OpenMP: An API for Writing Portable SMP Application Software

Tim Mattson

Intel Corporation

Computational Sciences Laboratory

Rudolf Eigenmann

Purdue University

School of Electrical and Computer Engineering

SC’99 Tutorial: Agenda

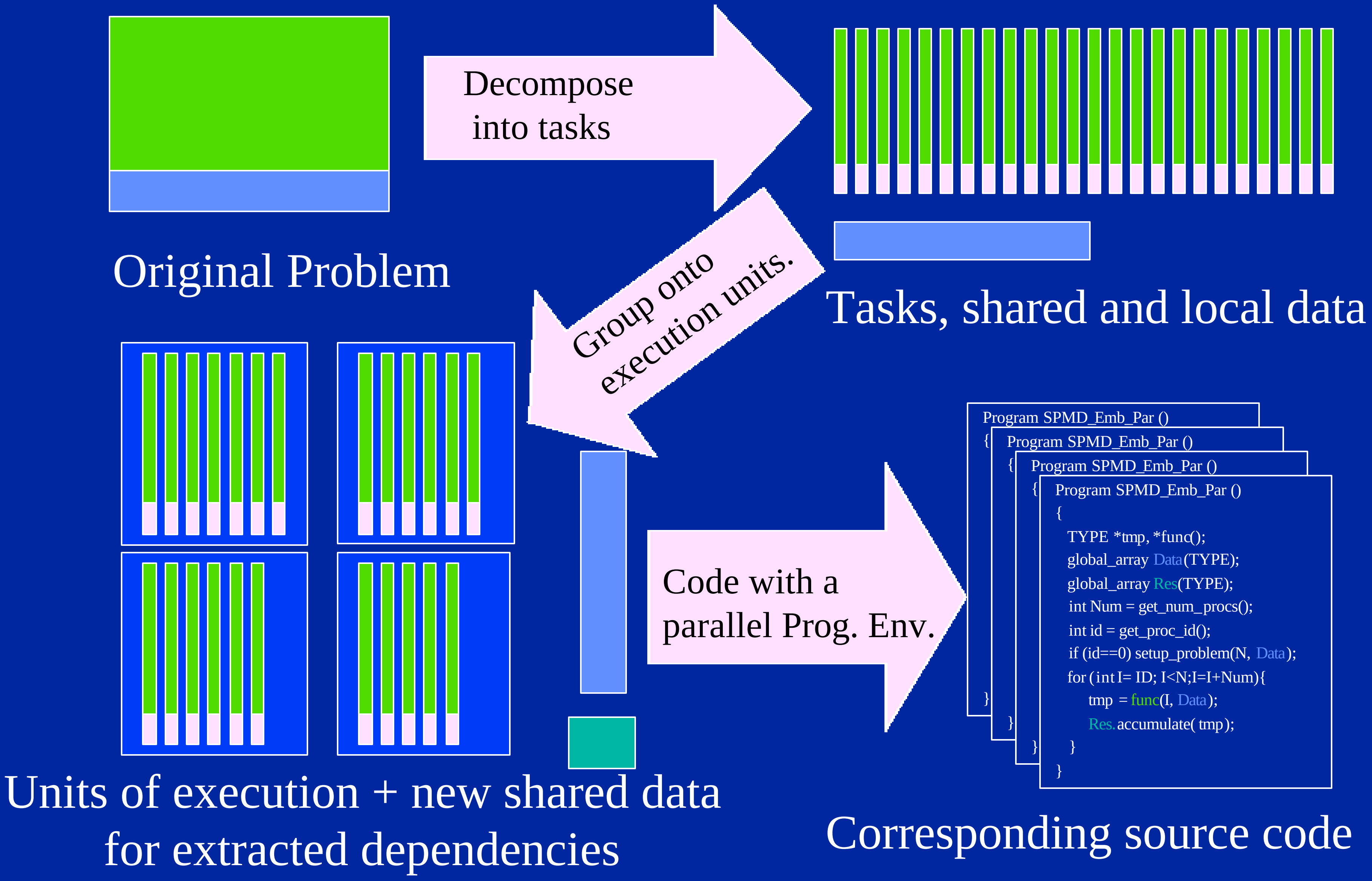
- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

Parallel Computing
What is it?

- Parallel computing is when a program uses concurrency to either:
  - decrease the runtime for the solution to a problem.
  - Increase the size of the problem that can be solved.

Parallel Computing gives you more performance to throw at your problems.

Parallel Computing:
Writing a parallel application.

