OpenMP* in Action

Tim Mattson
Intel Corporation

Barbara Chapman
University of Houston

Acknowledgements:

Rudi Eigenmann of Purdue, Sanjiv Shah of Intel and others too numerous to name have contributed content for this tutorial.

1

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

2

OpenMP* Overview:

`C$OMP FLUSH`

`#pragma omp critical`

`C$OMP THREADPRIVATE(/ABC/)`

`CALL OMP_SET_NUM_THREADS(10)`

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

`C$OMP PARALLEL COPYIN(/blk/)`

`C$OMP DO lastprivate(XX)`

`Nthrds = OMP_GET_NUM_PROCS()`

`omp_set_lock(lck)`

3

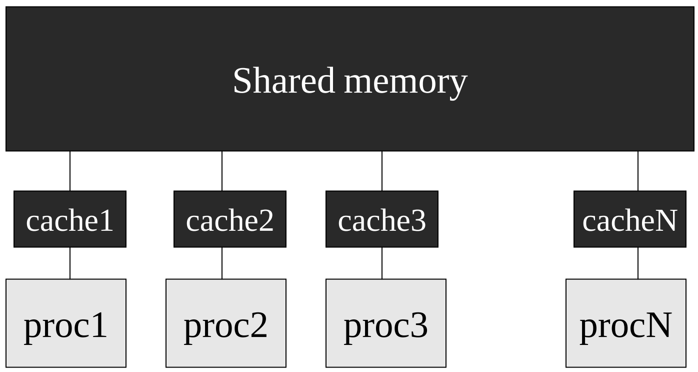
* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

What are Threads?

- Thread: an independent flow of control
  - Runtime entity created to execute sequence of instructions
- Threads require:
  - A program counter
  - A register state
  - An area in memory, including a call stack
  - A thread id
- A process is executed by one or more threads that share:
  - Address space
  - Attributes such as UserID, open files, working directory, etc.

4

A Shared Memory Architecture



5

How Can We Exploit Threads?

- A thread programming model must provide (at least) the means to:
  - Create and destroy threads
  - Distribute the computation among threads
  - Coordinate actions of threads on shared data
  - (usually) specify which data is shared and which is private to a thread

6

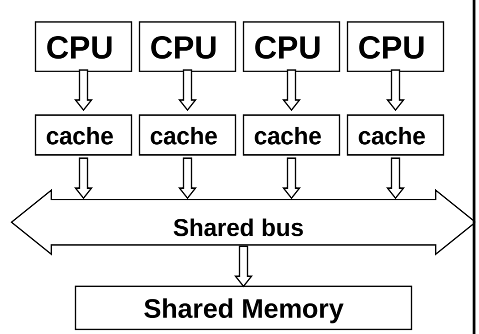
How Does OpenMP Enable Us to Exploit Threads?

- OpenMP provides thread programming model at a “high level”.
  - The user does not need to specify all the details
    - Especially with respect to the assignment of work to threads
    - Creation of threads
- User makes strategic decisions
- Compiler figures out details
- Alternatives:
  - MPI
  - POSIX thread library is lower level
  - Automatic parallelization is even higher level (user does nothing)
    - But usually successful on simple codes only

7

Where Does OpenMP Run?

Hardware Platforms	OpenMP support
Shared Memory Systems	Available
Distributed Shared Memory Systems (ccNUMA)	Available
Distributed Memory Systems	Available via Software DSM
Machines with Chip MultiThreading	Available



Shared memory architecture

8

OpenMP Overview:
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

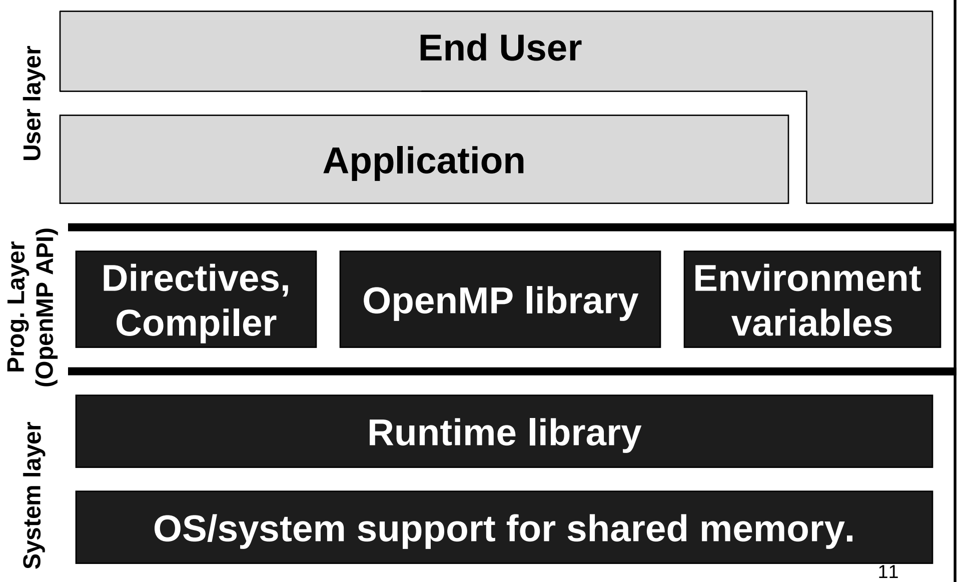
9

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

10

OpenMP Parallel Computing Solution Stack



11

OpenMP:
Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives.
  - For C and C++, the directives are pragmas with the form:
    #pragma omp *construct [clause [clause]…]*
  - For Fortran, the directives are comments and take one of the forms:
    - Fixed form
      *$OMP *construct [clause [clause]…]*
      C$OMP *construct [clause [clause]…]*
    - Free form (but works for fixed form too)
      !$OMP *construct [clause [clause]…]*
- Include file and the OpenMP lib module
  #include <omp.h>
  use omp_lib

12

OpenMP:
Structured blocks (C/C++)

- **Most OpenMP* constructs apply to structured blocks.**
  - **Structured block: a block with one point of entry at the top and one point of exit at the bottom.**
  - **The only "branches" allowed are STOP statements in Fortran and exit() in C/C++.**

```
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
        if(!conv(res[id]) goto more;
}
   printf(" All done \n");
```

```
      if(go_now()) goto more;
#pragma omp parallel
{
        int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
        if(conv(res[id]) goto done;
        go to more;
}
done:     if(!really_done()) goto more;
```

A structured block | **<u>Not</u> A structured block**

OpenMP:
Structured blocks (Fortran)

– **Most OpenMP constructs apply to structured blocks.**
  - **Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.**
  - **The only "branches" allowed are STOP statements in Fortran and exit() in C/C++.**

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(.not.conv(res(id)) goto 10
C$OMP END PARALLEL
     print *,id
```

```
C$OMP  PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id))goto 20
      go to 10
C$OMP END PARALLEL
     if(not_DONE) goto 30
20    print *, id
```

14

A structured block | **<u>Not</u> A structured block**

OpenMP:
Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
    id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

```
#pragma omp for
  for(I=0;I<N;I++){
      res[I] = big_calc(I);
      A[I] = B[I] + res[I];
  }
```

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(.not.conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
     do I=1,N
          res(I)=bigComp(I)
      end do
C$OMP END PARALLEL DO
```

5

OpenMP Definitions:
"Constructs" vs. "Regions" in OpenMP

OpenMP constructs occupy a single compilation unit while a region can span multiple source files.

poo.f

```
      call whoami
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
```

bar.f

```
      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

+

A Parallel *construct*

The Parallel *region* is the text of the construct plus any code called inside the construct

Orphan constructs can execute outside a parallel region

16

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

17

The OpenMP* API

Parallel Regions

- You create threads in OpenMP* with the "omp parallel" pragma.
- For example, To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- **Each thread calls pooh(ID,A) for `ID` = `0` to `3`**

18

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

The OpenMP* API

Parallel Regions

- You create threads in OpenMP* with the “omp parallel” pragma.
- For example, To create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

Runtime function returning a thread ID

- **Each thread calls pooh(ID,A) for `ID` = `0` to `3`**

19

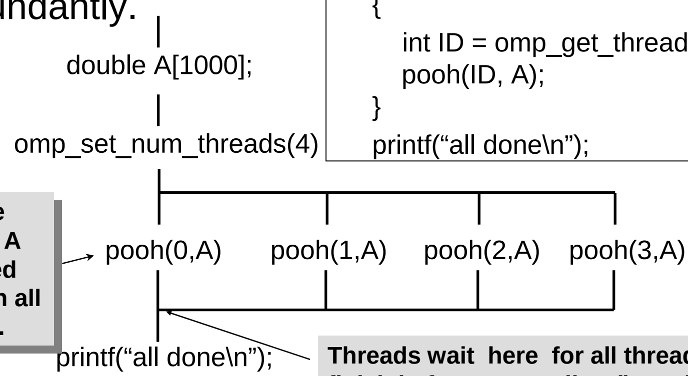
* The name “OpenMP” is the property of the OpenMP Architecture Review Board

The OpenMP* API

Parallel Regions

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

Exercise:

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
void main()
{

    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

21

Exercise:

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

22

Parallel Regions and the "if" clause

Active vs *inactive* parallel regions.

- An optional if clause causes the parallel region to be active only if the logical expression within the clause evaluates to true.
- An if clause that evaluates to false causes the parallel region to be inactive (i.e. executed by a team of size one).

```
double A[N];

#pragma omp parallel if(N>1000)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

23

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

24

OpenMP: Work-Sharing Constructs

- The "for" Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

By default, there is a barrier at the end of the "omp for". Use the "nowait" clause to turn off the barrier.

#pragma omp for nowait

"nowait" is useful between two consecutive, independent omp for loops.

25

Work Sharing Constructs

A motivating example

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
        for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP For/Do construct:
The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - **schedule(dynamic[,chunk])**
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - **schedule(guided[,chunk])**
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - **schedule(runtime)**
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

27

The OpenMP API

The schedule clause

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

Least work at runtime : scheduling done at compile-time

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

28

OpenMP: Work-Sharing Constructs

- The *Sections* work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
        X_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

29

OpenMP: Work-Sharing Constructs

- The master construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma barrier
        do_many_other_things();
}
```

30

OpenMP: Work-Sharing Constructs

- The single construct denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

31

The OpenMP* API

Combined parallel/work-share

- OpenMP* shortcut: Put the "parallel" and the work-share on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
```

These are equivalent

- **There's also a "parallel sections" construct.**

32

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

33

Data Environment: Default storage attributes

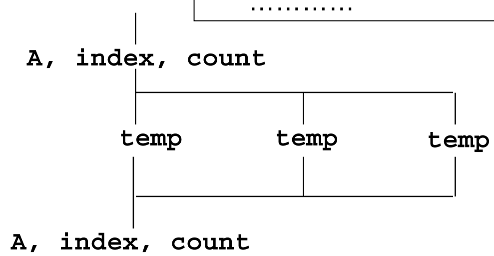
- **Shared Memory programming model:**
  - **Most variables are shared by default**
- **Global variables are SHARED among threads**
  - **Fortran: COMMON blocks, SAVE variables, MODULE variables**
  - **C: File scope variables, static**
- **But not everything is shared...**
  - **Stack variables in sub-programs called from parallel regions are PRIVATE**
  - **Automatic variables within a statement block are PRIVATE.**

34

Data Sharing Examples

```
      program sort
      common /input/ A(10)
      integer index(10)
!$OMP PARALLEL
      call work(index)
!$OMP END PARALLEL
      print*, index(1)
```

```
subroutine work (index)
common /input/ A(10)
integer index(*)
real temp(10)
integer count
save count
   …………
```



A, index and count are shared by all threads.

temp is local to each thread

35

* Third party trademarks and names are the property of their respective owner.

Data Environment:

Changing storage attributes

- One can selectively change storage attributes constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE

All the clauses on this page apply to the *OpenMP Construct* NOT the entire region.

- The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
  - LASTPRIVATE
- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

All data clauses apply to parallel regions and worksharing constructs except “shared” which only applies to parallel regions.

36

Private Clause

- private(var) creates a local copy of var for each thread.
  - The value is uninitialized
  - Private copy is *not* storage-associated with the original
  - The original is undefined at the end

```
      program wrong
      IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
      DO J=1,1000
           IS = IS + J
      END DO
      print *, IS
```

Regardless of initialization, IS is undefined at this point

IS was not initialized

37

Firstprivate Clause

- Firstprivate is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```
      program almost_right
      IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

Each thread gets its own IS with an initial value of 0

Regardless of initialization, IS is undefined at this point

38

Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```
      program closer
      IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP+ LASTPRIVATE(IS)
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

Each thread gets its own IS with an initial value of 0

IS is defined as its value at the "last sequential" iteration (I.e. for J=1000)

39

OpenMP: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
      variables A,B, and C = 1
C$OMP PARALLEL PRIVATE(B)
C$OMP& FIRSTPRIVATE(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and after the parallel region?

Inside this parallel region ...

- **"A" is shared by all threads; equals 1**
- **"B" and "C" are local to each thread.**
  - **B's initial value is undefined**
  - **C's initial value equals 1**

Outside this parallel region ...

- **The values of "B" and "C" are undefined.**

40

OpenMP: Reduction

- Combines an accumulation operation across threads:

  reduction (op : list)
- 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 "+").
  - Compiler finds standard reduction expressions containing "op" and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in "list" must be shared in the enclosing parallel region.

41

OpenMP: Reduction example

- Remember the code we used to demo private, firstprivate and lastprivate.

```
      program closer
      IS = 0
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

- **Here is the correct way to parallelize this code.**

```
      program closer
      IS = 0
#pragma omp parallel for reduction(+:IS)
      DO J=1,1000
           IS = IS + J
1000  CONTINUE
      print *, IS
```

42

OpenMP:
Reduction operands/initial-values

- A range of associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operand	Initial value
+	0
*	1
-	0
.AND.	.true.
.OR.	.false.
.neqv.	.false.
.eqv.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

Operand	Initial value
iand	All bits on
ior	0
ieor	o
&	~0
|	0
^	0
&&	1
||	0

43

* Min and Max are not available in C/C++

Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to use it)
- To change default: DEFAULT(PRIVATE)
  - *each* variable in *static* extent of the parallel region is made private as if specified in a private clause
  - mostly saves typing
- DEFAULT(NONE)*:* *no* default for variables in static extent. Must list storage attribute for each variable in static extent

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Default Clause Example

```
        itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      ………
C$OMP END PARALLEL
```

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      ………
C$OMP END PARALLEL
```

Are these two codes equivalent?

yes

45

Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.

