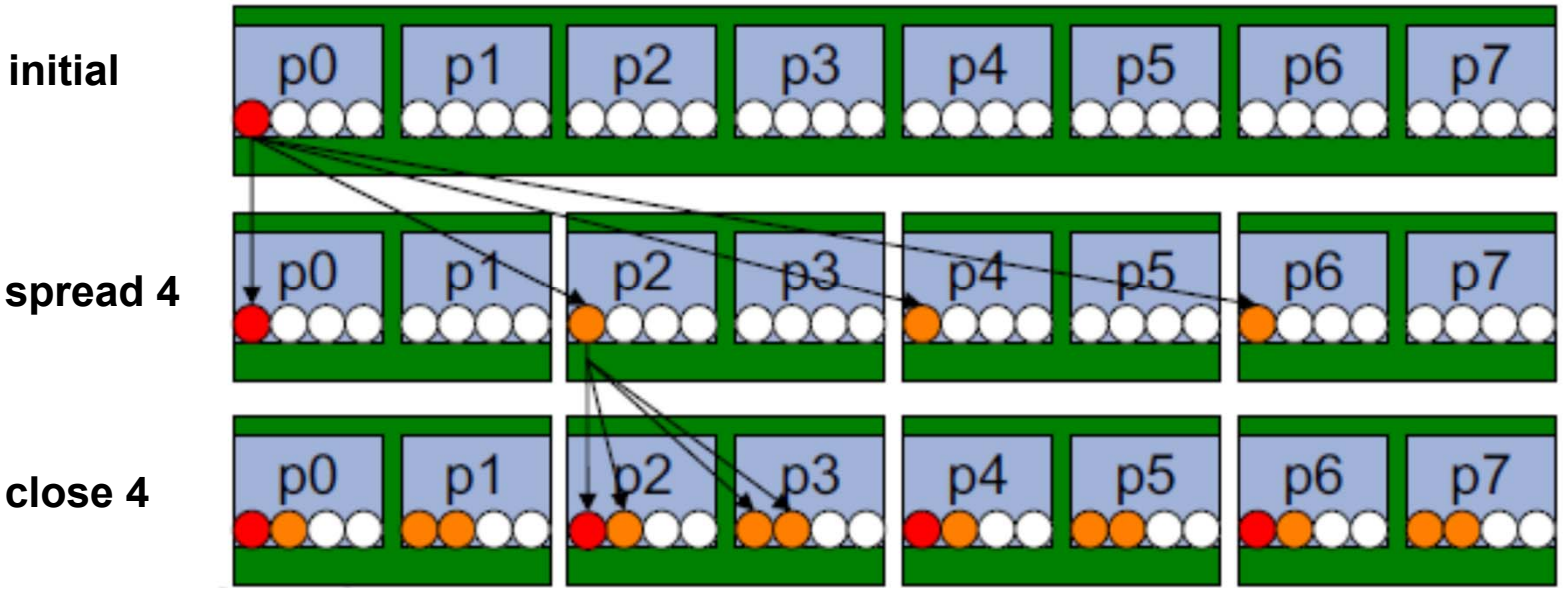
- **#pragma omp parallel proc_bind(spread)**
- nr of partitions = min(P,N) with P nr. of places and N nr. of threads



Example: Places and Binding

- separate cores for outer loop and near cores for innter loop
- outer parallel region: proc_bind(spread) num_threads(4)
  - spread creates partition,
- inner parallel region: proc_bind(close) num_threads(4)
  - binds threads within respective partition
- OMP_PLACES=(0,1,2,3), (4,5,6,7), … = (0-3):8:4 = cores

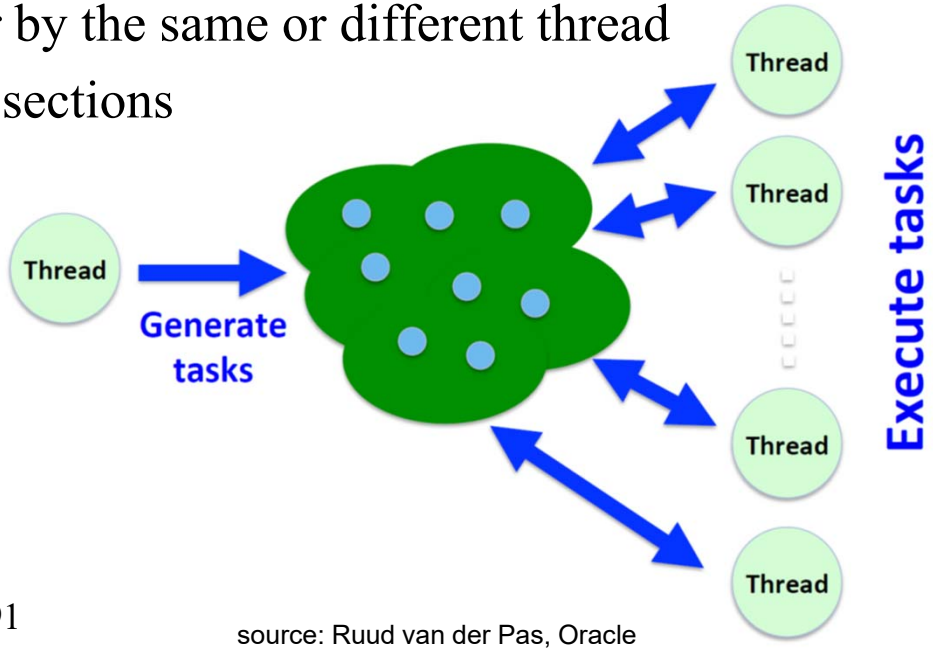
```
#pragma omp parallel proc_bind(spread) num_threads(4)
    #pragma omp parallel proc_bind(close) num_threads(4)
```



Source: C. Terboven et al. Advanced OpenMP Tutorial

Parallel Tasks

- OpenMP tasks
  - dynamically created independent pieces of work executed asynchronously (irregular, dynamic parallelism)
  - producer creates tasks
  - consumers execute tasks (possible later)
  - tasks can be executed by any thread in a team as soon as a thread becomes available
  - execution order of pool of task is unpredictable
  - execution of task immediately or later by the same or different thread
  - can be nested within parallel loops or sections
- allows to parallelize irregular problems
  - unbounded loops (e.g. while loops)
  - recursive algorithms



source: Ruud van der Pas, Oracle

OpenMP Tasks vs. OpenMP Sections

- OpenMP tasks break work into independent tasks that are executed asynchronously in the form of dynamically generated units of work (irregular parallelism).
  - more flexible scheduling opportunities
- OpenMP sections
  - Parallel sections are executed immediately by distinct threads.
  - Different threads execute different code. Units of work are statically defined.

OpenMP Tasks

- A task has
  - code to execute
  - a data environment (shared, private, …)
  - a thread that executes the code and uses the data
- Tasks can be nested
  - as part of another task
  - into a worksharing construct
- Tasks can be outside of OMP parallel region
  - Only a master thread will execute these tasks.

Tasking Constructs

```
#pragma omp task          // define a task

#pragma omp taskwait      // wait on completion of
                          // direct child tasks -
                          // not descendants.

#pragma omp taskgroup     // wait on completion,
                          // including descendant tasks

#pragma omp barrier       // barrier synchronization of all
                          // threads of a parallel region
                          // and complete execution of all
                          // tasks bound to this region
#pragma omp taskyield     // current task can be suspended
```

Taskgroup

- taskgroup is for deep synchronization against hierarchies of tasks
  - specifies a wait on completion of child tasks of the current task AND their descendant tasks
  - does not create a task region; only for group and synchronization purposes
- taskwait waits on the completion of the child tasks of the current tasks (siblings not their descendants)

source: Barbara Chapman, Univ. of Houston

Taskgroup vs. Taskwait

```
#pragma omp taskgroup
{
 #pragma omp task {              // Task T1
        #pragma omp task {       // Task T2
            …
        }
 }
 #pragma omp task {             // Task T3
        …
 }
}    // wait for T1, T2, T3
```

```
#pragma omp task {              // Task T1
       #pragma omp task {       // Task T2
           …
       }
}          // T1 may not wait for T2
#pragma omp task {              // Task T3
       …
}
#pragma omp taskwait     // wait for T1 and T3
                         // but not for T2
```