46

A threadprivate example

Consider two different routines called within a parallel region.

```
      subroutine poo
      parameter (N=1000)
      common/buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
      do i=1, N
        B(i)= const* A(i)
      end do
      return
      end
```

```
      subroutine bar
      parameter (N=1000)
      common/buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
      do i=1, N
        A(i) = sqrt(B(i))
      end do
      return
      end
```

Because of the threadprivate construct, each thread executing these routines has its own copy of the common block /buf/.

47

Copyin

You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

48

Copyprivate

Used with a single region to broadcast values of privates from one member of a team to the rest of the team.

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
  int Nsize, choice;

  #pragma omp parallel private (Nsize, choice)
  {
      #pragma omp single copyprivate (Nsize, choice)
            input_parameters (Nsize, choice);

      do_work(Nsize, choice);
  }
}
```

Agenda

- Parallel Computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case Studies and Examples
- Background information and extra details

50

OpenMP: Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

51

OpenMP: Synchronization

- Only one thread at a time can enter a critical region.

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
        DO 100 I=1,NITERS
                B =  DOIT(I)
C$OMP CRITICAL
                CALL CONSUME (B, RES)
C$OMP END CRITICAL
100     CONTINUE
```

52

The OpenMP* API

Synchronization – critical (in C/C++)

- Only one thread at a time can enter a critical region.

```
float res;
#pragma omp parallel
{     float B;   int i;
      #pragma omp for
      for(i=0;i<niters;i++){
             B =  big_job(i);
#pragma omp critical
             consume (B, RES);
      }
}
```

Threads wait their turn – only one at a time calls consume()

53

OpenMP: Synchronization

- Atomic provides mutual exclusion execution but only applies to the update of a memory location (the update of X in the following example)

```
C$OMP PARALLEL PRIVATE(B)
        B =  DOIT(I)
        tmp = big_ugly();
 C$OMP ATOMIC
        X = X + temp
C$OMP END PARALLEL
```

54

OpenMP: Synchronization

- Barrier: **Each thread waits until all threads arrive.**

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc3(id);
}
```

implicit barrier at the end of a for work-sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

OpenMP: Synchronization

- The ordered region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

56

OpenMP:

Implicit synchronization

- Barriers are implied on the following OpenMP constructs:

```
end parallel
end do  (except when nowait is used)
end sections (except when nowait is used)
end single (except when nowait is used)
```

57

OpenMP: Synchronization

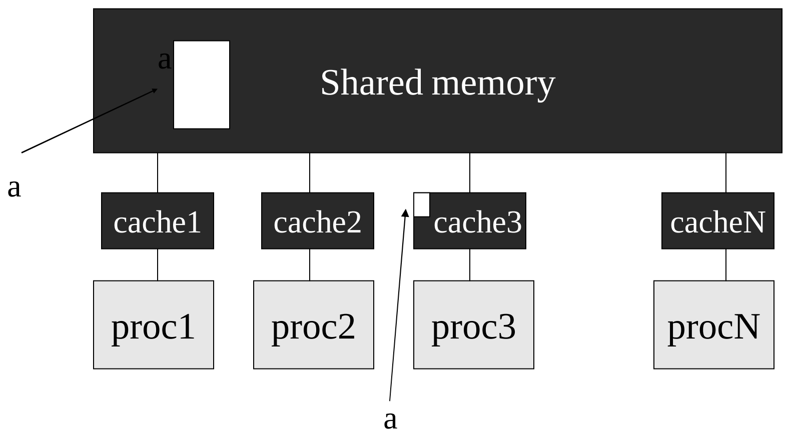
- The flush construct denotes a sequence point where a thread tries to create a consistent view of memory for a subset of variables called the *flush set*.
- Arguments to flush define the flush set:

  `#pragma omp flush(A, B, C)`
- The flush set is all *thread visible variables* if no argument list is provided

  `#pragma omp flush`
- For the variables in the flush set:
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.

58

Shared Memory Architecture



59

OpenMP: A flush example

- This example shows how flush is used to implement pair-wise synchronization.

```
        integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
        IAM = OMP_GET_THREAD_NUM()
        ISYNC(IAM) = 0
C$OMP BARRIER
        CALL WORK()
        ISYNC(IAM) = 1   ! I'm all done; signal this to other threads
C$OMP FLUSH(ISYNC)
        DO WHILE (ISYNC(NEIGH) .EQ. 0)
C$OMP FLUSH(ISYNC)
        END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write.

Make sure the read picks up a good copy from memory.

Note: OpenMP's flush is analogous to a fence in other shared memory API's.

60

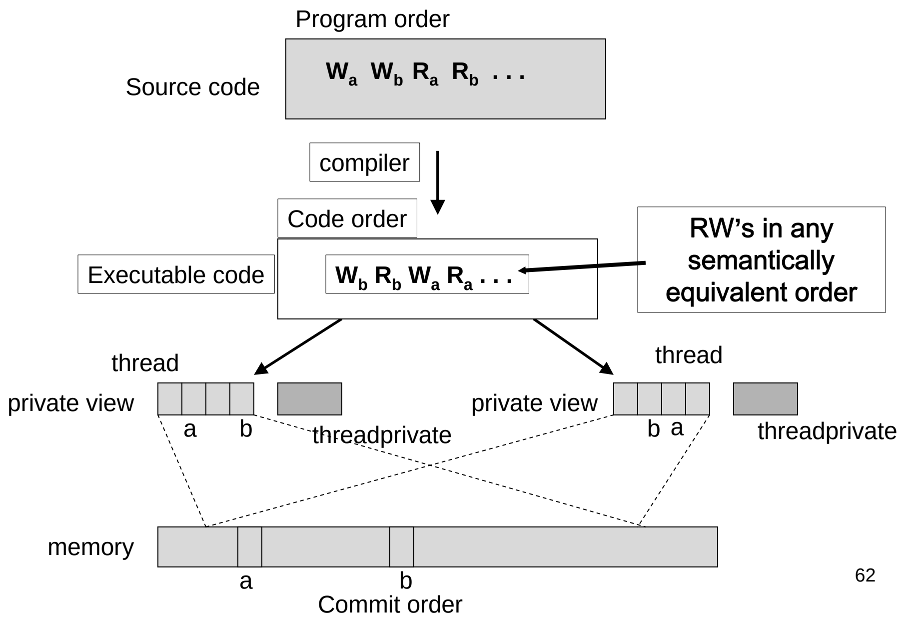
What is the Big Deal with Flush?

- Compilers reorder instructions to better exploit the functional units and keep the machine busy
- Flush interacts with instruction reordering:
  - A compiler CANNOT do the following:
    - Reorder read/writes of variables in a flush set relative to a flush.
    - Reorder flush constructs when flush sets overlap.
  - A compiler CAN do the following:
    - Reorder instructions NOT involving variables in the flush set relative to the flush.
    - Reorder flush constructs that don't have overlapping flush sets.
- So you need to use flush carefully

Also, the flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory.

61

OpenMP Memory Model: Basic Terms



62

OpenMP: Lock routines

In OpenMP 2.5, a lock implies a flush of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

Note: a thread always accesses the most recent copy of the lock, so you don’t need to use a flush on the lock variable.

63

OpenMP: Simple Locks

- Protect resources with locks.

```
    omp_lock_t lck;
    omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
      printf(“%d %d”, id, tmp);
    omp_unset_lock(&lck);
}
   omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

OpenMP: Nested Locks

```
#include <omp.h>
typedef struct{int a,b; omp_nest_lock_t lck;} pair;

void incr_a(pair *p, int a) { p->a +=a;}

void incr_b (pair *p, int b)
{ omp_set_nest_lock(&p->lck); p->b =+b; omp_unset_nest_lock(&p->lck);}

void incr_pair(pair *p, int a, int b)
{omp_set_nest_lock(&p->lck); incr_a(p,a); incr_b(p,b); omp_unset_nest_lock(&p->lck);}

void f(pair *p)
{ extern int work1(), work2(), work3();
#pragma omp parallel sections
  {
  #pragma omp section
    incr_pair(p,work1(), work2());
  #pragma omp section
    incr_b(p,work3());
  }
}
```

f() calls incr_b() and incr_pair() … but incr_pair() calls incr_b() too, so you need nestable locks

Synchronization challenges

- OpenMP only includes high level synchronization directives that "have a sequential reading". Is that enough?
  - Do we need conditions variables?
  - Monotonic flags?
  - Other pairwise synchronization?
- The flush is subtle … even experts get confused on when it is required.

66

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

67

OpenMP: Library routines:

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

…plus several less commonly used routines.

68

OpenMP: Library Routines

- To used a fixed, known number of threads used in a program, (1) set the number threads, then (2) save the number you got.

```
#include <omp.h>
void main()
{   int num_threads;
    omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
    {     int id=omp_get_thread_num();
#pragma omp single
              num_threads = omp_get_num_threads();
           do_lots_of_stuff(id);
     }
}
```

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

69

OpenMP: Environment Variables:

- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - OMP_SCHEDULE "schedule[, chunk_size]"
- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*

… Plus several less commonly used environment variables.

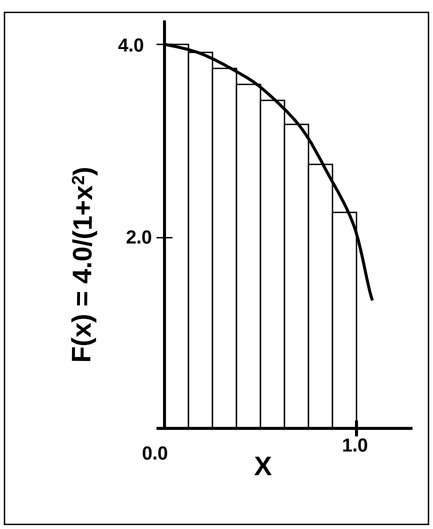
70

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background Information and extra details

71

Let's pause for a quick recap by example: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

72

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{         int i;   double x, pi, sum = 0.0;

          step = 1.0/(double) num_steps;

          for (i=0;i<= num_steps; i++){
                   x = (i+0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

73

OpenMP recap: Parallel Region

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, id, nthreads;  double x, pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x)
{
           id = omp_get_thread_num();
        #pragma omp single
             nthreads = omp_get_num_threads();
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthreads){
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
}
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

You can't assume that you'll get the number of threads you requested.

Prevent write conflicts with the single.

Performance will suffer due to false sharing of the sum array.

74

OpenMP recap: Synchronization (critical region)

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            int i, id, nthreads;  double x, pi, sum;
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x, sum)
{
             id = omp_get_thread_num();
            #pragma omp single
                 nthreads = omp_get_num_threads();
             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
             }
           #pragma omp critical
                  pi += sum * step;
}
}
```

No array, so no false sharing.

Note: this method of combining partial sums doesn't scale very well.

OpenMP recap : Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            int i;    double x, pi, sum = 0.0;
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction(+:sum)
             for (i=0;i<= num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
             }
             pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 4 simple lines!

OpenMP recap :

Use Environment variables to set number of threads

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{          int i;    double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
           for (i=0;i<= num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

In practice, you set number of threads by setting the environment variable, OMP_NUM_THREADS

77

MPI: Pi program

```
#include <mpi.h>
static long num_steps = 100000;
void main (int argc, char *argv[])
{
          int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
          step = 1.0/(double) num_steps ;
          MPI_Init(&argc, &argv) ;
          MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
          MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
          for (i=my_id; i<num_steps ; i+numprocs)
          {
                    x = (i+0.5)*step;
                    sum += 4.0/(1.0+x*x);
          }
          sum *= step ;
          MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                              MPI_COMM_WORLD) ;
}
```

78

Pi Program: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;


  start = *(int *) arg;
  step = 1.0/(double) num_steps;

  for (i=start;i<= num_steps; i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
          thread_handles[i] = CreateThread(0, 0,
                         (LPTHREAD_START_ROUTINE) Pi,
                         &threadArg[i], 0, &threadID);
}

  WaitForMultipleObjects(NUM_THREADS,
                         thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Doubles code size!

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

80

How is OpenMP Compiled ?

Most Fortran/C compilers today implement OpenMP

- The user provides the required switch or switches
- Sometimes this also needs a specific optimization level, so manual should be consulted
- May also need to set threads' stacksize explicitly

Examples

- Commercial: -openmp (Intel, Sun, NEC), -mp (SGI, PathScale, PGI), --openmp (Lahey, Fujitsu), -qsmp=omp (IBM) /openmp flag (Microsoft Visual Studio 2005), etc.
- Freeware: Omni, OdinMP, OMPi, Open64.UH, …

Check information at http://www.compunity.org

81

Under the Hood: How Does OpenMP Really Work?

Programmer

- States what is to be carried out in parallel by multiple threads
- Gives strategy for assigning work to threads
- Arranges for threads to synchronize
- Specify data sharing attributes: shared, private, firstprivate, threadprivate,…

Compiler (with the help of a runtime library)

- Transforms OpenMP programs into multi-threaded code
- Manages threads: creates, suspends, wakes up, terminates threads
- Figures out the details of the work to be performed by each thread
- Implements thread synchronization
- Arranges storage for different data and performs
- their initializations: shared, private…

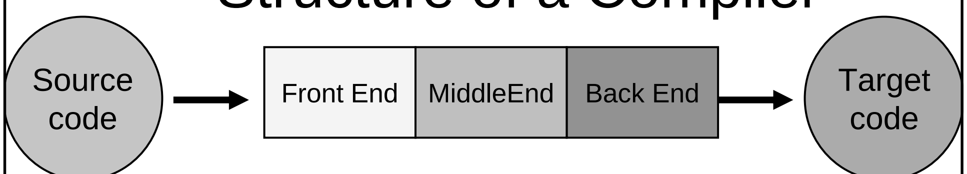
The details of how OpenMP is implemented varies from one compiler to another. We can only give an idea of how it is done here!!

Overview of OpenMP Translation Process

- Compiler processes directives and uses them to create explicitly multithreaded code
- Generated code makes calls to a runtime library
  - The runtime library also implements the OpenMP user-level run-time routines
- Details are different for each compiler, but strategies are similar
- Runtime library and details of memory management also proprietary
- Fortunately the basic translation is not all that difficult

83

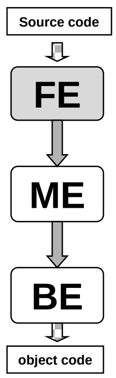
Structure of a Compiler



- **Front End**:
  - Read in source program, ensure that it is error-free, build the intermediate representation (IR)
- **Middle End**:
  - Analyze and optimize program. "Lower" IR to machine-like form
- **Back End**:
  - Complete optimization. Determine layout of program data in memory. Generate object code for the target architecture

84

OpenMP Compiler Front End



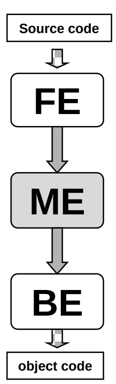
In addition to reading in the base language (Fortran, C or C++)

- Parse OpenMP directives
- Check them for correctness
  - Is directive in the right place? Is the information correct? Is the form of the for loop permitted? ….
- Create an intermediate representation with OpenMP annotations for further handling

Nasty problem: incorrect OpenMP sentinel means directive may not be recognized. And there might be no error message!!