Taskgroup Example

```
int main() {
   int i;
   tree_t tree;
   init_tree(tree);

  #pragma omp parallel
  {
     #pragma omp task
         start_background_work();
     #pragma omp master
        for (i=0; i < max_steps; i++) {
           #pragma omp taskgroup {
                #pragma omp task
                   compute_tree(tree);
           } // wait on ALL tasks to complete
        }
 } // background work completed
  print_results();
}
```

```
void compute_tree(tree_t tree)
{
    if (tree->left) {
       #pragma omp task
           compute_tree(tree->left);
    }
    if (tree->right) {
       #pragma omp task
           compute_tree(tree->right);
    }
    #pragma omp task
           compute_something(tree);
}
```

Task Data Scoping

- static and global variables are shared
- If *shared* scoping is not inherited
  - variables are *firstprivate* unless *shared* in the enclosing context.
- variables declared inside task without explicit data scoping are private

Data Scoping for Tasks Example

```
int a = 1;
void foo()
{
   int b = 2, c = 3;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d = 4;
        #pragma omp task
        {
               int e = 5;

               // Scope of a: shared,          value of a: 1
               // Scope of b: firstprivate,    value of b: 0 / undefined
               // Scope of c: shared,          value of c: 3
               // Scope of d: firstprivate,    value of d: 4
               // Scope of e: private,         value of e: 5
} } }
```

Source: C. Terboven, RWTH Aachen

Recursive Task Parallelism
Example: Fibonacci

- skip OpenMP overhead once a certain n is reached
- stop creating new tasks for small n to avoid recursion overhead.

```
int main(int argc, char* argv[])
{
      …
   #pragma omp parallel {
      #pragma omp single {
         fib(input);
      }
   }
      …
 }
```

```
int fib(int n) {
    if (n < 2) return n;
    if (n <= 30) return serial_fib(n);
 int x, y;         // private variables

 #pragma omp task shared(x) {
     x = fib(n-1);
 }
 #pragma omp task shared(y) {
     y = fib(n-2);
 }

 #pragma omp taskwait
     return x+y;
 }
```

Source: C. Terboven, RWTH Aachen

Recursive Task Parallelism
Example: Fibonacci

- Active thread should do more then just creating two tasks and then wait for them to finish

```
int main(int argc, char* argv[])
{
      …
   #pragma omp parallel {
      #pragma omp single {
         fib(input);
      }
   }
      …
 }
```

```
int fib(int n) {
    if (n < 2) return n;
    if (n <= 30) return serial_fib(n);
 int x, y;

 #pragma omp task shared(x) {
     x = fib(n-1);
 }

     y = fib(n-2);  // to be executed
                    // by this thread

 #pragma omp taskwait
     return x+y;
 }
```

Source: C. Terboven, RWTH Aachen

Task Dependence

- #pragma omp task depend(dependence-type: list)
- enforce execution dependences between sibling tasks
- dependence-type:
  - in: generated task depends on all previously *generated* siblings that reference at least one of the list items in an out or inout clause.
  - out and inout: generated task depends on all previously *generated* siblings that reference at least one of the list items in an in, out or inout clause.
- dependence types are solely for expressing task dependences.
  - do not indicate any memory access patterns inside task regions

Task Dependence
Example 1

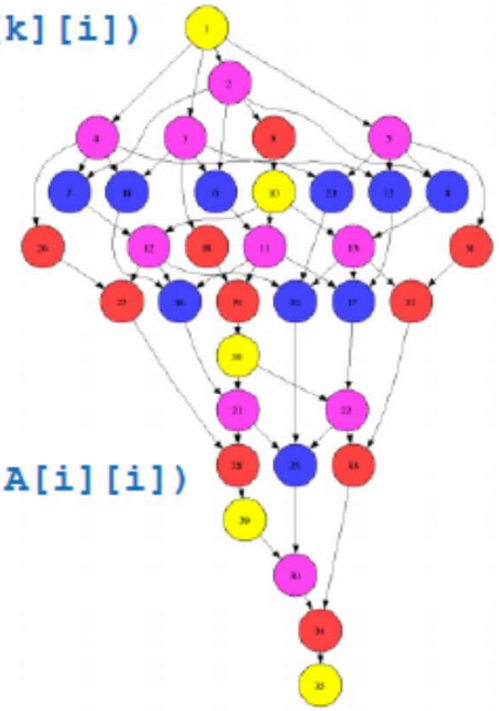
```
int x = 1;
#pragma omp parallel
#pragma omp single
   {
      #pragma omp task shared(x) depend(out: x)  // Task 1
         x = 2;
      #pragma omp task shared(x) depend(in: x)   // Task 2
         printf("x + 1 = %d. ", x+1);
      #pragma omp task shared(x) depend(in: x)   // Task 3
         printf("x + 2 = %d\n", x+2);
   }
```

Task Dependence
Example 2

```
int a,b,c;
#pragma omp parallel  {
   #pragma omp master {
       #pragma omp task depend(out:a) {
            #pragma omp critical
                printf ("Task 1\n");
       }
       #pragma omp task depend(out:b) {
           #pragma omp critical
                printf ("Task 2\n");
           #pragma omp task depend(out:a,b,c) {
                sleep(1);
                #pragma omp critical
                printf ("Task 5\n");
           }
       }
       #pragma omp task depend(in:a,b) depend(out:c) {
                printf ("Task 3\n");
       }
       #pragma omp task depend(in:c) { printf ("Task 4\n"); }
   }    // master
}
```

Task Dependence
Example 3

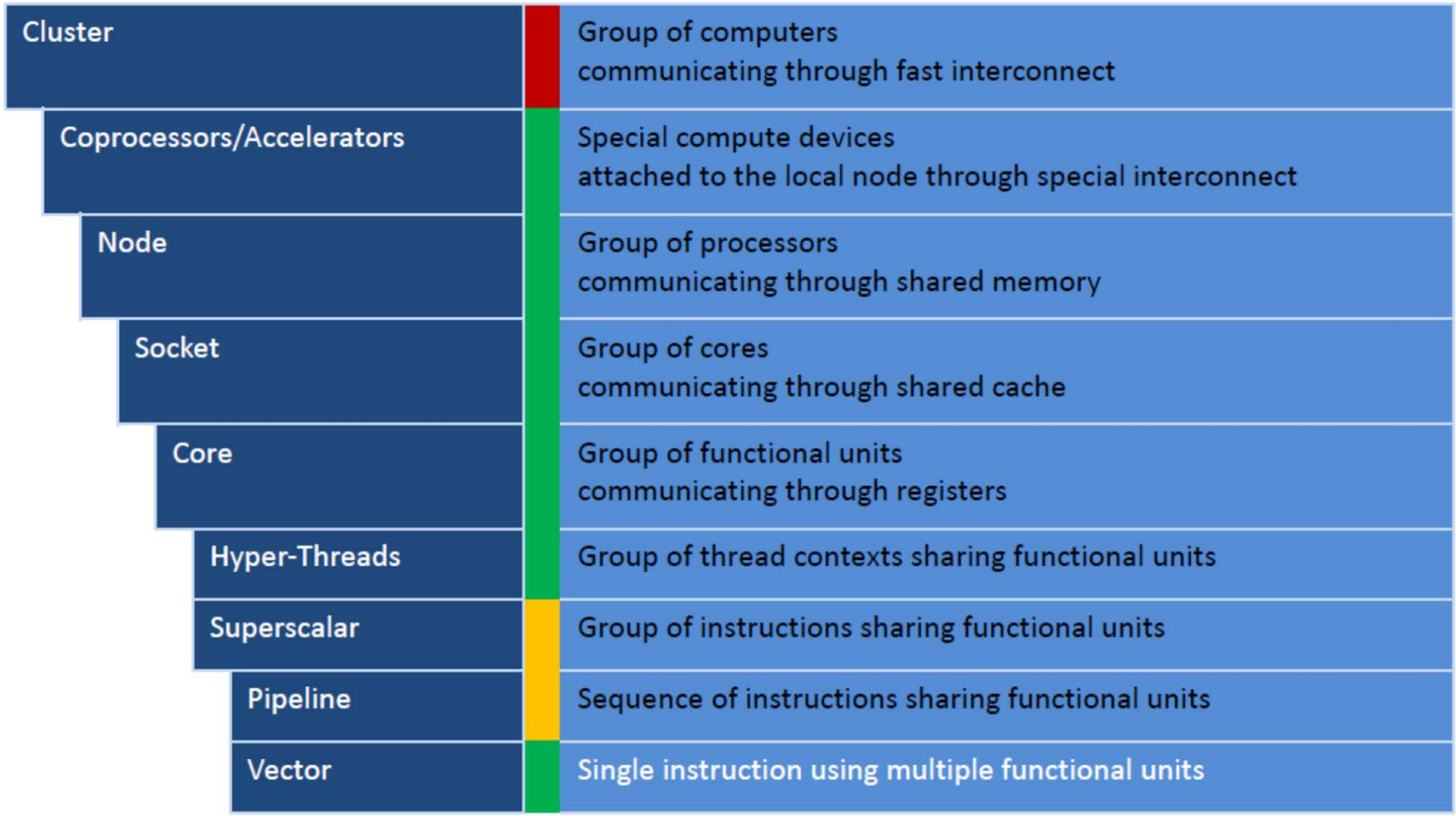
```
void blocked_cholesky( int NB, float A[NB][NB] ) {
   int i, j, k;
   for (k=0; k<NB; k++) {
     #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
     for (i=k+1; i<NT; i++)
       #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
          strsm (A[k][k], A[k][i]);
       // update trailing submatrix
       for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
           #pragma omp task depend(in:A[k][i],A[k][j])
                            depend(inout:A[j][i])
             sgemm( A[k][i], A[k][j], A[j][i]);
         #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
            ssyrk (A[k][i], A[i][i]);
       }
   }
}
```



* image from BSC

Source: C. Terboven, M. Klemm

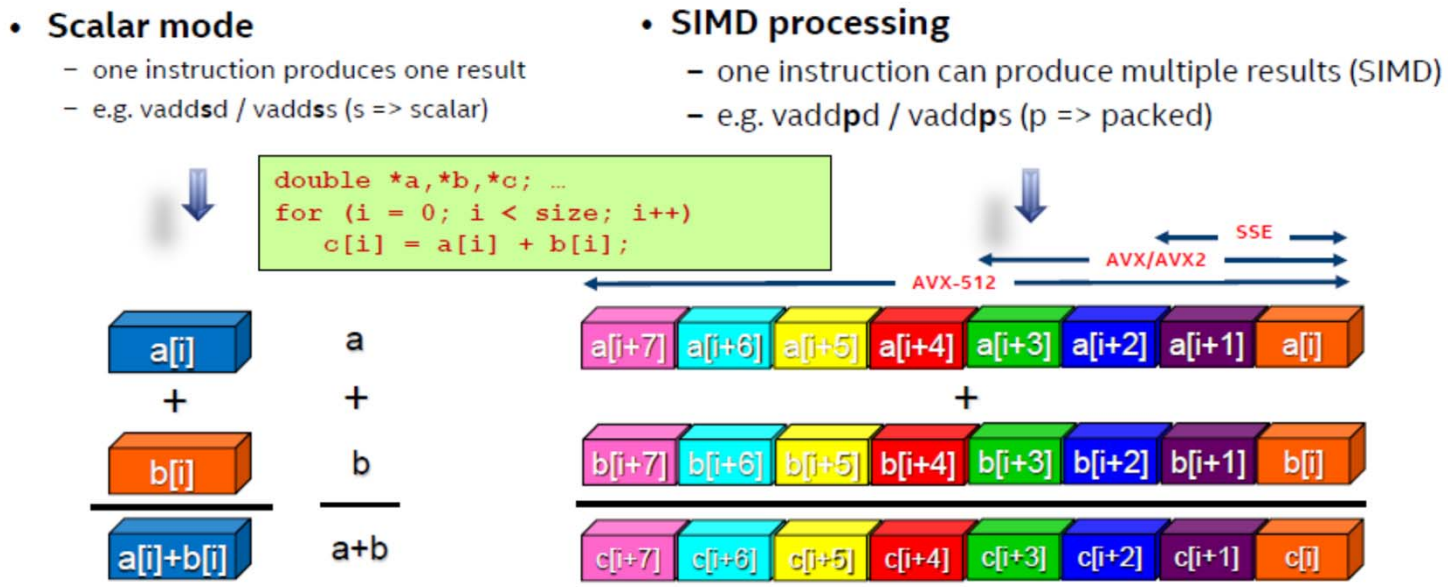
Vectorization with OpenMP (1)



Source: Michael Voss, Intel

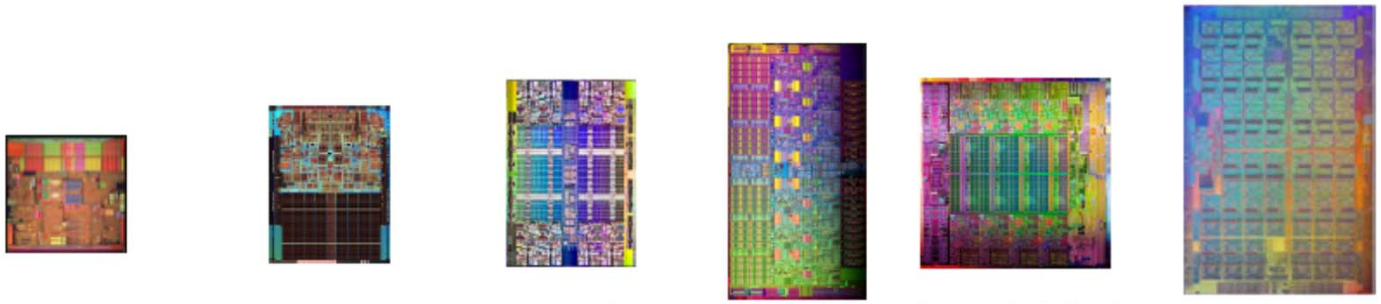
Vectorization with OpenMP (2)

- SIMD Single Instruction Multiple Data
- Vectorization: transform a scalar operation acting on a single data at a time (SISD) to an operation acting on multiple data elements at once (SIMD).