85

OpenMP Compiler Middle End

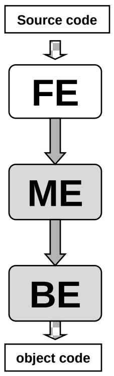


- Preprocess OpenMP constructs
  - Translate SECTIONs to DO/FOR constructs
  - Make implicit BARRIERs explicit
  - Apply even more correctness checks
- Apply some optimizations to code to ensure it performs well
  - Merge adjacent parallel regions
  - Merge adjacent barriers

OpenMP directives reduce scope in which some optimizations can be applied. Compiler writer must work hard to avoid a negative impact on performance.

86

OpenMP Compiler: Rest of Processing



- Translate OpenMP constructs
  - Simple direct one-to-one substitution
    - Replace certain OpenMP constructs by calls to runtime routines.
    - e.g.: barrier, atomic, flush, etc
  - Complex code transformation: to get multi-threaded code with calls to runtime library
    - For slave threads: create a separate task that contains the code in a parallel region
    - For master thread: fork slave threads so they execute their tasks, as well as carrying out the task along with slave threads.
    - Add necessary synchronization via runtime library
    - Translate parallel and worksharing constructs and clauses e.g.: parallel, for, etc
- Also implement variable data attributes, set up storage and arrange for initialization
  - Thread’s stack might be used to hold all private data
  - Instantiate new variables to realize private, reduction, etc
  - Add assignment statements to realize firstprivate, lastprivate, etc

87

Outlining

Create a procedure containing the region enclosed by a parallel construct

- Shared data passed as arguments
  - Referenced via their address in routine
- Private data stored on thread’s stack
  - Threadprivate may be on stack or heap
- Visible during debugging
- An alternative is called microtasking

Outlining introduces a few overheads, but makes the translation comparatively straightforward. It makes the scope of OpenMP data attributes explicit.

88

An Outlining Example: Hello world

- Original Code

```
#include <omp.h>
void main()
{
#pragma omp parallel
 {
 int
  ID=omp_get_thread_num
  ();
 printf("Hello
  world(%d)",ID);
 }
}
```

- Translated multi-threaded code with runtime library calls

```
//here is the outlined code
void __ompregion_main1(…)
{
 int ID =ompc_get_thread_num();
 printf("Hello world(%d)",ID);
} /* end of ompregion_main1*/

void main()
{
…
__ompc_fork(&__ompregion_main1
   ,…);
…
}
```

89

OpenMP Transformations – Do/For

- Transform original loop so each thread performs only its own portion
- Most of scheduling calculations usually hidden in runtime
- Some extra work to handle firstprivate, lastprivate

- Original Code

```
#pragma omp for
for( i = 0; i < n; i++ )
{  …}
```

- Transformed Code

```
  tid = ompc_get_thread_num();
  ompc_static_init (tid,
   lower,uppder, incr,.);
  for( i = lower;i < upper;i +=
   incr ) { … }
// Implicit BARRIER
  ompc_barrier();
```

90

OpenMP Transformations – Reduction

- Reduction variables can be translated into a two-step operation
- First, each thread performs its own reduction using a private variable
- Then the global sum is formed
- A critical construct might be used to ensure atomicity of the final reduction

- Original Code

```
#pragma omp parallel for \
reduction (+:sum) private (x)
for(i=1;i<=num_steps;i++)
{  …
sum=sum+x ;}
```

- Transformed Code

```
float local_sum;
…
ompc_static_init (tid, lower,uppder,
 incr,.);
for( i = lower;i < upper;i += incr )
 { … local_sum = local_sum +x;}
ompc_barrier();
ompc_critical();
sum = (sum + local_sum);
 ompc_end_critical();
```

91

OpenMP Transformation –Single/Master

- Master thread has a threadid of 0, very easy to test for.
- The runtime function for the single construct might use a lock to test and set an internal flag in order to ensure only one thread get the work done

- Original Code

```
#pragma omp parallel
{#pragma omp master
  a=a+1;
#pragma omp single
  b=b+1;}
```

- Transformed Code

```
Is_master= ompc_master(tid);
if((Is_master == 1))
{   a = a + 1;  }
Is_single = ompc_single(tid);
if((Is_single == 1))
{   b = b + 1;  }
ompc_barrier();
```

92

OpenMP Transformations – Threadprivate

- Every threadprivate variable reference becomes an indirect reference through an auxiliary structure to the private copy
- Every thread needs to find its index into the auxiliary structure – This can be expensive
  - Some OS'es (and codegen schemes) dedicate register to identify thread
  - Otherwise OpenMP runtime has to do this

- Original Code

```
static int px;

int foo() {
  #pragma omp threadprivate(px)
   bar( &px );
  }
```

- Transformed Code

```
static int px;
static int ** thdprv_px;

int _ompregion_foo1() {
int* local_px;
  …
tid = ompc_get_thread_num();
local_px=get_thdprv(tid,thdprv_px,
  &px);
bar( local_px );
}
```

93

OpenMP Transformations – WORKSHARE

- WORKSHARE can be translated to OMP DO during preprocessing phase
- If there are several different array statements involved, it requires a lot of work by the compiler to do a good job
- So there may be a performance penalty

- Original Code

```
      REAL AA(N,N), BB(N,N)
     !$OMP PARALLEL
     !$OMP WORKSHARE
           AA = BB
     !$OMP END WORKSHARE
     !$OMP END PARALLEL
```

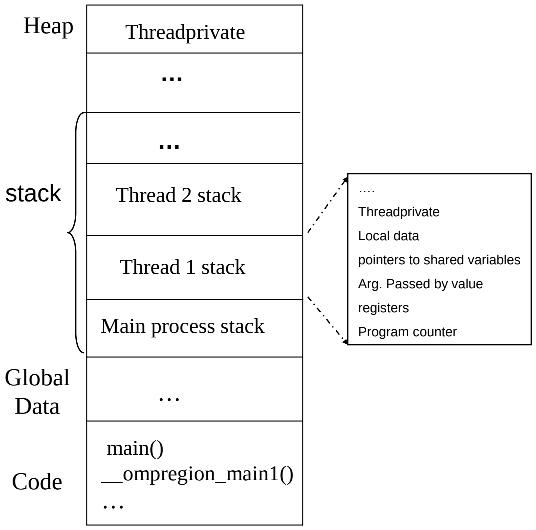
- Transformed Code

```
  REAL AA(N,N), BB(N,N)
 !$OMP PARALLEL
 !$OMP DO
    DO J=1,N,1
       DO I=1,N,1
         AA(I,J) = BB(I,J)
      END DO
   END DO
 !$OMP END PARALLEL
```

94

Runtime Memory Allocation

One possible organization of memory



- Outlining creates a new scope: private data become local variables for the outlined routine.
- Local variables can be saved on stack
  - Includes compiler-generated **temporaries**
  - **Private** variables, including **firstprivate** and **lastprivate**
  - Could be a lot of data
  - Local variables in a procedure called within a parallel region are private by default
- **Location of threadprivate data depends on implementation**
  - On heap
  - On local stack

95

Role of Runtime Library

- Thread management and work dispatch
  - Routines to create threads, suspend them and wake them up/ spin them, destroy threads
  - Routines to schedule work to threads
    - Manage queue of work
    - Provide schedulers for static, dynamic and guided
- Maintain internal control variables
  - threadid, numthreads, dyn-var, nest-var, sched_var, etc
- Implement library routines omp_..() and some simple constructs (e.g. barrier, atomic)

Some routines in runtime library – e.g. to return the threadid - are heavily accessed, so they must be carefully implemented and tuned. The runtime library should avoid any unnecessary internal synchronization.

96

Synchronization

- Barrier is main synchronization construct since many other constructs may introduce it implicitly. It in turn is often implemented using locks.

One simple way to implement barrier

- **Each thread team maintains a barrier counter and a barrier flag.**
- **Each thread increments the barrier counter when it enters the barrier and waits for a barrier flag to be set by the last one.**
- **When the last thread enters the barrier and increment the counter, the counter will be equal to the team size and the barrier flag is reset.**
- **All other waiting threads can then proceed.**

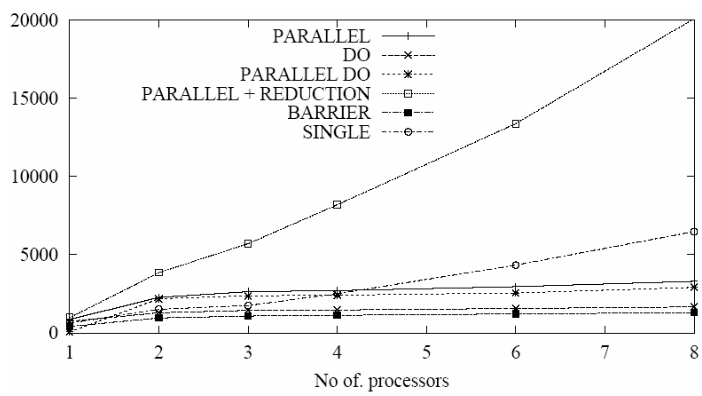
```
void __ompc_barrier (omp_team_t *team)
{
     ...
     pthread_mutex_lock(&(team->barrier_lock));
     team->barrier_count++;
     barrier_flag = team->barrier_flag;

    /* The last one reset flags*/
     if (team->barrier_count == team->team_size)
     {
          team->barrier_count = 0;
           team->barrier_flag = barrier_flag ^ 1; /* Xor: toggle*/
           pthread_mutex_unlock(&(team->barrier_lock));
           return;
     }
     pthread_mutex_unlock(&(team->barrier_lock));

     /* Wait for the last to reset the barrier*/
     OMPC_WAIT_WHILE(team->barrier_flag == barrier_flag);
}
```

97

Constructs That Use a Barrier



Synchronization Overheads (in cycles) on SGI Origin 2000*

- Careful implementation can achieve modest overhead for most synchronization constructs.
- Parallel reduction is costly because it often uses critical region to summarize variables at the end.

98

*** Courtesy of J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", EWOMP '99, Lund, Sep., 1999**

Static Scheduling: Under The Hood

```
// The OpenMP code
// possible unknown loop upper bound: n
// unknown number of threads to be used
#pragma omp for schedule(static)
  for (i=0;i<n;i++)
   {
    do_sth();
   }
```

```
/ *Static even: static without specifying
    chunk size; scheduler divides loop
    iterations evenly onto each thread. */
// the outlined task for each thread
 _gtid_s1 = __ompc_get_thread_num();
  temp_limit = n – 1
  __ompc_static_init(_gtid_s1 , static,
     &_do_lower, &_do_upper,
     &_do_stride,..);
  if(_do_upper > temp_limit)
  {     _do_upper = temp_limit;  }
 for(_i = _do_lower; _i <= _do_upper; _i ++)
  {
       do_sth();
  }
```

- **Most (if not all) OpenMP compilers choose static as default scheduling method**
- **Number of threads and loop bounds possibly unknown, so final details usually deferred to runtime**
- **Two simple runtime library calls are enough to handle static case:**
  Constant overhead

99

Dynamic Scheduling : Under The Hood

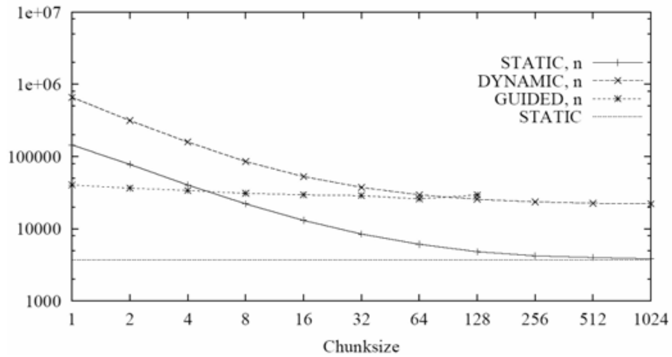
```
_gtid_s1 = __ompc_get_thread_num();
  temp_limit = n -1;                          // Schedule(dynamic, chunksize)
 _do_upper = temp_limit;
 _do_lower = 0;
 __ompc_scheduler_init(__ompv_gtid_s1, dynamic ,do_lower, _do_upper, stride,
    chunksize..);
 _i = _do_lower;
 mpni_status = __ompc_schedule_next(_gtid_s1, &_do_lower, &_do_upper, &_do_stride);
 while(mpni_status)
 {
   if(_do_upper > temp_limit)
   {      _do_upper = temp_limit;    }
   for(_i = _do_lower; _i <= _do_upper; _i = _i + _do_stride)
   {          do_sth();    }
   mpni_status = __ompc_schedule_next(_gtid_s1, &_do_lower, &_do_upper, &_do_stride);
 }
```

- **Scheduling is performed during runtime.**
- **A while loop to grab available loop iterations from a work queue**
  - **Similar way to implement STATIC with a chunk size and GUIDED scheduling**

Average overhead= c1*(iteration space/chunksize)+c2

100

Using OpenMP Scheduling Constructs



Scheduling Overheads (in cycles) on Sun HPC 3500*

- Conclusion:
  - Use default static scheduling when work load is balanced and thread processing capability is constant.
  - Use dynamic/guided otherwise

* Courtesy of J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", EWOMP '99, Lund, Sep., 1999.

101

Implementation-Defined Issues

- Each implementation must decide what is in compiler and what is in runtime
- OpenMP also leaves some issues to the implementation
  - Default number of threads
  - Default schedule and default for schedule (runtime)
  - Number of threads to execute nested parallel regions
  - Behavior in case of thread exhaustion
  - And many others..

Despite many similarities, each implementation is a little different from all others.

102

Agenda

- Parallel Computing, threads, and OpenMP
- The core elements of OpenMP
  – Thread creation
  – Workshare constructs
  – Managing the data environment
  – Synchronization
  – The runtime library
  – Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

103

Turning Novice Parallel programmers into Experts

- How do you pass-on expert knowledge to novices quickly and effectively?
  1. Understand expert knowledge, i.e. "how do expert parallel programmers think?"
  2. Express that expertise in a consistent framework.
  3. Validate (peer review) the framework so it represents a true consensus view.
  4. Publish the framework.
- The Object Oriented Software Community has found that a language of design patterns is a useful way to construct such a framework.

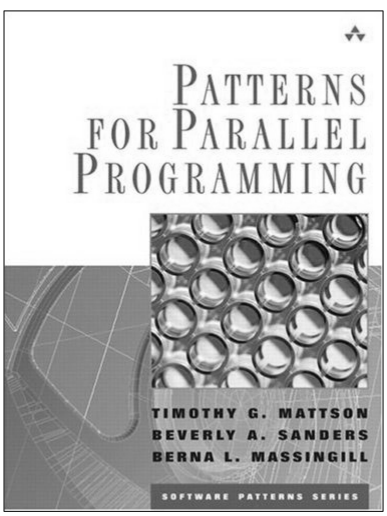
104

Design Patterns:
A silly example

- Name: Money Pipeline
- Context: **You want to get rich and all you have to work with is a C.S. degree and programming skills.** How can you use software to get rich?
- Forces: **The solution must resolve the forces:**
  - **It must give the buyer something they believe they need.**
  - **It can't be too good, or people won't need to buy upgrades.**
  - **Every good idea is worth stealing -- anticipate competition.**
- Solution**: Construct a money pipeline**
  - **Create SW with enough functionality to do something useful most of the time. This will draw buyers into your money pipeline.**
  - **Promise new features to thwart competitors.**
  - **Use bug-fixes and a slow trickle of new features to extract money as you move buyers along the pipeline.**

105

A Shameless plug



A pattern language for parallel algorithm design with examples in MPI, OpenMP and Java.

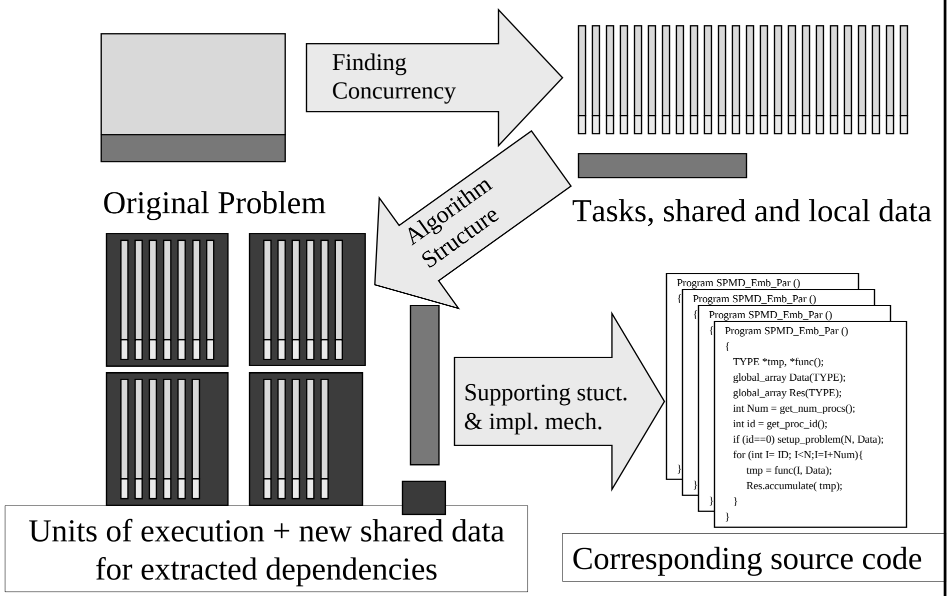
This is our hypothesis for how programmers think about parallel programming.

Now available at a bookstore near you!

106

Pattern Language's structure:

Four design spaces in parallel software development



Algorithm Structure Design space

Common patterns in OpenMP

Name: The Task Parallelism Pattern

- Context:
  - How do you exploit concurrency expressed in terms of a set of distinct tasks?
- Forces
  - Size of task – small size to balance load vs. large size to reduce scheduling overhead.
  - Managing dependencies without destroying efficiency.
- Solution
  - Schedule tasks for execution with balanced load – use master worker, loop parallelism, or SPMD patterns.
  - Manage dependencies by:
    - removing them (replicating data),
    - transforming induction variables,
    - exposing reductions,
    - explicitly protecting (shared data pattern).

108

Supporting Structures Design space

Common patterns in OpenMP

Name: The SPMD Pattern

- Context:
  - How do you structure a parallel program to make interactions between threads manageable yet easy to integrate with the core computation?
- Forces
  - Fewer programs are easier to manage, but complex algorithms often need very different instruction streams on each thread.
  - Balance the conflicting needs of scalability, maintainability, and portability.
- Solution
  - Use a single program for all the threads.
  - Keep it simple … use the threads ID to select different pathways through the program.
  - Keep interactions between threads explicit and at a minimum.

109

Common patterns in OpenMP

Supporting Structures Design space

Name: The Loop Parallelism Pattern

- Context:
  - How do you transform a serial program dominated by compute intensive loops into a parallel program without radically changing the semantics?
- Forces
  - An existing program implies expected output … and this must be preserved even as the programs execution changes due to parallelism.
  - High performance requires restructuring data to optimize for cache … but this must be done carefully to avoid changing semantics.
  - Assuring correctness suggests that the parallelization process should be incremental with each transformation subject to testing.
- Solution
  - Identify key compute intensive loops.
  - Use directives/pragmas to parallelize these loops (i.e. at no point do you use the ID to manage loop parallelization by hand).
  - Optimize incrementally testing at each step of the way – merging loops, modifying schedules, etc.

110

Design pattern example:
The SPMD pattern

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{            int i, id, nthreads;  double x, pi, sum;
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x, sum)
{
             id = omp_get_thread_num();
          #pragma omp single
                nthreads = omp_get_num_threads();
             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
             }
          #pragma omp critical
                   pi += sum * step;
}
}
```

Every thread executes the same code … use the thread ID to distribute work.

Data replication used to manage dependencies

111

Design pattern example:
The Loop Parallelism pattern

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{            int i;    double x, pi, sum = 0.0;
             step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
             for (i=0;i<= num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
             }
             pi = step * sum;
}
```

Parallelism inserted as semantically neutral directives to parallelize key loops

Reduction used to manage dependencies

112

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

113

Case Studies - General Comments: Performance Measurement

- What is the baseline performance?
- What is the desired improvement?
- Where are the computation-intensive regions of the program?
- Is timing done on a dedicated machine? If a large machine, where is data? What about file access? Does set of processors available impact this?

On large machines, it is best to use custom features to pin threads to specific resources. A poor data layout may kill performance on a platform with many CPUs.

114

Case Studies - General Comments: Performance Issues

- Overheads of OpenMP constructs, thread management
  - E.g. dynamic loop schedules have much higher overheads than static schedules
  - Synchronization is expensive, so use NOWAIT if possible and privatize data where possible
- Overheads of runtime library routines
  - Some are called frequently
- Structure and characteristics of program
  - Sequential part of program
  - Load balance
  - Cache utilization and false sharing (it can kill any speedup)
  - Large parallel regions help reduce overheads, enable better cache usage and standard optimizations

System or helper threads may also be active – managing or assigning work. Give them a resource of their own by using one less CPU than the available number. Otherwise, they may degrade performance. Pin threads to CPUs.

Case Studies - General Comments: Optimal Number of Threads

- May need to experiment to determine the optimal number of threads for a given problem
  - In SMT, *`reducing`* the number of threads may alleviate resource (register, datapath, cache, or memory) conflicts
    - Using one thread per physical processor is preferred for memory-intensive applications
  - *`Adding`* one or more threads to share the computation is an alternative to solve the load imbalance problem
    - Technique described by Dr. Danesh Tafti and Weicheng Huang, see www.ncsa.uiuc.edu/News/Access/Stories/LoadBalancing/

116

Case Studies - General Comments: Cache Optimizations

- Loop interchange and array transposition
  - TOMCATV code example

```
REAL rx(jdim, idim)
C$OMP PARLLEL DO
DO i=2, n-1
   do j=2, n
     rx(i,j) = rx(i,j-1)+...
   ENDDO
ENDDO
```

```
REAL rx(idim, jdim)
C$OMP PARLLEL DO
DO i=2, n-1
   do j=2, n
     rx(j,i) = rx(j-1,i)+...
   ENDDO
ENDDO
```

- Outer loop unrolling of a loop nest to block data for the cache
- Array padding

```
Common /map/ A(1024), B(1024)
```

```
Common /map/ A(1024), PAD(8), B(1024)
```

  - Changing the memory address affects cache mapping
  - Increases cache hit rates

117

*third party names are the property of their respective owners.

Case Studies - General Comments: Cache Optimizations

- Loop interchange may increase cache locality through stride-1 references

```
!$OMP PARALLEL DO
DO i=1,N
  DO j=1,N
    a(i,j) = b(i,j) + c(i,j)
  END DO
END DO
```

```
!$OMP PARALLEL DO
DO j=1,N
  DO i=1,N
    a(i,j) = b(i,j) + c(i,j)
  END DO
END DO
```

- In version on right, the cache miss ration is greatly reduced
  - We executed these codes on Cobalt @ NCSA (one node of SGI Altix System with 32-way 1.5 GHz Itanium 2 processors) using the Intel compiler 8.0 with –O0
  - The level 2 cache miss rate is changed from 0.744 to 0.032
  - The bandwidth used for level 2 cache is decreased from 11993.404 MB/s to 4635.817 MB/s
  - Wall clock time is reduced from 77.383s to 8.265s

118

Case Studies - General Comments: Cache Optimizations

- A variety of methods to improve cache performance
  - loop fission
  - loop fusion
  - loop unrolling (outer and inner loops)
  - loop tiling
  - array padding

119

Case Studies - General Comments: Sources of Errors

- Incorrect use of synchronization constructs
  - Less likely if user sticks to directives
  - Erroneous use of locks can lead to deadlock
  - Erroneous use of NOWAIT can lead to race conds.
- Race conditions (true sharing)
  - Can be very hard to find
- Wrong declared data attributes (shared vs. private)
- Wrong "spelling" of sentinel

It can be very hard to track race conditions. Tools may help check for these, but they may fail if your OpenMP code does not rely on directives to distribute work. Moreover, they can be quite slow.

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

121

Seismic Data Processing Example

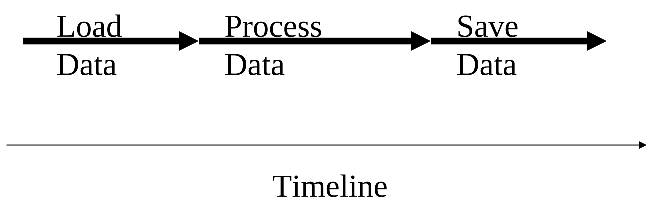
- ***The KINGDOM Suite+* from Seismic Micro-Technology, Inc. Software to find oil and gas.**
- **Integrated package for Geological/Geophysical interpretation and risk reduction.**
  - **Interpretation**
    - 2d/3dPAK
    - EarthPAK
    - VuPAK
  - **Risk Reduction**
    - AVOPAK
    - Rock Solid Attributes
    - SynPAK
    - TracePAK
    - ModPAK

SMT Seismic Micro-Technology, Inc.
KINGDOM Software • (RC)[2] Software • SURE Software

*third party names are the property of their respective owners.

Seismic Data Processing

```
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
{
    Loadline(iLineIndex,...);
   for(j=0;j<iNumTraces;j++)
        for(k=0;k<iNumSamples;k++)
             processing();
    SaveLine(iLineIndex);
}
```



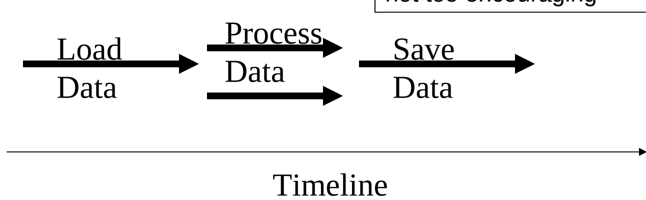
123

First OpenMP Version of Seismic Data Processing

```
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
{
    Loadline(iLineIndex,...);
#pragma omp parallel for
   for(j=0;j<iNumTraces;j++)
        for(k=0;k<iNumSamples;k++)
             processing();
    SaveLine(iLineIndex);
}
```

Overhead for entering and leaving the parallel region

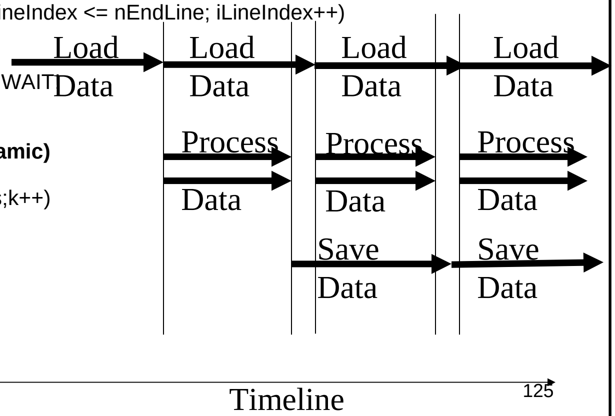
Better performance, but not too encouraging



124

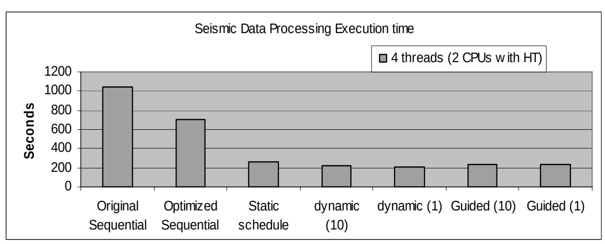
Optimized OpenMP Version of Seismic Data Processing

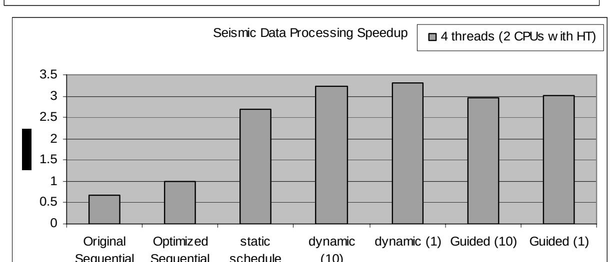
```
 Loadline(nStartLine,...);  // preload the first line of data
#pragma omp parallel
{
   for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
   {
#pragma omp single nowait
  {// loading the next line data, NO WAIT
        Loadline(iLineIndex+1,...);
   }
#pragma omp for schedule(dynamic)
      for(j=0;j<iNumTraces;j++)
              for(k=0;k<iNumSamples;k++)
                      processing();
#pragma omp single nowait
      {
         SaveLine(iLineIndex);
      }
   }
}
```



125

Performance of Optimized Seismic Data Processing Application





2 3.4GHz Intel® Xeon™ processors with Hyper-threading and Intel® Extended Memory 64 Technologies

3G RAM

126

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

127

Case Studies: a Jacobi Example

- Solving the Helmholtz equation with a finite difference method on a regular 2D-mesh
- Using an iterative Jacobi method with over-relaxation
  - Well suited for the study of various approaches of loop-level parallelization
- Taken from
  - Openmp.org
    - The original OpenMP program contains 2 parallel regions inside the iteration loop
  - Dieter an Mey, Thomas Haarmann, Wolfgang Koschel. **"Pushing Loop-Level Parallelization to the Limit "**, *EWOMP '02*.
    - This paper shows how to tune the performance for this program step by step

128

A Jacobi Example: Version 1

Two parallel regions inside the iteration loop

```
error = 10.0 * tol
k = 1
do while (k.le.maxit .and. error.gt. tol) ! begin iteration loop
error = 0.0
!$omp parallel do
do j=1,m
 do i=1,n
   uold(i,j) = u(i,j)
enddo
enddo
!$omp end parallel do
!$omp parallel do private(resid) reduction(+:error)
   do j = 2,m-1 do i = 2,n-1
     resid = (ax*(uold(i-1,j) + uold(i+1,j))
&      + ay*(uold(i,j-1) + uold(i,j+1))    + b * uold(i,j) - f(i,j))/b
       u(i,j) = uold(i,j) - omega * resid
       error = error + resid*resid
     end do
   enddo
!$omp end parallel do
   k = k + 1
   error = sqrt(error)/dble(n*m)
enddo ! end iteration loop
```

129

A Jacobi Example: Version 2

Merging two parallel regions inside the iteration loop.