Source: Michael Voss, Intel

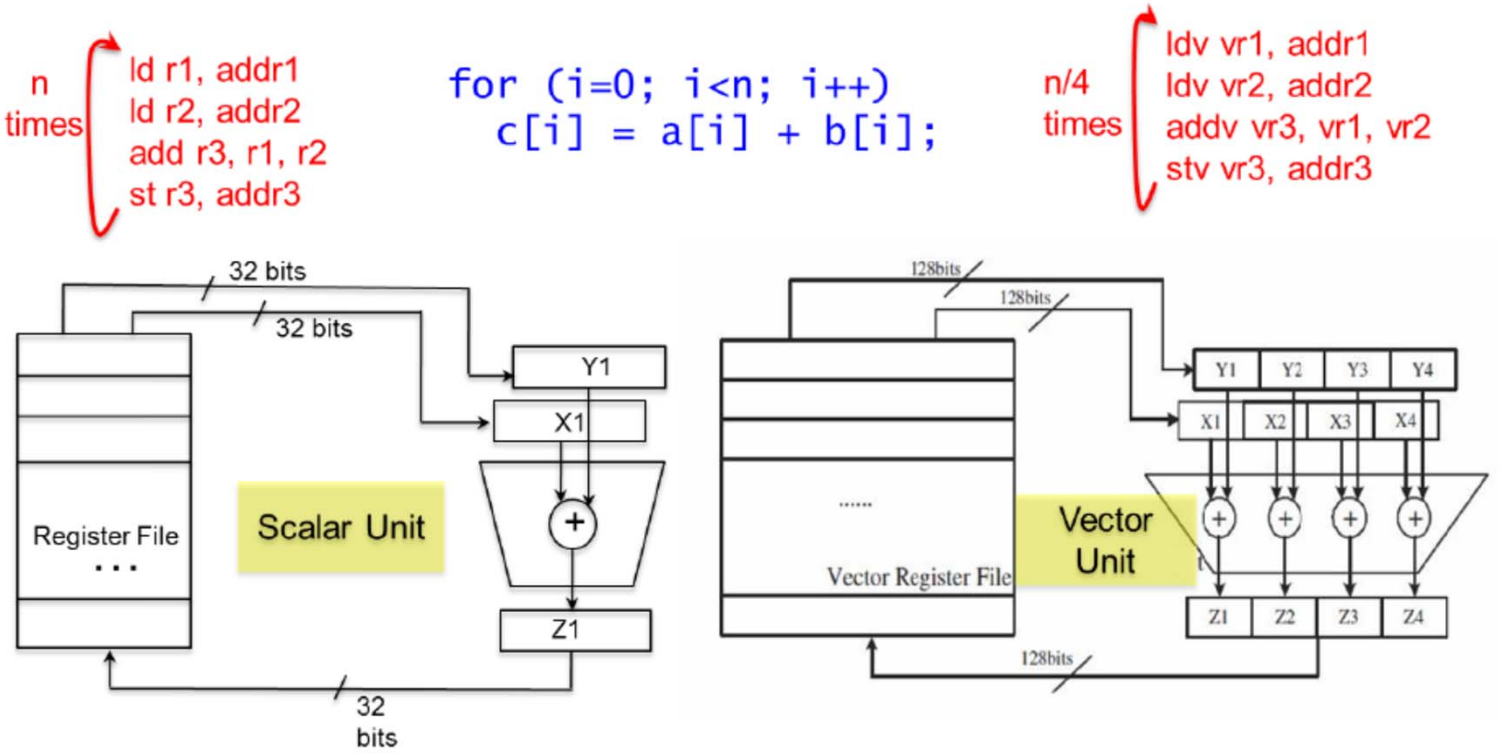
Evolution of Intel Hardware



Images not intended to reflect actual die sizes

	64-bit Intel® Xeon® processor	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor E5-2600v3 series	Intel® Xeon® Scalable Processor
Frequency	3.6 GHz	3.0 GHz	3.2 GHz	3.3 GHz	2.3 GHz	2.5 GHz
Core(s)	1	2	4	6	18	28
Thread(s)	2	2	8	12	36	56
SIMD width	128 (2 clock)	128 (1 clock)	128 (1 clock)	128 (1 clock)	256 (1 clock)	512 (1 clock)

Source: Michael Klemm, Intel



source: Maria Garzaran, Saeed Maleki, William Gropp, David Padua

- Legal to vectorize
  - A statement inside a loop which is not involved in a cycle of the dependence graph can be vectorized.

```
   for (i=0; i<n; i++){
S1   a[i] = b[i] + 1;
   }
```

```
a[0:n-1] = b[0:n-1] + 1;
```

```
   for (i=1; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i-1] + 2;
   }
```

```
a[1:n-1] = b[1:n-1] +1;
c[1:n-1] = a[0:n-2] +2;
```

- Benefit of vectorization
  - increase available memory bandwidth to cache
  - increase throughput of compute operations
  - more energy efficient

Source: Michael Voss, Intel

Vectorize Innermost Loop

```
for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
    a[i][j]=a[i][j]+a[i-1][j];
  }
}
```

Ignoring (freezing) the outer loop:

```
for (j=1; j<n; j++) {
    a[i][j]=a[i][j]+a[i-1][j];
}
```

S1

```
for (i=1; i<n; i++) {
    a[i][1:n-1]=a[i][1:n-1]+a[i-1][1:n-1];
}
```


Source: Michael Voss, Intel

How Do Use Vectorization?

- Libraries
  - MKL, OpenBLAS, BLIS, fftw, numpy, OpenCV
- Autovectorization of compilers
- Directive-based languages: OpenMP SIMD pragma
- Writing intrinsics/assembly code

Auto-vectorization

- Compilers offer auto-vectorization as an optimization pass
  - Code analysis detects code properties that inhibit SIMD vectorization
  - Heuristics determine if SIMD execution might be beneficial.
  - Compiler generates SIMD instrucations
- Example: Intel Composer XE
  - vec (automatically enabled with –O2)
  - qopt-report

Source: Michael Klemm, Intel

Prevent Auto-Vectorization

- data dependences
  - loop-carried dependencies
- pointer aliasing
- function call inside of loops
  - functions might not be vectorized because of side-effects
- complex control flow
  - different vector lanes executing different code
- loops not countable (loop bounds can be variables but must be invariant within the loop; exit from loop cannot be data-dependent)
- mixed data types
- non-unit stride between elements
- loop body too complex (register pressure)
- cost model by compiler: vectorization seems inefficient
- etc.

```
void process (int *a, int *b) {
    for (int i=1; i<N; i++){
        a[i] = b[i-1] + …;
}
```

```
void mix (float  *restrict a, double *restrict b, float *c) {
    for (int i=1; i<N; i++){
        b[i] = b[i] + c[i];
        a[i] = a[i] - b[i];
    }
}
```


Source: Michael Klemm, Intel

Slow-down Vectorization

- loop iteration count not a multiple of vector length
- serial function calls inside of loops
- indirect memory accesses
- loop stride different from 1
- data structures not aligned at a multiple of the vector length

```
for (i=0; i<N; i++)
    A[i] = B[C[i]]*d
```

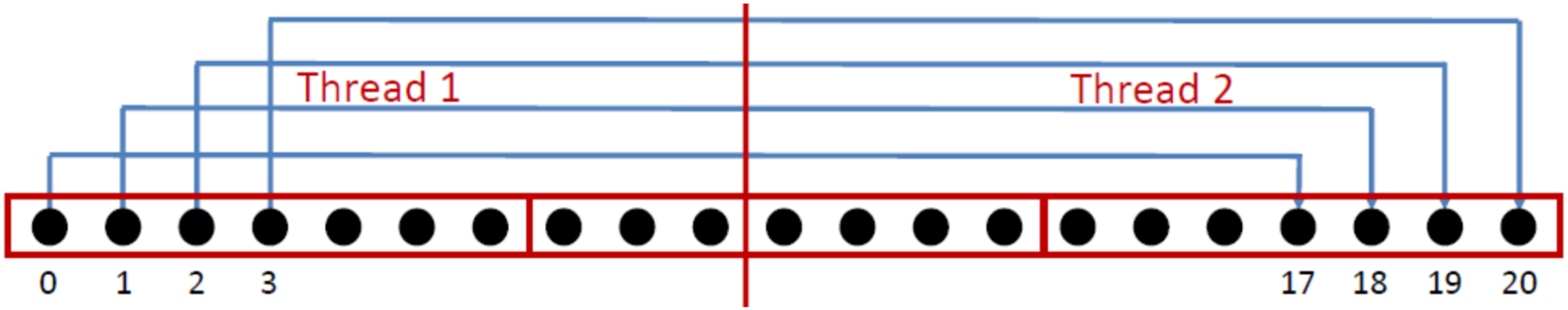
```
for (i=0; i<N; i+=2)
    A[i] = B[i]*d
```


Source: Zakhar Matveev, Intel

Loop-carried dependencies

- Can we parallelize or vectorize the loop?

```
void lcd_ex(float* a, float* b, size_t n, float c1, float c2) {
    for (int i = 0; i < n; i++) {
        a[i] = c1 * a[i + 17] + c2 * b[i];
}   }
```



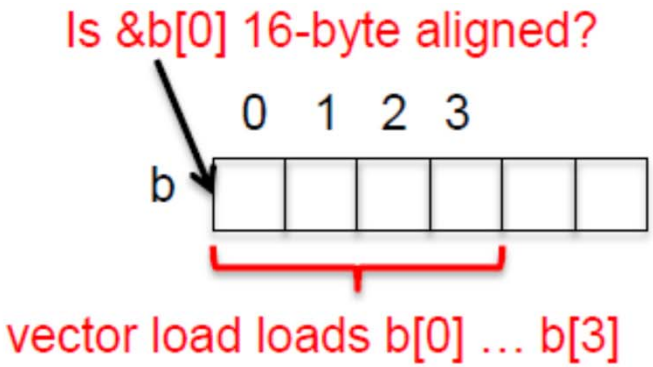
- Parallelization
  - Not as is.
  - Yes, if we apply variable renaming to eliminate anti-dependence.
- Vectorization
  - Yes, if vector lenght (e.g. 16) is shorter than any distance of any dependency.

Source: Michael Klemm, Intel

Alignment

- SSE Vector loads/stores 128 consecutive bits to/from a vector register.
- Data addresses need to be 16-byte (128 bit) aligned to be loaded/stored for SSE, 32-byte aligend for AVX/AVX2 and 64-byte aligned for AVX512
- Intel platforms support aligned and unaligned load/stores, but unaligned is slower.