An nowait is added.

```
error = 10.0 * tol
k = 1
do while (k.le.maxit .and. error.gt. tol) ! begin iteration loop
 error = 0.0
!$omp parallel private(resid)
!$omp do
 do j=1,m
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp end do
!$omp do reduction(+:error)
 do j = 2,m-1
  do i = 2,n-1
   resid = (ax*(uold(i-1,j) + uold(i+1,j))
& + ay*(uold(i,j-1) + uold(i,j+1))
& + b * uold(i,j) - f(i,j))/b
   u(i,j) = uold(i,j) - omega * resid
   error = error + resid*resid
  end do
 enddo
!$omp end do nowait
!$omp end parallel
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo ! end iteration loop
```

130

A Jacobi Example: Version 3

One parallel region containing the whole iteration loop.

```
error = 10.0d0 * tol
!$omp parallel private(resid, k_priv)
k_priv = 1
do while (k_priv.le.maxit .and. error.gt.tol) ! begin iteration loop
!$omp do
 do j=1,m
   do i=1,n
    uold(i,j) = u(i,j)
   enddo
  enddo
!$omp end do
!$omp single
  error = 0.0d0
!$omp end single
!$omp do reduction(+:error)
  do j = 2,m-1
      ……
      error = error + resid*resid
   enddo
!$omp end do
  k_priv = k_priv + 1
!$omp single
  error = sqrt(error)/dble(n*m)
!$omp end single
enddo ! end iteration loop
!$omp single
k = k_priv
!$omp end single nowait
!$omp end parallel
```

131

A Jacobi Example: Version 4

By replacing the shared variable **`error`** by a private copy **`error_priv`** in the termination condition of the iteration loop, one of the four barriers can be eliminated. An "end single" with an implicit barrier was here in Version 3.

```
!$omp parallel private(resid, k_priv,error_priv)
k_priv = 1 error_priv = 10.0d0 * tol
do while (k_priv.le.maxit .and. error_priv.gt.tol) ! begin iter. loop
!$omp do
  do j=1,m
   do i=1,n
     uold(i,j) = u(i,j)
   enddo
   enddo
!$omp end do
!$omp single
    error = 0.0d0
!$omp end single
!$omp do reduction(+:error)
   do j = 2,m-1
    do i = 2,n-1
       ……
       error = error + resid*resid
    end do
   enddo
!$omp end do
   k_priv = k_priv + 1
  error_priv = sqrt(error)/dble(n*m)
enddo ! end iteration loop
!$omp barrier
!$omp single
  k = k_priv
  error = error_priv
!$omp end single
!$omp end parallel
```

132

Performance Tuning of the Jacobi example

- V1: the original OpenMP program with two parallel regions inside the iteration loop
- V2: merges two parallel regions into one region
- V3: moves the parallel region out to include the iteration loop inside
- V4: replaces a shared variable by a private variable to perform the reduction so that one out of four barriers can be eliminated
- V5: the worksharing constructs are eliminated in order to reduce the outlining overhead by the compiler

133

An Jacobi Example: Version 5

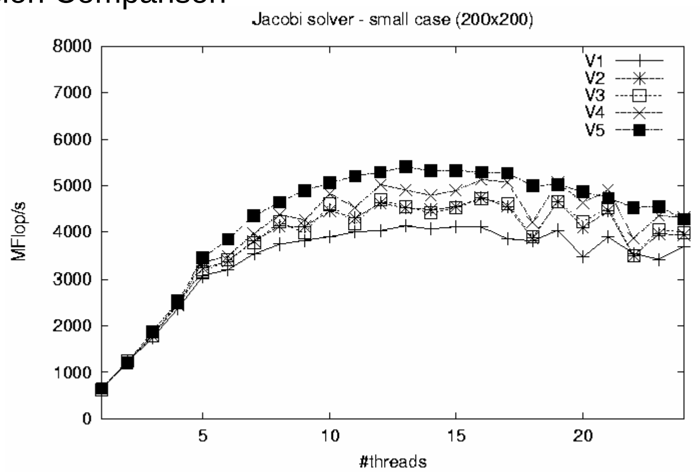
The worksharing constructs replaced to avoid the outlining overhead by the compiler.

The reduction is replaced by an atomic directive.

```
do j=1,m, m-1
  do i=1,n
    uold(i,j) = u(i,j)
  enddo
enddo
do j=2,m-1
  do i=1,n,n-1
    uold(i,j) = u(i,j)
  enddo
enddo
! all parallel loops run from 2 to m-1
nthreads = omp_get_max_threads()
ilo = 2;  ihi = m-1
nrem = mod ( ihi - ilo + 1, nthreads )
nchunk = ( ihi - ilo + 1 - nrem ) / nthreads
!$omp parallel private(me,is,ie,resid,
k_priv,error_priv)
me = omp_get_thread_num()
if ( me < nrem ) then
  is = ilo + me * ( nchunk + 1 );  ie = is + nchunk
else
  is = ilo + me * nchunk + nrem;  ie = is + nchunk - 1
end if
error_priv = 10.0 * tol;   k_priv = 1
do while (k_priv.le.maxit .and.
error_priv.gt.tolh) ! begin iter. loop
   do j=is,ie
     do i=2,n-1
       uold(i,j) = u(i,j)
     enddo
   enddo
!$omp barrier
!$omp single
   error = 0
!$omp end single
   error_priv = 0
   do j = is,ie
     do i = 2,n-1
       ……
       error_priv = error_priv + resid*resid
     end do
   enddo
!$omp atomic
   error = error + error_priv
   v = k_priv + 1
   k_pri
!$omp barrier
   error_priv = sqrt ( error ) / dble(n*m)
enddo ! end iteration loop
 !$omp single
k = k_priv
!$omp end single
!$omp end parallel
error = sqrt ( error ) / dble(n*m)
```

134

An Jacobi Example:
Version Comparison



Comparison of 5 different versions of the Jacobi solver on a Sun Fire 6800, grid size 200x200 , 1000 iterations

135

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

136

SpecOMP: Gafort

- Global optimization using a genetic algorithm.
  - Written in Fortran
  - 1500 lines of code
- Most "interesting" loop: shuffle the population.
  - Original loop is not parallel; performs pair-wise swap of an array element with another, randomly selected element. There are 40,000 elements.

137

Shuffle populations

```
DO j=1,npopsiz-1

         CALL ran3(1,rand,my_cpu_id,0)
         iother=j+1+DINT(DBLE(npopsiz-j)*rand)
        itemp(1:nchrome)=iparent(1:nchrome,iother)
        iparent(1:nchrome,iother)=iparent(1:nchrome,j)
        iparent(1:nchrome,j)=itemp(1:nchrome)
        temp=fitness(iother)
        fitness(iother)=fitness(j)
        fitness(j)=temp

END DO
```

138

SpecOMP: Gafort

- Parallelization idea:
  - Perform the swaps in parallel.
  - Must protect swap to prevent races.
  - High level synchronization (critical) would prevent all opportunity for speedup.
- Solution:
  - use one lock per array element → 40,000 locks.

139

Gafort parallel shuffle

Exclusive access to array elements. Ordered locking prevents deadlock.

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp, my_cpu_id)
      my_cpu_id = 1
!$    my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
      DO j=1,npopsiz-1
        CALL ran3(1,rand,my_cpu_id,0)
        iother=j+1+DINT(DBLE(npopsiz-j)*rand)
!$      IF (j < iother) THEN
!$        CALL omp_set_lock(lck(j))
!$        CALL omp_set_lock(lck(iother))
!$      ELSE
!$        CALL omp_set_lock(lck(iother))
!$        CALL omp_set_lock(lck(j))
!$      END IF
        itemp(1:nchrome)=iparent(1:nchrome,iother)
        iparent(1:nchrome,iother)=iparent(1:nchrome,j)
        iparent(1:nchrome,j)=itemp(1:nchrome)
        temp=fitness(iother)
        fitness(iother)=fitness(j)
        fitness(j)=temp
!$      IF (j < iother) THEN
!$        CALL omp_unset_lock(lck(iother))
!$        CALL omp_unset_lock(lck(j))
!$      ELSE
!$        CALL omp_unset_lock(lck(j))
!$        CALL omp_unset_lock(lck(iother))
!$      END IF
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

140

Gafort Results

- 99.9% of the code was inside parallel section.
- Speedup was OK (6.5 on 8 processors) but not great.
- This application led us to make a change to OpenMP:
  - OpenMP 1.0 required that locks be initialized in a serial region.
  - With 40,000 of them, this just wasn't practical.
  - So in OpenMP 1.1 we changed locks so they could be initialized in parallel.

141

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

142

SPEComp: Equake

- Earthquake modeling
  - simulates the propagation of elastic waves in large, highly heterogeneous valleys
  - Input: grid topology with nodes, coordinates, seismic event characteristics, etc
  - Output: reports the motion of the earthquake for a certain number of simulation timesteps
- Most time consuming loop: smvp(..)
  - sparse matrix calculation: accumulate the results of matrix-vector product

143

Smvp()

- Frequent and scattered accesses to shared arrays w1 and w2
- Naive OpenMP: uses critical to synchronize each access
  - serializes most of the execution
  - Not scalable

```
for (i = 0; i < nodes; i++) {
   Anext = Aindex[i];     Alast = Aindex[i + 1];
   sum0 = A[Anext][0][0]*v[i][0] .. + A[Anext][0][2]*v[i][2];
    ...
   Anext++;
   while (Anext < Alast) {
     col = Acol[Anext];

     sum0 += A[Anext][0][0]*v[col][0] ..+ A[Anext][0][2]*v[col][2];
     ....
     if (w2[col] == 0) {
             w2[col] = 1;
             w1[col].first = 0.0;
              ...   }
     w1[col].first += A[Anext][0][0]*v[i][0] .. + A[Anext][2][0]*v[i][2];
    ……
     Anext++;
   }

   if (w2[i] == 0) {
     w2[i] = 1;
     w1[i].first = 0.0;
     ...
   }
   w1[i].first += sum0;
  ……
}
```

144

Solution

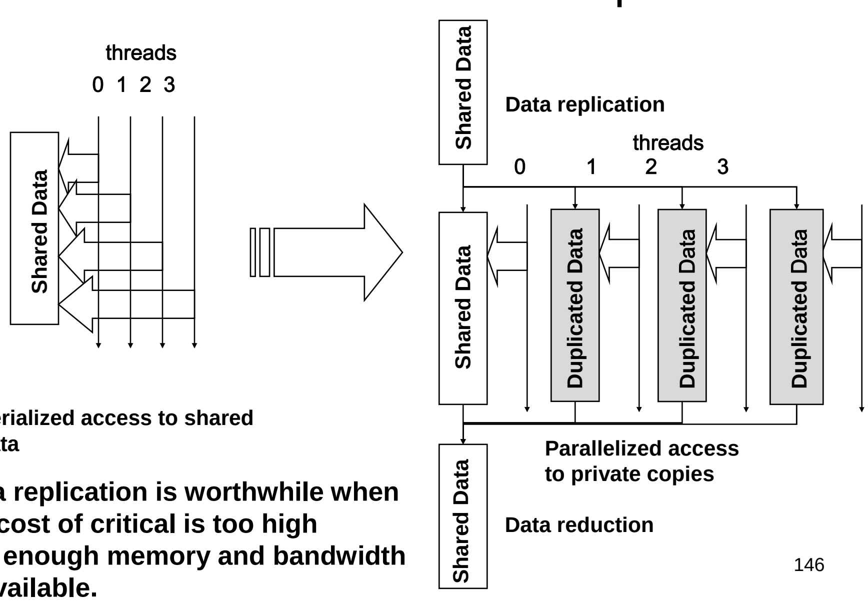
```
#pragma omp parallel private(my_cpu_id,i,..col,sum0,sum1,sum2)
{  #ifdef _OPENMP
     my_cpu_id = omp_get_thread_num();
     numthreads=omp-get_num_threads();
  #else
     my_cpu_id=0; numthreads=1;
  #endif
  #pragma omp for
   for (i = 0; i < nodes; i++) {
   …..
   sum0 = A[Anext][0][0]*v[i][0] ... +A[Anext][0][2]*v[i][2];
   ….
   while (Anext < Alast) {
   sum0 += A[Anext][0][0]*v[col][0] ...+ A[Anext][0][2]*v[col][2];
     ....
   if (w2[my_cpu_id][col] == 0) {
           w2[my_cpu_id][col] = 1;
           w1[my_cpu_id][col].first = 0.0;
      ......      }
    w1[my_cpu_id][col].first += A[Anext][0][0]*v[i][0] ...+ A[Anext][2][0]*v[i][2]
     ....  }
    if (w2[my_cpu_id][i] == 0) {
      w2[my_cpu_id][i] = 1;
      w1[my_cpu_id][i].first = 0.0;        ....    }
     w1[my_cpu_id][i].first += sum0;
      ...  }   }
#pragma omp parallel for private(j) // manual reduction
  for (i = 0; i < nodes; i++) {
   for (j = 0; j < numthreads; j++) {
     if (w2[j][i]) { w[i][0] += w1[j][i].first;
.... }   }  }
```

SPMD-style

- Replicate w1,w2 for each thread.
  - exclusive access to arrays
  - no synchronization
- Downside:
  - large memory consumption
  - extra time to duplicate data and reduce copies back to one
- Performance result:
  - good scaling up to 64 CPUs for medium dataset

145

Lessons Learned from Equake



Serialized access to shared data

Data replication is worthwhile when the cost of critical is too high and enough memory and bandwidth is available.