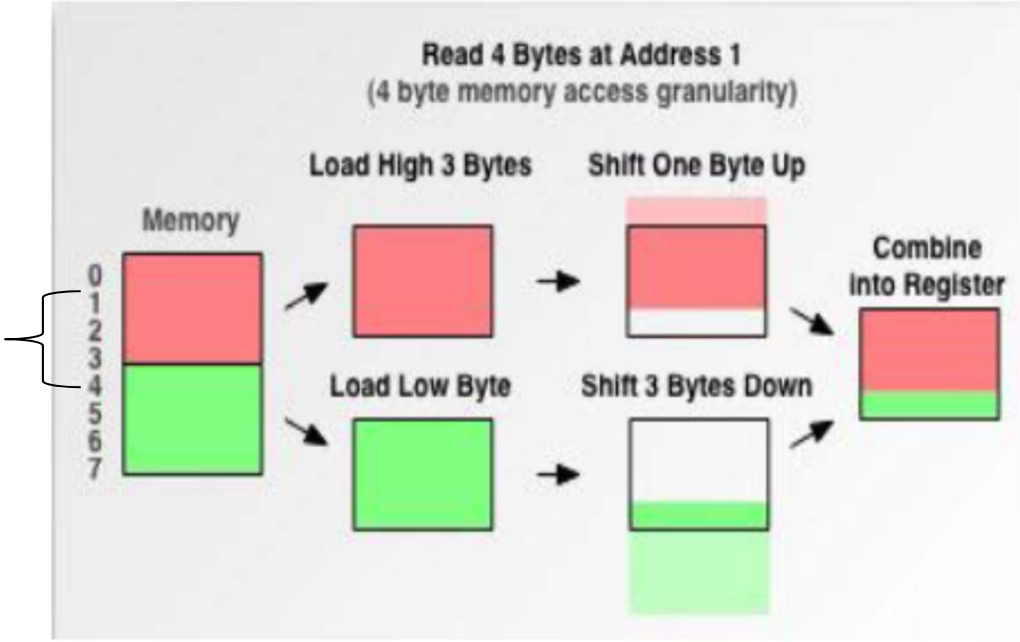
```
void test1(float *a,float *b,float *c)
{
      for (int i=0;i<LEN;i++){
        a[i] = b[i] + c[i];
}
```



Source: Michael Voss, Intel

Why Data Alignment May Improve Efficiency

- Vector load/store from aligned data requires one more memory access.
- Vector load/store from unaligned data requires multiple memory accesses and some shift operations.



Reading 4 bytes from address 1 requires two loads

Source: Michael Voss, Intel

Alignment

- Use posix_memalign to align data at 16/32/64/.. byte addresses.
- Make the compiler aware of x-byte alignment.
- Code may fail if data are not aligned.

```
float * a;
posix_memalign(&a, ...);
float * b;
posix_memalign(&b, ...);
float s;
...
#pragma omp simd aligned(a, b: 32)
for (int i = 0; i < N; i++) {
  a[i] += s * b[i];
}
```

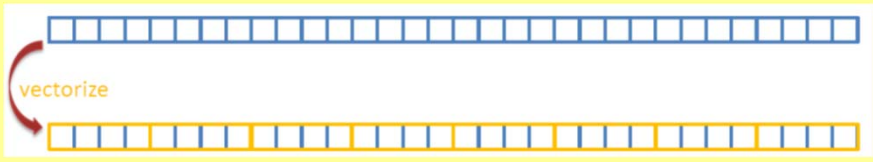
Source: Markus Höhnerbach, RWTH Aachen

OpenMP SIMD Loop Construct

- Vectorize a loop nest
  - cut loop into chunks that fit a SIMD vector register
  - no parallelization of the loop body
- SIMD chunk: set of iterations that are executed concurrently by a SIMD instruction in a single thread. Number of iterations in a SIMD chunk is the vector length.
- Each iteration in a chunk is executed by a SIMD lane.
- Compiler free to select proper vector length suitable to the architecture.

```
#pragma omp simd [clause[, clause]…]
for-loops
```

```
void simd_loop(double *a, double *b, double *c, int n){
  int i;
#pragma omp simd
  for (int i=0; i<n; i++)
      a[i] = b[i]+ c[i];
}
```

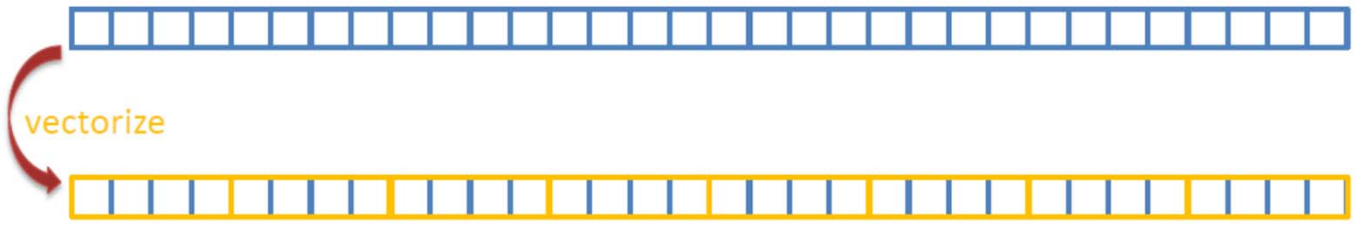


Source: Using OpenMP - the next steps, MIT Press

OpenMP SIMD Loop Construct

```
#pragma omp simd [clause [, clause]…]
        for-loops
```

```
float sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction (+:sum)
  for (int k=0; k<n; k++)
      sum += a[k] * b[k];
  return sum;
}
```



Programmer asserts:

- n is loop invariant
- sum not aliased with a[] and b[]
- + operator is associative (compiler can reorder for better vectorization)

Vectorization is done even if no performance gain achieved.

Source: Michael Klemm, Intel

Why use OpenMP SIMD instead of intrinsics?

- OpenMP is portable vs intrinsics are compiler/architecture specific
- With OpenMP, you do not select an ISA (i.e. SSE, AVX, etc.)
- OpenMP enables us to describe properties of loops and instruct compiler to vectorize it – in a portable form.
- It is less likely that with OpenMP we have to modify the code when moving to a different architecture/compiler.

Source: Michael Voss, Intel

OpenMP SIMD Clauses

- private (variables that can be privatized)
- lastprivate (private but last value is needed)
- reduction (guarantees associativity of operations)
- collapse (combined nested loops)
- linear (describe induction variables)
- simdlen (preferred number of iterations to execute concurrently)
- safelen (max. iterations that can be executed concurrently)
- aligned (data alignment information)

Private Variables in SIMD constructs

- SIMD-enabled functions with declare
- Privatize return values of function calls to avoid race conditions.
- One instance for t1 and t2 is created per SIMD lane.

```
#pragma omp declare simd
      double func1(double a, double b) { return a*b;}
#pragma omp declare simd
      double func2(double a, double b) { return a/b;}

e = simd_func(a,b,c,128);    // call the function

void simd_func(double *a, double *b, double *c, int n){
   int i; double t1,t2;
  #pragma omp simd private(t1,t2)
     for (int i=0; i<n; i++) {
       t1 = func1(b[i],c[i]);
       t2 = func2(b[i],c[i]);
       a[i] = t1 * t2;
  }
//  a[:] = func1(b[:],c[:])*func2(b[:]c[:]);
}
```

Source: Using OpenMP - the next steps, MIT Press

Safelen Clause

- Set the limit on the vector length.
  - safelen(m) clause specifies that a maximum of m iterations of the loop can be combined to a vector.
- Impact vector length used by the generated SIMD instruction.
- This is not a command but a proposal to the compiler.
- Example with vector length of up to 16.

Source: Using OpenMP - the next steps, MIT Press

```
void simd_safelen(double *a, double *b, double *c, int
  n, int offset) {
  int i;

  #pragma omp simd safelen(16)
  for (i=offset; i<n; i++) {
      a[i] = b[i-offset] + c[i];
  }
}
```

Aligned Clause

- Important for good vector performance.
- Align clause with variables that must have an array or pointer type for C.
- Data elements (x) should be aligned in memory that is a multiple of the size of the alignment value (16) in bytes.
- All variables in the clause are guaranteed to point to an object that is aligned in memory at an address that is a multiple of the number of bytes specified by the algorithm.
- Assumption: float data types with 4 bytes, *x* aligned to 16 byte boundary
  - Compiler can generate 128-bit vector load and store instructions.