146

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

147

Getting more concurrency to work with: Art (SpecOMP 2001)

- Art: Adaptive Resonance Theory) is an Image processing application based on neural networks and used to recognize objects in a thermal image.
- Source has over 1300 lines of C code.

148

Key Loop in Art

Problem:

which loop to parallelize?

Inum and jnum aren’t that big – insufficient parallelism to support scalability and compensate for parallel overhead.

```
for (i=0; i<inum;i++) {
  for (j=0; j<jnum; j++) {

      k=0;
      for (m=j;m<(gLheight+j);m++)
        for (n=i;n<(gLwidth+i);n++)
          f1_layer[o][k++].I[0] = cimage[m][n];

      gPassFlag =0;
      gPassFlag = match(o,i,j, &mat_con[ij], busp);

      if (gPassFlag==1) {
        if (set_high[o][0]==TRUE) {
          highx[o][0] = i;
          highy[o][0] = j;
          set_high[o][0] = FALSE;
        }
       if (set_high[o][1]==TRUE)  {
         highx[o][1] = i;
         highy[o][1] = j;
         set_high[o][1] = FALSE;
       }
      }
    }
}
```

149

Coalesce Multiple Parallel Loops

- Original Code

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(K,J,I) = Func(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

- Loop Coalescing

```
ITRIP = N ; JTRIP = M
!$OMP PARALLEL DO
DO IJ = 0, ITRIP*JTRIP-1
  I = 1 +      IJ/JTRIP
  J = 1 + MOD(IJ,JTRIP)
  DO K = 1, L
    A(K,J,I) = Func(I,J,K)
  ENDDO
ENDDO
```

…

- >1 loop is parallel
- None of N, M, L very large
- What is best parallelization strategy?

150

Indexing to support Loop Coalescing

Key loop in Art

```
#pragma omp for private (k,m,n, gPassFlag) schedule(dynamic)
  for (ij = 0; ij < ijmx; ij++)  {
    j = ((ij/inum) * gStride) + gStartY;
    i = ((ij%inum) * gStride) +gStartX;
    k=0;
    for (m=j;m<(gLheight+j);m++)
      for (n=i;n<(gLwidth+i);n++)
        f1_layer[o][k++].I[0] = cimage[m][n];

    gPassFlag =0;
    gPassFlag = match(o,i,j, &mat_con[ij], busp);

    if (gPassFlag==1) {
      if (set_high[o][0]==TRUE) {
        highx[o][0] = i;
        highy[o][0] = j;
        set_high[o][0] = FALSE;
      }
     if (set_high[o][1]==TRUE)  {
       highx[o][1] = i;
       highy[o][1] = j;
       set_high[o][1] = FALSE;
     }
    }
  }
```

Note:

Dynamic Schedule needed because of embedded conditionals

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

152

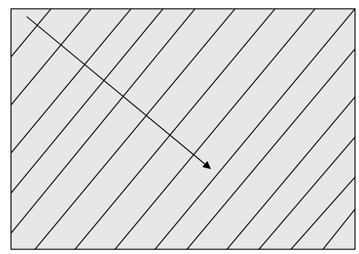
SOR algorithms

- Original Code

```
DO I = 2, N
  DO J = 2, M
    DO K = 2, L
      A(K,J,I) =
        Func( A(K-1,J,
I),
              A(K,  J-
1,I),
              A(K,  J,
I-1))
    ENDDO
  ENDDO
ENDDO
```

- No loop is parallel
- A wavefront dependence pattern
- What is best parallelization strategy?

- What is a wavefront?
  - Each point on the wavefront can be computed in parallel
  - Each wavefront must be completed before next one
  - A two-dimensional example



153

Wave-fronting SOR

- Manual Wavefront – combine loops and restructure to compute over wavefronts

```
!$OMP PARALLEL PRIVATE(J)
DO IJSUM= 4, N+M
  !$OMP DO SCHEDULE(STATIC,1)
  DO I = max(2,IJSUM-M), min(N,IJSUM-2)
    J = IJSUM - I
    DO K = 2, L
      A(K,J,I) = Func( A(K-1,J,I), A(K,J-1,I), A(K,J,I-1) )
    ENDDO
  ENDDO
ENDDO
!$OMP END PARALLEL
```

- Notice only I and J loops wave-fronted
  - Relatively easy to wavefront all three loops, but stride-1 inner loop helps cache access

154

Case Studies and Examples

- More difficult problems to parallelize
  - Seismic Data Processing: Dynamic Workload
  - Jacobi: Stepwise improvement
  - Fine Grained locks
  - Equake: Time vs. Space
  - Coalescing loops
  - Wavefront Parallelism
  - Work Queues

155

OpenMP Enhancements : Work queues

OpenMP doesn't handle pointer following loops very well

```
nodeptr list, p;
For (p=list; p!=NULL; p=p->next)
        process(p->data);
```

Intel has proposed (and implemented) a taskq construct to deal with this case:

```
nodeptr list, p;
#pragma omp parallel taskq
For (p=list; p!=NULL; p=p->next)
#pragma omp task
        process(p->data);
```

156

Reference: Shah, Haab, Petersen and Throop, EWOMP'1999 paper.

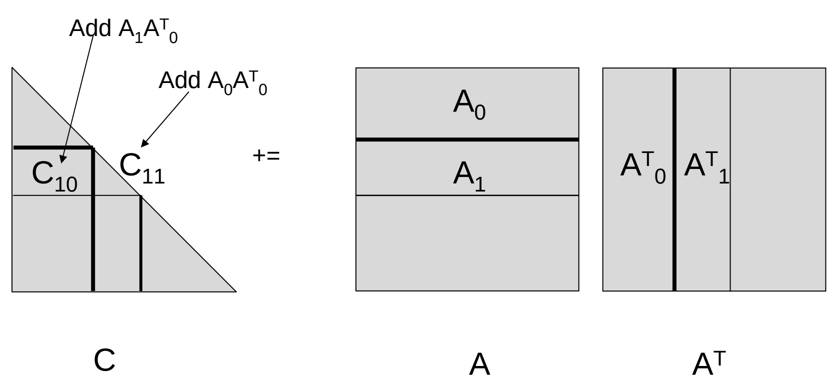
Task queue example - FLAME: Shared Memory Parallelism in Dense Linear Algebra

- Traditional approach to parallel linear algebra:
  - Only parallelism from multithreaded BLAS
- Observation:
  - Better speedup if parallelism is exposed at a higher level
- FLAME approach:
  - OpenMP task queues

Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. "Parallelizing FLAME Code with OpenMP Task Queues." *TOMS* , submitted.

157

Symmetric rank-k update



Note: the iteration sweeps through C and A, creating a new block of rows to be updated with new parts of A. These updates are completely independent.

Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. "Parallelizing FLAME Code with OpenMP Task Queues." *TOMS* , submitted.

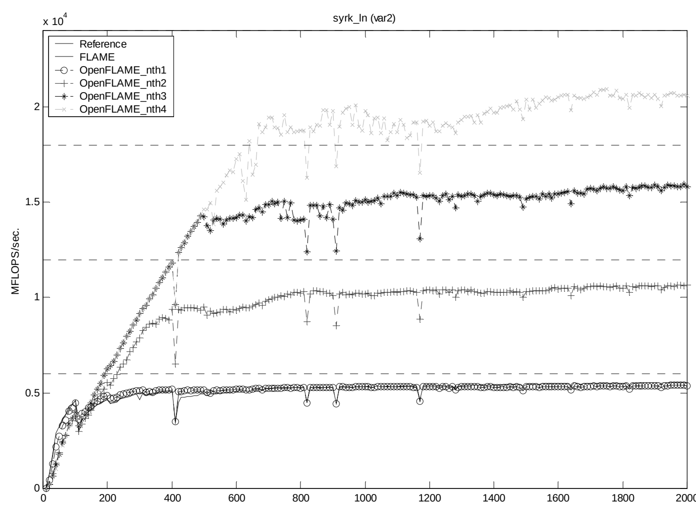
158

```
#pragma intel omp parallel taskq
{
  while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,   &C00, /**/ &C01, &C02,
                          /************/  /********************/
                                           &C10, /**/ &C11, &C12,
                           CBL, /**/ CBR,   &C20, /**/ &C21, &C22,
                           b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,              &A0,
                        /* ** */          /* ** */
                                            &A1,
                           AB,              &A2,   b, FLA_BOTTOM );
    /*------------------------------------------------------------*/
    #pragma intel omp task captureprivate( A0, A1, C10, C11 )
    {
      FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
      FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
    } /* end task */
    /*------------------------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                               C10, C11, /**/ C12,
                              /*************/ /******************/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                                  A1,
                              /* ** */           /* ** */
                              &AB,                A2,     FLA_TOP );
  }
}   /* end of taskq */
```

159

Top line represents peak of
Machine (Itanium2 1.5GHz, 4CPU)



Note: the above graphs is for the most naïve way of marching through the matrices.
By picking blocks dynamically, much faster ramp-up can be achieved.

160

Agenda

- Parallel computing, threads, and OpenMP
- The core elements of OpenMP
  - Thread creation
  - Workshare constructs
  - Managing the data environment
  - Synchronization
  - The runtime library and environment variables
  - Recapitulation
- The OpenMP compiler
- OpenMP usage: common design patterns
- Case studies and examples
- Background information and extra details

161

Reference Material on OpenMP

- OpenMP architecture review board URL, the primary source of information about OpenMP:
  www.openmp.org
- OpenMP User's Group (cOMPunity) URL:
  www.compunity.org
- Books:
  Parallel programming in OpenMP, Chandra, Rohit, San. : Francisco, Calif Morgan Kaufmann ; London : Harcourt, 2000, ISBN: 1558606718

  Using OpenMP; Chapman, Jost, Van der Pas, Mueller; MIT Press (to appear, 2006)

  Patterns for Parallel Programming, Mattson, Sanders, Massingill, Addison Wesley, 2004

162

OpenMP Papers

- Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.
- Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.
- Bentz J., Kendall R., "Parallelization of General Matrix Multiply Routines Using OpenMP", Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 1, 2005
- Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.
- Chapman B, Mehrotra P. OpenMP and HPF: integrating two paradigms. [Conference Paper] Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Proceedings. Springer-Verlag. 1998, pp.650-8. Berlin, Germany
- Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.
- Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp.566-71. Cary, NC, USA.

163

OpenMP Papers (continued)

- Jost G., Labarta J., Gimenez J., What Multilevel Parallel Programs do when you are not watching: a Performance analysis case study comparing MPI/OpenMP, MLP, and Nested OpenMP, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 29, 2005
- Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40.
- B. Chapman, O. Hernandez, L. Huang, T.-H. Weng, Z. Liu, L. Adhianto, Y. Wen, "Dragon: An Open64-Based Interactive Program Analysis Tool for Large Applications," *Proc. 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 03)*. 792-796. 2003.
- Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.
- Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini. Parallel Programming with Message Passing and Directives; SIAM News, Volume 32, No 9, Nov. 1999.
- Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Lecture Notes in Computer Science Vol.1662. Springer-Verlag. 1999, pp.339-50.
- Liu Z., Huang L., Chapman B., Weng T., Efficient Implementationi of OpenMP for Clusters with Implicit Data Distribution, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 121, 2005

164

OpenMP Papers (continued)

- **B. Chapman, F. Bregier, A. Patil, A. Prabhakar, “Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems,” *Concurrency and Computation: Practice and Experience.* 14(8-9): 713-739, 2002.**
- **J. M. Bull and M. E. Kambites. JOMP: an OpenMP-like interface for Java. Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.**
- **Mattson, T.G. An Introduction to OpenMP 2.0, Proceedings 3rd International Symposium on High Performance Computing, Lecture Notes in Computer Science, Number 1940, Springer, 2000 pp. 384-390, Tokyo Japan.**
- **Magro W, Petesen P, Shah S. Hyper-Threading Technology: Impact on Computer-Intensive Workloads. Intel Technology Journal, Volume 06, Issue 01, 2002. ISSN 1535-766X**
- **Mattson, T.G., How Good is OpenMP? Scientific Programming, Vol. 11, Number 2, p.81-93, 2003.**
- **Duran A., Silvera R., Corbalan J., Labarta J., “Runtime Adjustment of Parallel Nested Loops”, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 137, 2005**
- **Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP; Concurrency: Practice and Experience, 2000; 12:1219-1239. Publisher John Wiley & Sons, Ltd.**

165

OpenMP Papers (continued)

- **Voss M., Chiu E., Man P., Chow Y. Wong C., Yuen K., “An evaluation of Auto-Scoping in OpenMP”, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 98, 2005**
- **T.-H. Weng, B. M. Chapman, “Implementing OpenMP using Dataflow execution Model for Data Locality and Efficient Parallel Execution,” In Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments, (HIPS-7), IEEE, April 2002,**
- **T-H. Weng and B. Chapman, “Toward Optimization of OpenMP Codes for Synchronization and Data Reuse,” *Int. Journal of High Performance Computing and Networking (IJHPCN),* Vol. 1, 2004.**
- **Z. Liu, B. Chapman, T.-H. Weng, O. Hernandez. “Improving the Performance of OpenMP by Array Privatization, ” Workshop on OpenMP Applications and Tools, WOMPAT’2002. LNCS 2716, Spring Verlag, pp. 244-259, 2002.**
- **Hu YC, Honghui Lu, Cox AL, Zwaenepoel W. OpenMP for networks of SMPs. Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999. IEEE Comput. Soc. 1999, pp.302-10. Los Alamitos, CA, USA.**
- **Scherer A, Honghui Lu, Gross T, Zwaenepoel W. Transparent adaptive parallelism on NOWS using OpenMP. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.34, no.8, Aug. 1999, pp.96-106. USA.**
- **L. Huang, B. Chapman and Z. Liu, “Towards a More Efficient Implementation of OpenMP for Clusters via Translation to Global Arrays,” *Parallel Computing*. To appear, 2005.**
- **M. Bull, B. Chapman (Guest Editors), *Special Issues on OpenMP. Scientific Programming 9, 2 and 3*, 2001.**

166

Backup material

- History
- Foundations of OpenMP
- Memory Models and the Infamous Flush
- Performance optimization in OpenMP
- The future of OpenMP

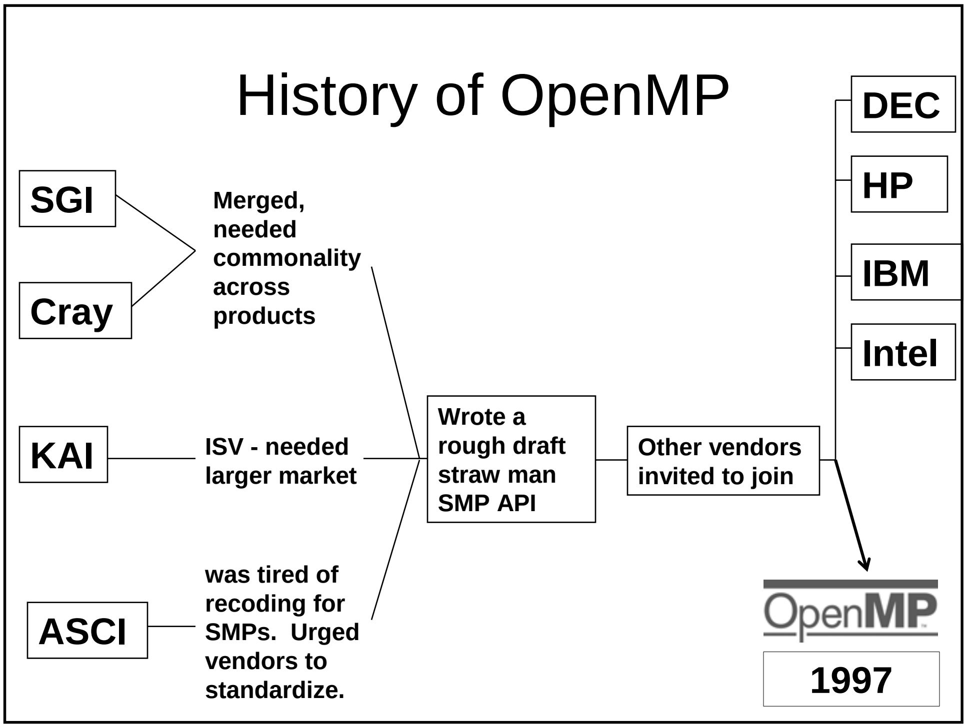
167

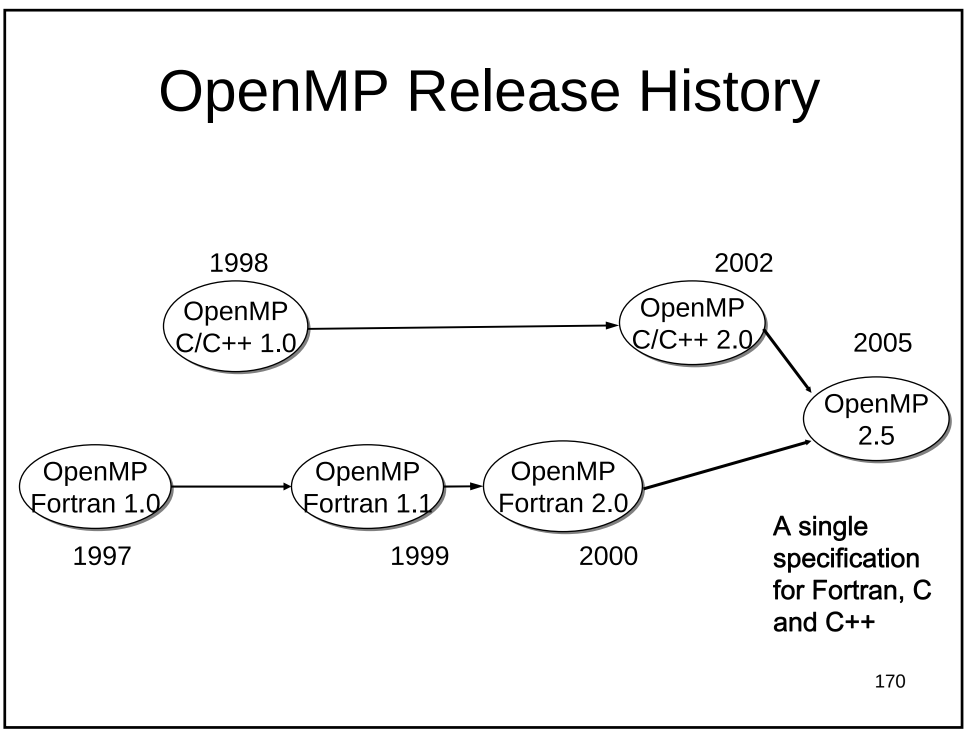
OpenMP pre-history

- OpenMP based upon SMP directive standardization efforts PCF and aborted ANSI X3H5 – late 80's
- Nobody fully implemented either
- Only a couple of partial solutions
- Vendors considered proprietary API's to be a competitive feature:
  - Every vendor had proprietary directives sets
  - Even KAP, a "portable" multi-platform parallelization tool used different directives on each platform

168

PCF – Parallel computing forum **KAP – parallelization tool from KAI.**


History of OpenMP
SGI
Cray
Merged, needed commonality across products
KAI
ISV - needed larger market
ASCI
was tired of recoding for SMPs. Urged vendors to standardize.
Wrote a rough draft straw man SMP API
Other vendors invited to join
DEC
HP
IBM
Intel
OpenMP
1997

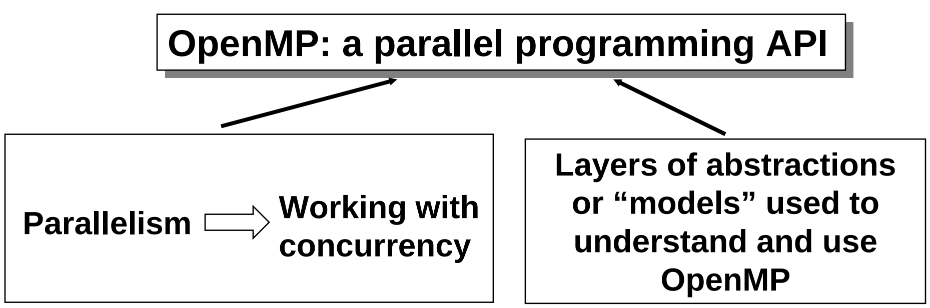

OpenMP Release History
1998
OpenMP C/C++ 1.0
2002
OpenMP C/C++ 2.0
2005
OpenMP 2.5
OpenMP Fortran 1.0
1997
OpenMP Fortran 1.1
1999
OpenMP Fortran 2.0
2000
A single specification for Fortran, C and C++
170

Backup material

- History
- Foundations of OpenMP
- Memory Models and the Infamous Flush
- Performance optimization in OpenMP
- The future of OpenMP

171

The Foundations of OpenMP:



172

Concurrency:

- Concurrency:
  - When multiple tasks are active simultaneously.
- "Parallel computing" occurs when you use concurrency to:
  - Solve bigger problems
  - Solve a fixed size problem in less time
- For parallel computing, this means you need to:
  - Identify exploitable concurrency.
  - Restructure code to expose the concurrency.
  - Use a parallel programming API to express the concurrency within your source code.

173

The Foundations of OpenMP:

OpenMP: a parallel programming API

Parallelism ⇨ Working with concurrency

Layers of abstractions or "models" used to understand and use OpenMP

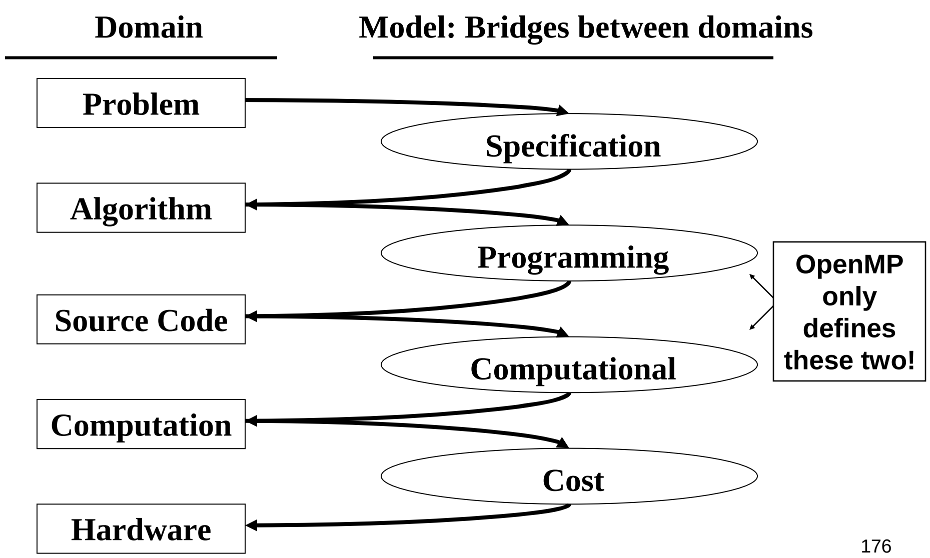
174

Reasoning about programming

- Programming is a process of successive refinement of a solution relative to a hierarchy of models.
- The models represent the problem at a different level of abstraction.
  – The top levels express the problem in the original problem domain.
  – The lower levels represent the problem in the computer's domain.
- The models are informal, but detailed enough to support simulation.

Source: J.-M. Hoc, T.R.G. Green, R. Samurcay and D.J. Gilmore (eds.), *Psychology of Programming*, Academic Press Ltd., 1990

Layers of abstraction in programming

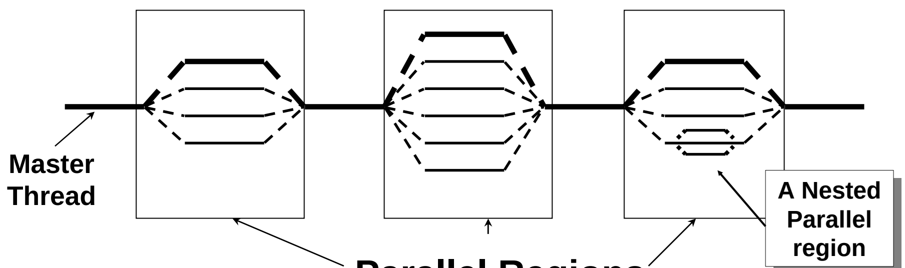


176

OpenMP Programming Model:

Fork-Join Parallelism:

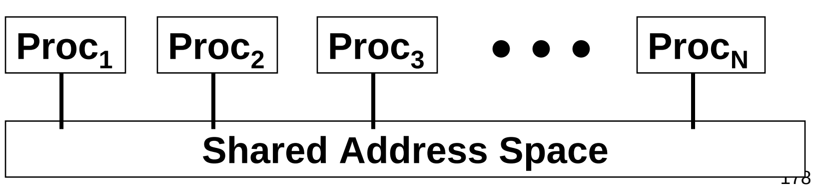
- ◆Master thread spawns a team of threads as needed.
- ◆Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.



177

OpenMP Computational model

- OpenMP was created with a particular abstract machine or ***computational model*** in mind:
  - Multiple processing elements.
  - A shared address space with "equal-time" access for each processor.
  - Multiple light weight processes (threads) managed outside of OpenMP (the OS or some other "third party").



178

What about the other models?

- Cost Models:
  - OpenMP doesn't say anything about the cost model – programmers are left to their own devices.
- Specification Models:
  - Some parallel algorithms are natural in OpenMP:
    - loop-splitting.
    - SPMD (single program multiple data).
  - Other specification models are hard for OpenMP
    - Recursive problems and list processing is challenging for OpenMP's models.

179

Backup material

- History
- Foundations of OpenMP
- ⇨ Memory Models and the Infamous Flush
- Performance optimization in OpenMP
- The future of OpenMP

180

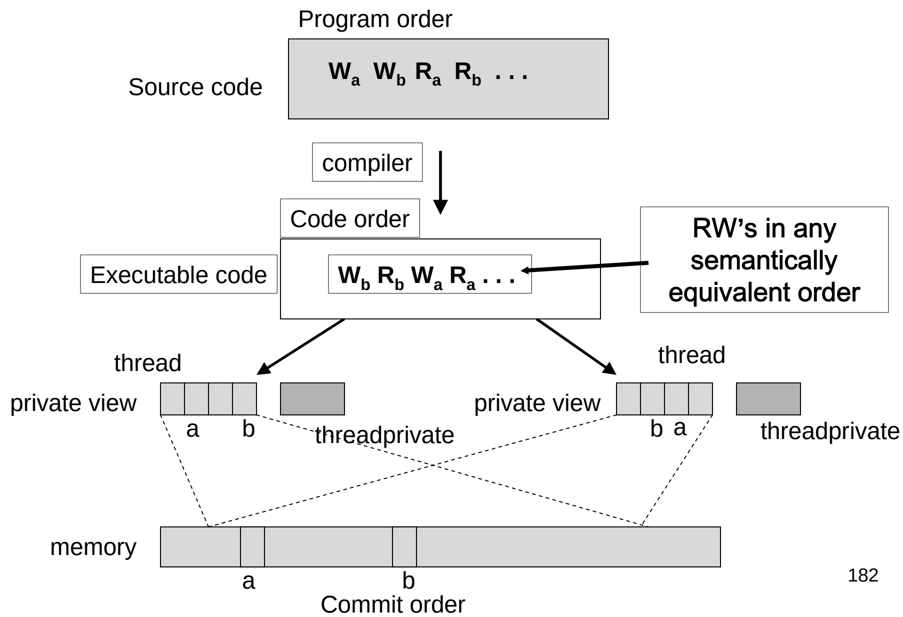
OpenMP and Shared memory

- What does OpenMP assume concerning the shared memory?
    - Implies that all threads access memory at the same cost, but the details were never spelled out (prior to OMP 2.5).
- Shared memory is understood in terms of:
    - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
    - Consistency: Orderings of accesses to different addresses by multiple threads.
- OpenMP was never explicit about its memory model. This was fixed in OpenMP 2.5.
    - If you want to understand how threads interact in OpenMP, you need to understand the memory model.

“The OpenMP Memory Model”, Jay Hoeflinger and Bronis de Supinski, IWOMP’2005

181

OpenMP Memory Model: Basic Terms



182

Coherence rules in OpenMP 2.5

```
#pragma omp parallel private(x) shared(p0, p1)

Thread 0                Thread 1
X = …;                  X = …;
P0 = &x;                P1 = &x;

/*   references in the following line are not allowed   */
… *p1 …                 … *p0 …
```

You can not reference another's threads private variables … even if you have a shared pointer between the two threads.

183

Coherence rules in OpenMP 2.5

```
#pragma omp parallel private(x) shared(p0, p1)

Thread 0                Thread 1
X = …;                  X = …;
P0 = &x;                P1 = &x;

/*   references in the following line are not allowed   */
… *p1 …                 … *p0 …

#pragma omp parallel shared(x)

Thread 0   Thread 1          Thread 0   Thread 1
… X …      … X …             X = …;     X = …;
…*p0 …     … *p0 …           … *p1 …    … *p1 …

/*   the following are not allowed   */
…*p1 …     … *p1 …           … *p0 …    … *p0 …
```

Nested parallel regions must keep track of the privates pointers reference.