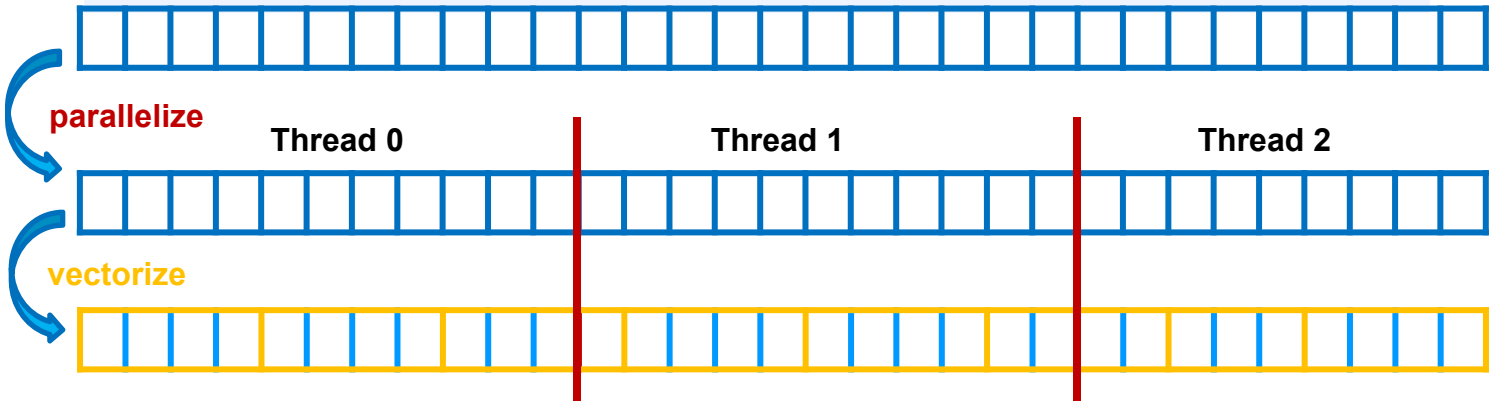
```
void simd_align(float *x, float scale, int n) {
  int i;

  #pragma omp simd aligned(x:16)
  for (i=0; i<n; i++) {
      x[i] = x[i]*scale;
  }
}
```


Source: Using OpenMP - the next steps, MIT Press

Composite FOR SIMD Construct

- Combine SIMD with parallel loop worksharing construct.
  - First, chunks of loop iterations are distributed across the threads in a team.
  - Second, chunks of loop iterations may be converted to fit SIMD vector.

```
float sprod(float *a, float *b, int n)      {
    float sum = 0.0f;
#pragma omp parfor simd reduction (+:sum)
    for (int k = 0; k < n; k++)
        sum += a[k] * b[k];
    return sum;
}
```



- Solution: chose chunk size as multiple of SIMD length

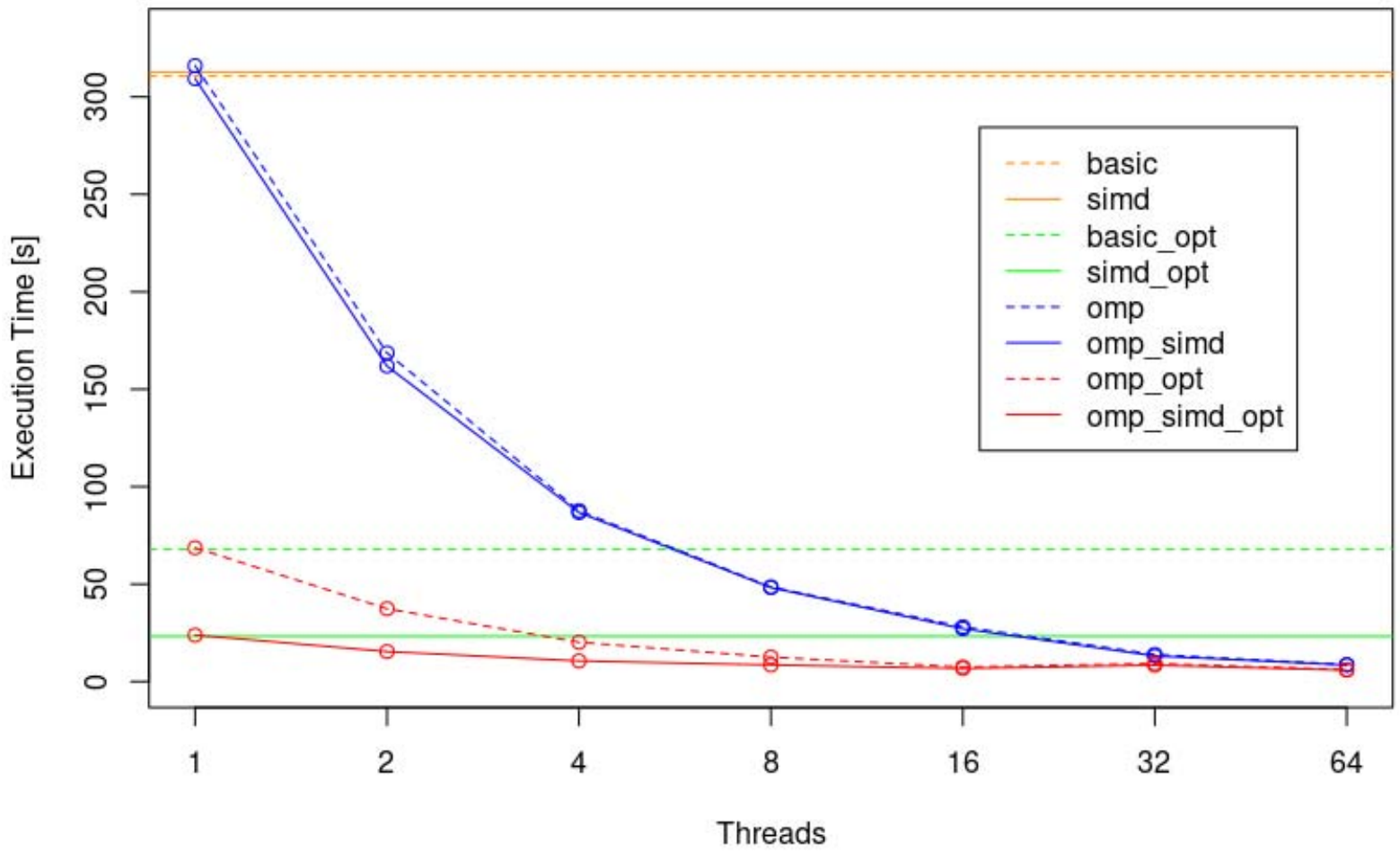
```
#pragma omp par for simd reduction (+:sum) schedule (simd:static)
```

SIMD Vectorization for a Cellular Automata Algorithm (1)

- see part 8 for details of the algorithm

```
void CellularAutomataNonConditional(uint32_t *array, size_t size,
                                    size_t iterations) {
    uint32_t* aux = (uint32_t*) AlignedAllocation((size + 2) * sizeof(uint32_t),
                          MEMORY_PAGE_SIZE);

    for(size_t i = 0; i < iterations; ++i) {
        size_t cell_index = 1;

        #pragma omp simd linear(cell_index : 1) aligned(array, aux : 32)
        for (cell_index = 1; cell_index < size + 1; ++cell_index) {
            if(i % 2 == 0) {
                UpdateCellNonConditional(array, aux, cell_index);
            }
            else {
                UpdateCellNonConditional(aux, array, cell_index);
            }
        }
        …
```

SIMD Vectorization for a Cellular Automata Algorithm (2)



Environment Variables

- OMP_NUM_THREADS=4
  - Number of threads used by default in a team of a parallel region
- OMP_SCHEDULE=”dynamic”, OMP_SCHEDULE=”GUIDED,4“
  - Selects scheduling strategy to be applied at runtime
- OMP_DYNAMIC=TRUE
  - Allow runtime system to adjust the number of threads.
- OMP_NESTED=TRUE
  - Allow nesting of parallel regions.

Example Codes
Sort in Ascending Order

Parallelize the following code fragment (sort in ascending order) using OpenMP:

```
void selection_sort(int numbers[], int array_size) {
    int i, j;
    int min, temp;
    for (i = 0; i < array_size-1; i++) {
        min = i;
        for (j = i+1; j < array_size; j++)
              if (numbers[j] < numbers[min])
                    min = j;
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

Solution Idea

```
void selection_sort(int numbers[], int array_size) {
    int i, j;
    int min, temp;
    for (i = 0; i < array_size-1; i++) {
        min = i;
        #pragma omp parallel for
        for (j = i+1; j < array_size; j++)
                if (numbers[j] < numbers[min])
                        min = j;
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

Why is this not correct?

Solution

```
void selection_sort(int numbers[], int array_size) {
    int i, j, k;
    int gmin, temp;
    int min[omp_get_max_threads()];
    for (i = 0; i < array_size-1; i++) {
        #pragma omp parallel private(t){
                used_thread = omp_get_thread_num();
                nr_threads = omp_get_num_threads();
                int t = used_thread;
                min[t] = i;
                #pragma omp for
                for (j = i+1; j < array_size; j++)
                        if (numbers[j] < numbers[min[t]])
                                min[t] = j;
        }
        gmin = i;
        for (k = 0; k<nr_threads; ++k)
            if (numbers[min[k]] < numbers[gmin])
                gmin = min[k];
        temp = numbers[i];
        numbers[i] = numbers[gmin];
        numbers[gmin] = temp;
    }
```

MERGESORT

- Mergesort is an example of a recursive sorting algorithm.
- It is based on the divide-and-conquer paradigm
- It uses the merge operation as its fundamental component (which takes in two sorted sequences and produces a single sorted sequence)
- Drawback of mergesort: Not in-place (uses an extra temporary array)

source: Aydin Buluc, LBNL

Algorithm Complexity Analysis

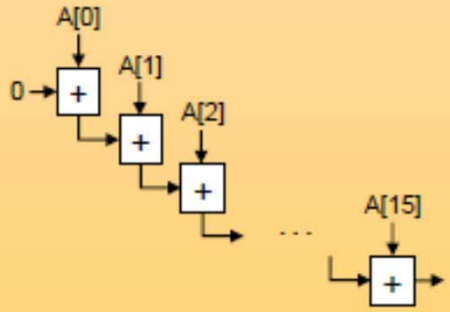
- *Work*:
  - T_1 = total number of operations in the algorithm for sequential execution. Equal to the time used for serial execution of algorithm.
  - T_p = sum over all operations of all processors for parallel algorithm with p processors.
- *Depth* or *span*: shortest possible execution time
  - T_∞ = longest series of operations to be performed sequentially due to data dependencies (critical path). Time it takes to execute the critical path with infinite number of processors. This is the shortest possible execution time.
- *Average parallelism*: measure of parallelism inherent in the algorithm $\frac{T_1}{T_\infty}$

source: Aydin Buluc, LBNL

Example: Algorithm Complexity Analysis

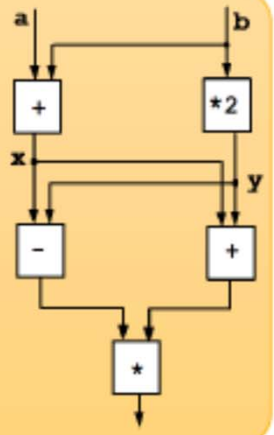
- Work = 16
- Depth = 16
- Average Par = 1

```
c = 0;
For (i=0; i<16; i++)
    c = c + A[i]
```



- Work = 5
- Depth = 3
- Average Par = 5/3

```
x = a + b;
y = b * 2
z =(x-y) * (x+y)
```

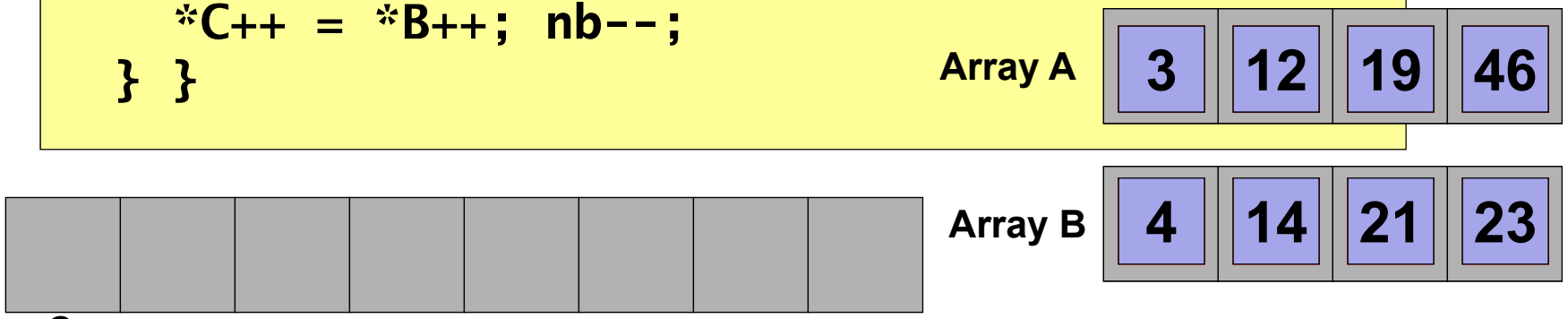


source: Nima Honarmand, Stony Brook University

Merging Two Sorted Arrays

```
template <typename T>
void Merge(T *C, T *A, T *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    } }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  } }
```

Time to merge n elements
= Θ(n).

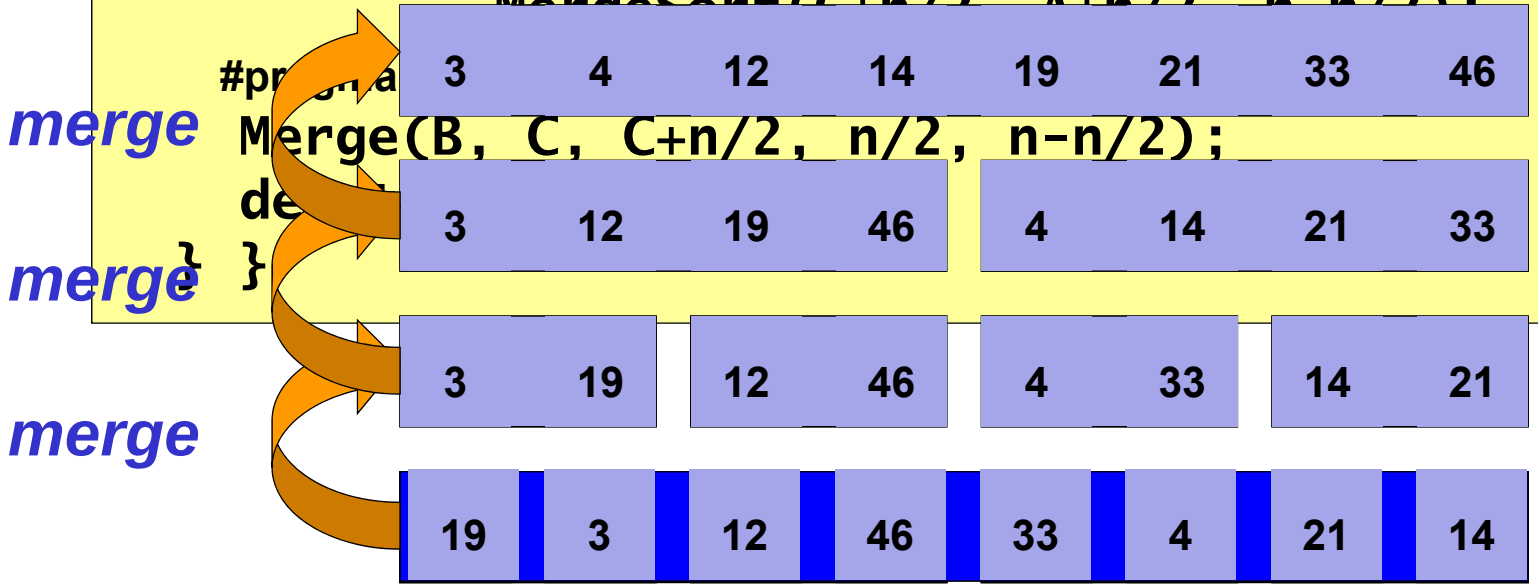


source: Aydin Buluc, LBNL

Parallel Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, int
  if (n==1) {
    B[0] = A[0];
  } else {
    T *C = new T[n];
      #pragma omp task
            MergeSort(C, A, n/2);
      #pragma omp task
            MergeSort(C+n/2, A+n/2, n-n/2);
      #pragma
    Merge(B, C, C+n/2, n/2, n-n/2);
    de
  } }
```

A: input (unsorted)
B: output (sorted)
C: temporary



source: Aydin Buluc, LBNL

Work of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C = new T[n];
      #pragma omp task
            MergeSort(C, A, n/2);
      #pragma omp task
            MergeSort(C+n/2, A+n/2, n-n/2);
    #pragma omp taskwait;
    Merge(B, C, C+n/2, n/2, n-n/2);
    delete[] C;
  } }
```