184

Consistency: Memory Access Re-ordering

- Re-ordering:
  - Compiler re-orders program order to the code order
  - Machine re-orders code order to the memory commit order
- At a given point in time, the temporary view of memory may vary from shared memory.
- Consistency models based on orderings or Reads (R), Writes (W) and Synchronizations (S):
  - R→R, W→W, R→W, R→S, S→S, W→S

185

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some or the ordering constrains for memory ops (R, W, S).

186

OpenMP 2.5 and Relaxed Consistency

- OpenMP 2.5 defines consistency as a variant of weak consistency:
  - S ops must be in sequential order across threads.
  - Can not reorder S ops with R or W ops on the same thread
    - Weak consistency guarantees

      S→W, S→R , R→S, W→S, S→S
- The Synchronization operation relevant to this discussion is flush.

187

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set":
  - The flush set is "all thread visible variables" for a flush without an argument list.
  - The flush set is the list of variables when the list is used.
    - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
    - All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush.
    - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

188

What is the Big Deal with Flush?

- Compilers routinely reorder instructions that implement a program
  - Helps exploit the functional units, keep machine busy
- Compiler generally cannot move instructions past a barrier
  - Also not past a flush on all variables
- But it can move them past a flush on a set of variables so long as those variables are not accessed
- So need to use flush carefully

Also, the flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory.

Why is it so important to understand the memory model?

- Question: According to OpenMP 2.0, is the following a correct program:

Thread 1
```
omp_set_lock(lockvar);
#pragma omp flush(count)
Count++;
#pragma omp flush (count)
Omp_unset_lock(lockvar)
```

Thread 2
```
omp_set_lock(lockvar);
#pragma omp flush(count)
Count++;
#pragma omp flush (count)
Omp_unset_lock(lockvar)
```

Not correct prior to OpenMP 2.5:

The Compiler can reorder flush of the lock variable and the flush of count

190

Solution:

- In OpenMP 2.0, you must make the flush set include variables sensitive to the desired ordering constraints.

Thread 1

```
omp_set_lock(lockvar);
#pragma omp flush(count,lockvar)
Count++;
#pragma omp flush(count,lockvar)
Omp_unset_lock(lockvar)
```

Thread 2

```
omp_set_lock(lockvar);
#pragma omp flush(count,lockvar)
Count++;
#pragma omp flush(count,lockvar)
Omp_unset_lock(lockvar)
```

191

Even the Experts are confused

- The following code fragment is from the SpecOMP benchmark ammp. Is it correct?

```
#ifdef _OPENMP
      omp_set_lock (&(a1->lock));
#endif
      a1fx = a1->fx;
      a1fy = a1->fy;
      a1fz = a1->fz;
      a1->fx = 0;
      a1->fy = 0;
      a1->fz = 0;
      xt = a1->dx*lambda + a1->x – a1->px;
      yt = a1->dy*lambda + a1->y – a1->py;
      zt = a1->dz*lambda + a1->z – a1->pz;
#ifdef _OPENMP
      omp_unset_lock(&(a1->lock));
#endif
```

In OpenMP 2.0, the locks don't imply a flush so this code is broken.

192

OpenMP 2.5: lets help out the "experts"

- The following code fragment is from the SpecOMP benchmark ammp. It is correct with OpenMP 2.5.

```
#ifdef _OPENMP
      omp_set_lock (&(a1->lock));
#endif
      a1fx = a1->fx;
      a1fy = a1->fy;
      a1fz = a1->fz;
      a1->fx = 0;
      a1->fy = 0;
      a1->fz = 0;
      xt = a1->dx*lambda + a1->x – a1->px;
      yt = a1->dy*lambda + a1->y – a1->py;
      zt = a1->dz*lambda + a1->z – a1->pz;
#ifdef _OPENMP
      omp_unset_lock(&(a1->lock));
#endif
```

To prevent problems like this, OpenMP 2.5 defines the locks to include a full flush.

That makes this program correct.

193

OpenMP 2.5 Memory Model: summary part 1

- For a properly synchronized program (without data races), the memory accesses of one thread appear to be sequentially consistent to each other thread.
- The only way for one thread to see that a variable was written by another thread is to read it. If done without an intervening synch, this is a race.
- After the synch (flush), the thread is allowed to read, and by then the flush guarantees the value is in memory, so thread can't determine if the order was jumbled by the other thread prior to the synch (flush).

194

OpenMP 2.5 Memory Model: summary part 2

- Memory ops must be divided into "data" ops and "synch" ops
- Data ops (reads & writes) are not ordered w.r.t. each other
- Data ops **are** ordered w.r.t. synch ops and synch ops are ordered w.r.t. each other
- Cross-thread access to private memory is forbidden.
- Relaxed consistency
  - Temporary view and memory may differ
- Flush operation
  - Moves data between threads
  - Write makes temporary view "dirty"
  - Flush makes temporary view "empty"

195

Backup material

- History
- Foundations of OpenMP
- Memory Models and the Infamous Flush
- Performance optimization in OpenMP
- The future of OpenMP

196

Performance & Scalability Hindrances

- Too fine grained
  - Symptom: high overhead
  - Caused by: Small parallel/critical
- Overly synchronized
  - Symptom: high overhead
  - Caused by: large synchronized sections
  - Dependencies real?
- Load Imbalance
  - Symptom: large wait times
  - Caused by: uneven work distribution

197

Hindrances (continued)

- True sharing
  - Symptom: cache ping ponging, serial region "cost" changes with number of threads.
  - Is parallelism worth it?
- False sharing
  - Symptom: cache ping ponging
- Hardware/OS cannot support
  - No visible timer-based symptoms
  - Hardware counters
  - Thread migration/affinity
  - Enough bandwidth?

198

Performance Tuning

- To tune performance, break all the good software engineering rules (I.e. stop being so portable).
- Step 1
  - Know your application
  - For best performance, also know your compiler, performance tool, and hardware
- The safest pragma to use is:
  - parallel do/for
- Everything else is risky!
  - Necessary evils
- So how do you pick which constructs to use?

199

Understand the Overheads!
Note: all numbers are approximate!

Operation	Minimum overhead (cycles)	Scalability
Hit L1 cache	1-10	Constant
Function call	10-20	Constant
Thread ID	10-50	Constant, log, linear
Integer divide	50-100	Constant
Static do/for, no barrier	100-200	Constant
Miss all caches	100-300	Constant
Lock acquisition	100-300	Depends on contention
Dynamic do/for, no barrier	1000-2000	Depends on contention
Barrier	200-500	Log, linear
Parallel	500-1000	Linear
Ordered	5000-10000	Depends on contention

200

Parallelism worth it?

- When would parallelizing this loop help?

```
DO I = 1, N
   A(I) = 0
ENDDO
```

- Unless you are very careful, not usually
- Some issues to consider
  - Number of threads/processors being used
  - Bandwidth of the memory system
  - Value of N
    - Very large N, so A is not cache contained
  - Placement of Object A
    - If distributed onto different processor caches, or about to be distributed
    - On NUMA systems, when using first touch policy for placing objects, to achieve a certain placement for object A

201

Too fine grained?

- When would parallelizing this loop help?

```
DO I = 1, N
   SUM = SUM + A(I) * B(I)
ENDDO
```

- Know your compiler!
- Some issues to consider
  - # of threads/processors being used
  - How are reductions implemented?
    - Atomic, critical, expanded scalar, logarithmic
  - All the issues from the previous slide about existing distribution of A and B

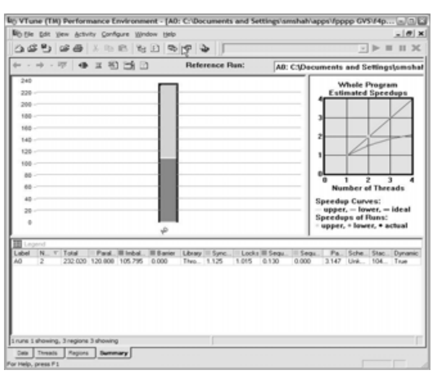
202

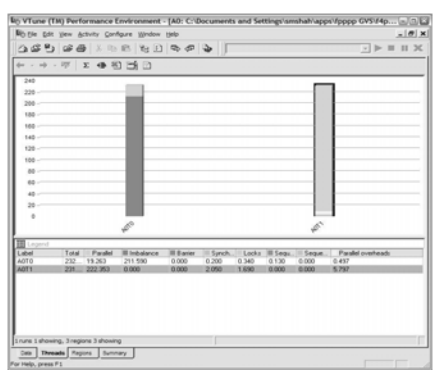
Tuning: Load Balancing

- Notorious problem for triangular loops
- OpenMP aware performance analysis tool can usually pinpoint, but watch out for “summary effects”
- Within a parallel do/for, use the schedule clause
  - Remember, dynamic much more expensive than static
  - Chunked static can be very effective for load imbalance
- When dealing with consecutive do’s/for’s, nowait can help, but be careful about races

203

Load Imbalance: Thread Profiler*





* Thread Profiler is a performance analysis tool from Intel Corporation.

204

Tuning: Critical Sections

- It often helps to chop up large critical sections into finer, named ones
- Original Code

```
#pragma omp critical (foo)
{
    update( a );
    update( b );
}
```

- Transformed Code

```
#pragma omp critical (foo_a)
    update( a );
#pragma omp critical (foo_b)
    update( b );
```

- Still need to avoid wait at first critical!

205

Tuning: Locks Instead of Critical

- Original Code

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

- Idea: cycle through different parts of the array using locks!

- Transformed Code

```
jstart = omp_get_thread_num();
 for( k = 0; k < nlocks; k++ )
 {
   j = ( jstart + k ) % nlocks;
    omp_set_lock( lck[j] );
   for( i=lb[j]; i<ub[j]; i++ )
 {
       a[i] = …
       b[i] = …
       c[i] = …
     }
    omp_unset_lock( lck[j] );
 }
```

- Adapt to your situation

206

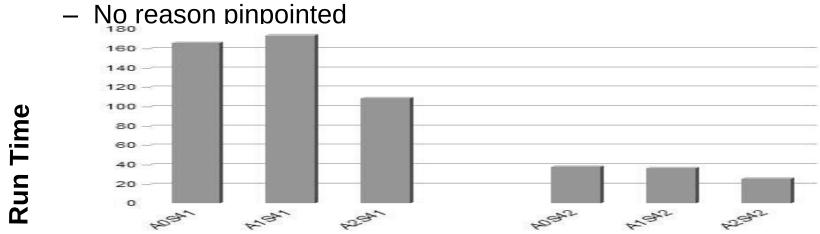
Tuning: Eliminate Implicit Barriers

- Work-sharing constructs have implicit barrier at end
- When consecutive work-sharing constructs modify (& use) different objects, the barrier in the middle can be eliminated
- When same object modified (or used), barrier can be safely removed if iteration spaces align

207

Cache Ping Ponging: Varying Times for Sequential Regions

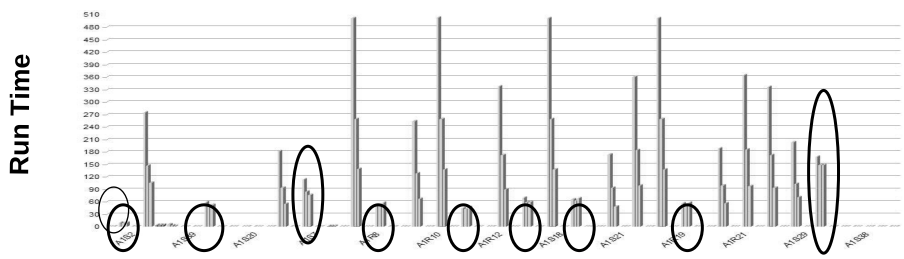
- Picture shows three runs of same program (4, 2, 1 threaded)
- Each set of three bars is a serial region
- Why does runtime change for serial regions?
  - No reason pinpointed
- Time to think!
  - Thread migration
  - Data migration
  - Overhead?



208

Limits: system or tool?

- Picture shows three runs of same program (4, 2, 1 threaded)
- Each set of three bars is a parallel region
- Some regions don’t scale well
  – The collected data does not pinpoint why
- Thread migration or data migration or some system limit?
- Understand Hardware/OS/Tool limits!
  – Your performance tool
  – Your system



209

Performance Optimization Summary

- Getting maximal performance is difficult
- Far too many things can go wrong
- Must understand entire tool chain
  – application
  – hardware
  – O/S
  – compiler
  – performance analysis tool
- With this understanding, it is possible to get good performance

210

Backup material

- History
- Foundations of OpenMP
- Memory Models and the Infamous Flush
- Performance optimization in OpenMP
- ⇨ The future of OpenMP

211

OpenMP's future

- OpenMP is great for array parallelism on shared memory machines.
- But it needs to do so much more:
  - Recursive algorithms.
  - More flexible iterative control structures.
  - More control over its interaction with the runtime environment
  - NUMA
  - Constructs that work with semi-automatic parallelizing compilers

… and much more

212

While workshare Construct

- Share the iterations from a while loop among a team of threads.
- Proposal from Paul Petersen of Intel Corp.

```
int i;
#pragma omp parallel while
  while(i<Last){

     … Independent loop iterations
  }
```

213

Automatic Data Scoping

- Create a standard way to ask the compiler to figure out data scooping.
- When in doubt, the compiler serializes the construct

```
int j;  double x, result[COUNT];
#pragma omp parallel for automatic
  for (j=0; j<COUNT; j++){

     x = bigCalc(j);

     res[j] = hugeCalc(x);
  }
```

Ask the compiler to figure out that "x" should be private.

214

OpenMP Enhancements :
How should we move OpenMP beyond SMP?

- OpenMP is inherently an SMP model, but all shared memory vendors build NUMA and DVSM machines.
- What should we do?
  - Add HPF-like data distribution.
  - Work with thread affinity, clever page migration and a smart OS.
  - Give up?

We have lots of ideas, but we are not making progress towards a consensus view.

This is VERY hard.

OpenMP Enhancements :
OpenMP must be more modular

- Define how OpenMP Interfaces to "other stuff":
  - How can an OpenMP program work with components implemented with OpenMP?
  - How can OpenMP work with other thread environments?
- Support library writers:
  - OpenMP needs an analog to MPI's contexts.

We don't have any solid proposals on the table to deal with these problems.

Other features under consideration

- Error reporting in OpenMP
  - OpenMP assumes all constructs work. But real programs need to be able to deal with constructs that break.
- Parallelizing loop nests
  - People now combine nested loops into a super-loop by hand. Why not let a compiler do this?
- Extensions to Data environment clauses
  - Automatic data scoping
  - Default(mixed) to make scalars private and arrays shared.
- Iteration driven scheduling
  - Pass a vector of block sizes to support non-homogeneous partitioning

… and many more

217