Work: $T_1(n)$ = $2T_1(n/2) + \Theta(n)$

= $\Theta(n \lg n)$

source: Aydin Buluc, LBNL

Span of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C = new T[n];
      #pragma omp task
            MergeSort(C, A, n/2);
      #pragma omp task
            MergeSort(C+n/2, A+n/2, n-n/2);
    #pragma omp taskwait;
    Merge(B, C, C+n/2, n/2, n-n/2);
    delete[] C;
  } }
```

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$

source: Aydin Buluc, LBNL

Parallelism of Merge Sort

Work: $T_1(n) = \Theta(n \lg n)$

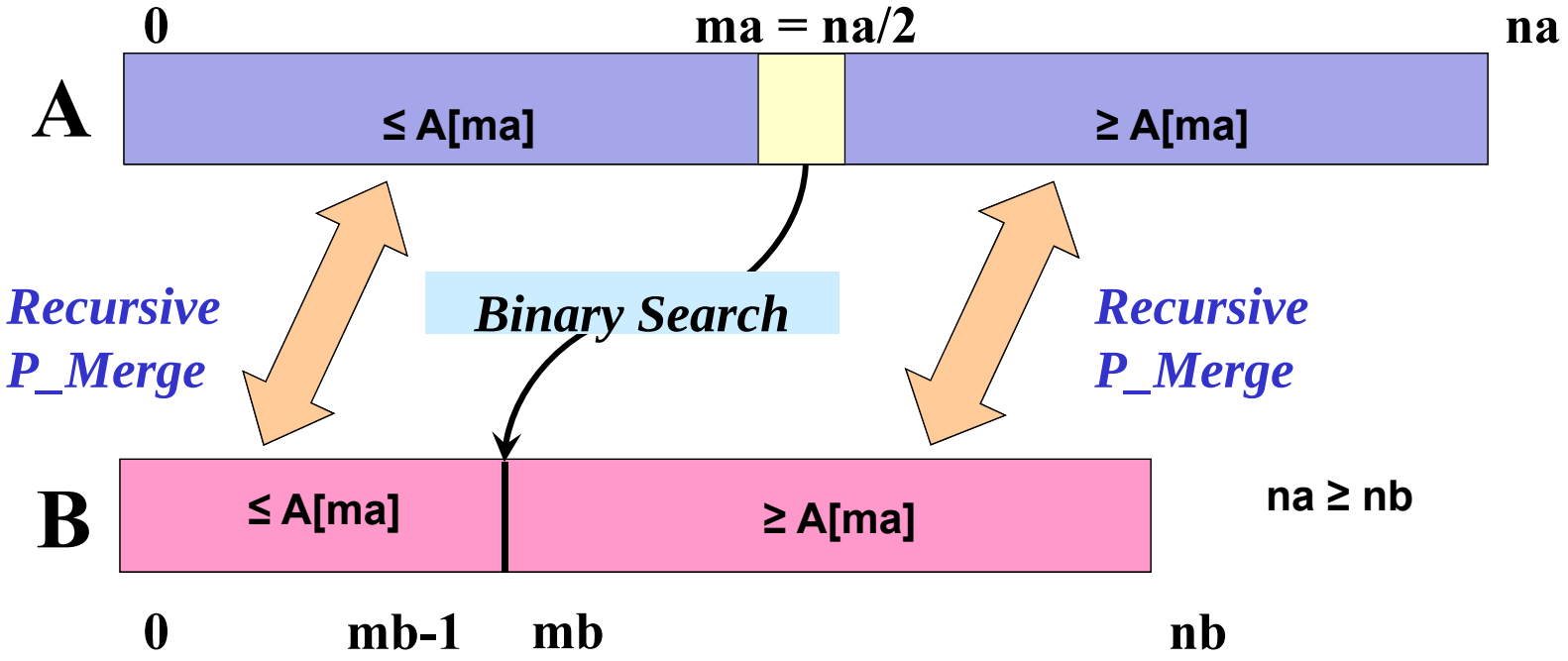
Span: $T_\infty(n) = \Theta(n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$$

We need to parallelize the merge!

source: Aydin Buluc, LBNL

Parallel Merge



$k = ma + mb$
$= na/2 + mb$
$\leq na/2 + nb$
$\leq (na+nb)/2 + (na+nb)/4$
$= ¾(na+nb)$

Smallest k for mb = 0, then
$k = ma = na/2$
$\geq (na + nb)/4$ *because* $na/4 \geq nb/4$

$\Rightarrow (na+nb)/4 \leq k \leq ¾(na+nb)$

source: Aydin Buluc, LBNL

Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    P_Merge(C, B, A, nb, na);
  } else if (na==0) {
      return;
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    #pragma omp task
        P_Merge(C, A, B, ma, mb);
    #pragma omp task
        P_Merge(C+ma+mb+1,A+ma+1,B+mb,na-ma-1,nb-mb);
    #pragma omp taskwait;
  }
}
```


source: Aydin Buluc, LBNL

Span of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    #pragma omp task
        P_Merge(C, A, B, ma, mb);
    #pragma omp task
        P_Merge(C+ma+mb+1,A+ma+1,B+mb,na-ma-1,nb-mb);
    #pragma omp taskwait;
  }
}
```

Span: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

source: Aydin Buluc, LBNL

Work of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    #pragma omp task
        P_Merge(C, A, B, ma, mb);
    #pragma omp task
            P_Merge(C+ma+mb+1,A+ma+1,B+mb,na-ma-1,nb-mb);
    #pragma omp taskwait;
  }
}
```

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$, where $1/4 \le \alpha \le 3/4$.

Claim: $T_1(n) = \Theta(n)$.

source: Aydin Buluc, LBNL

Parallelism of P_Merge

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

source: Aydin Buluc, LBNL

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
    #pragma omp task
        P_MergeSort(C, A, n/2);
    #pragma omp task
        P_MergeSort(C+n/2, A+n/2, n-n/2);
    #pragma omp taskwait;
    P_Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

source: Aydin Buluc, LBNL

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
    #pragma omp task
        P_MergeSort(C, A, n/2);
    #pragma omp task
        P_MergeSort(C+n/2, A+n/2, n-n/2);
    #pragma omp taskwait;
    P_Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$

$= \Theta(\lg^3 n)$

source: Aydin Buluc, LBNL

Parallelism of P_MergeSort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

$\Theta(n/\lg^2 n)$ is a lot larger than original $\Theta(\lg n)$ with serial merge.

source: Aydin Buluc, LBNL

Summary

- OpenMP is quasi-standard for shared memory programming based on fork-join model
- Parallel region and work sharing constructs
  - Declaration of private or shared variables
  - Reduction operations
  - Scheduling strategies
- Synchronization via barrier, critical section, atomic, and locks
- Thread affinity
- Tasks
- SIMD parallelism
- Example codes

Sources of Foils

- Christian Terboven, RWTH Aachen
- Michael Klemm, Intel
- Xavier Teruel, BSC
- Bronis R. de Supinski, Lawrence Livermore Natl. Lab.
- Michael Gerndt, TUM
- Tim Mattson, Intel
- Yang-Suk Kee, University of Southern California, San Diego.