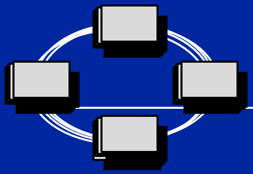

Decompose into tasks
Original Problem
Tasks, shared and local data
Group onto execution units.
Units of execution + new shared data for extracted dependencies
Code with a parallel Prog. Env.
Corresponding source code

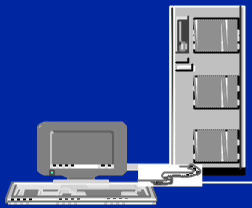
```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

Parallel Computing:

The Hardware is in great shape.

Time	1998	2000	2002
Cluster	16 Boxes 100Mb	32 Boxes VIA	128 Boxes? NGIO
SMP	1-4 CPUs	1-8 CPUs	1-16 CPUs
Processor	Pentium® II Xeon™	IA-64 Merced*	IA-64 McKinley*





Limited by what the market demands - not by technology

*Intel code name

Parallel Computing:
… but where is the software?

- **Most ISV's have ignored parallel computing** **(other than coarse-grained multithreading for GUI's and systems programming)**
- **Why?**
  - **The perceived difficulties of writing parallel software out-weigh the benefits**
- **The *benefits* are clear. To increase the amount of parallel software, we need to reduce the *perceived difficulties*.**

Solution: Effective Standards for parallel programming

- Thread Libraries
  - Win32 API - a low level approach.
  - threads.h++ - a high level class library.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI - www.mpi-softtech.com
- High Level Tools
  - www.apri.com, www.kai.com, www.pgroup.com

Our focus

SC’99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- ➡ OpenMP: A quick overview
- OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

OpenMP: Introduction

```
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
- Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 15 years of SMP practice

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

```
C$OMP DO lastprivate(XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

OpenMP: Supporters*

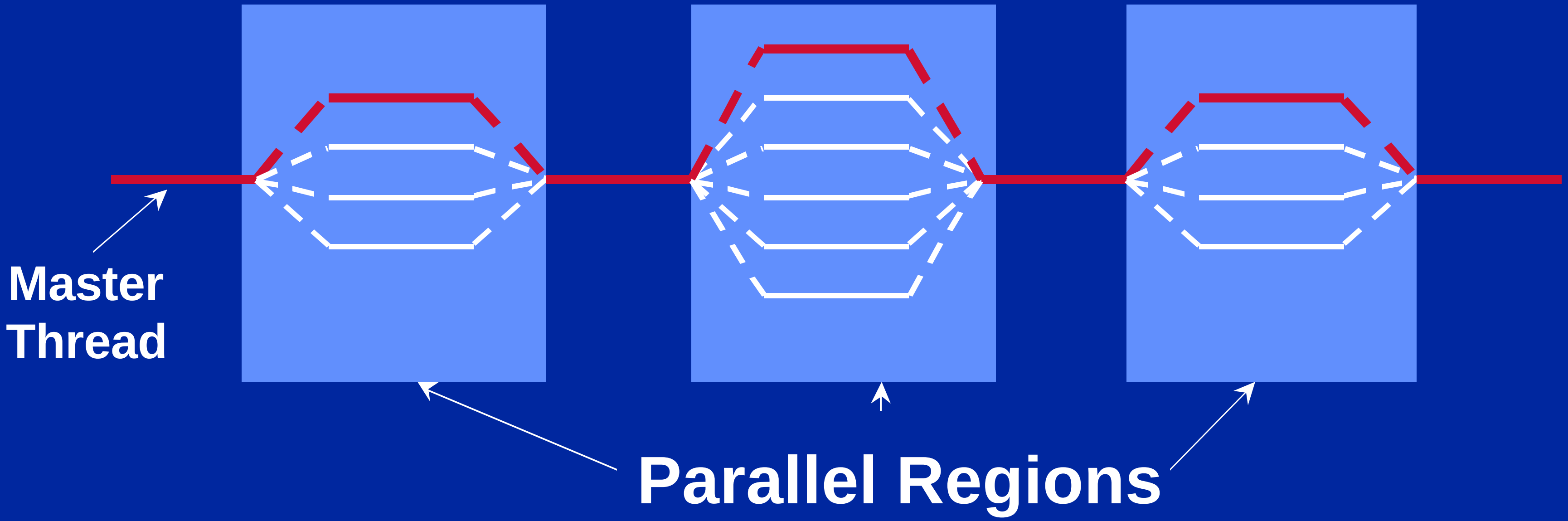
- Hardware vendors
  - Intel, HP, SGI, IBM, SUN, Compaq
- Software tools vendors
  - KAI, PGI, PSR, APR, Absoft
- Applications vendors
  - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, and many others

*These names of these vendors were taken from the OpenMP web site (www.openmp.org). We have made no attempts to confirm OpenMP support, verify conformity to the specifications, or measure the degree of OpenMP utilization.

OpenMP: Programming Model

Fork-Join Parallelism:

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



OpenMP:
How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Sequential Program

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Parallel Program

OpenMP:
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to 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 stored to minimize the need for synchronization.

SC’99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- ➡ OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

OpenMP:
Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp *construct [clause [clause]…]*
  - For Fortran, the directives take one of the forms:

    C$OMP *construct [clause [clause]…]*

    !$OMP *construct [clause [clause]…]*

    *$OMP *construct [clause [clause]…]*
- Since the constructs are directives, an OpenMP program can be compiled by compilers that don’t support OpenMP.

OpenMP:
Structured blocks

- 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 other branches allowed are STOP statements in Fortran and exit() in C/C++.

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

A structured block

```
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
```

<u>Not</u> A structured block

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is basically the same between Fortran and C/C++

OpenMP: Parallel Regions

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

Each thread redundantly executes the code within the structured block

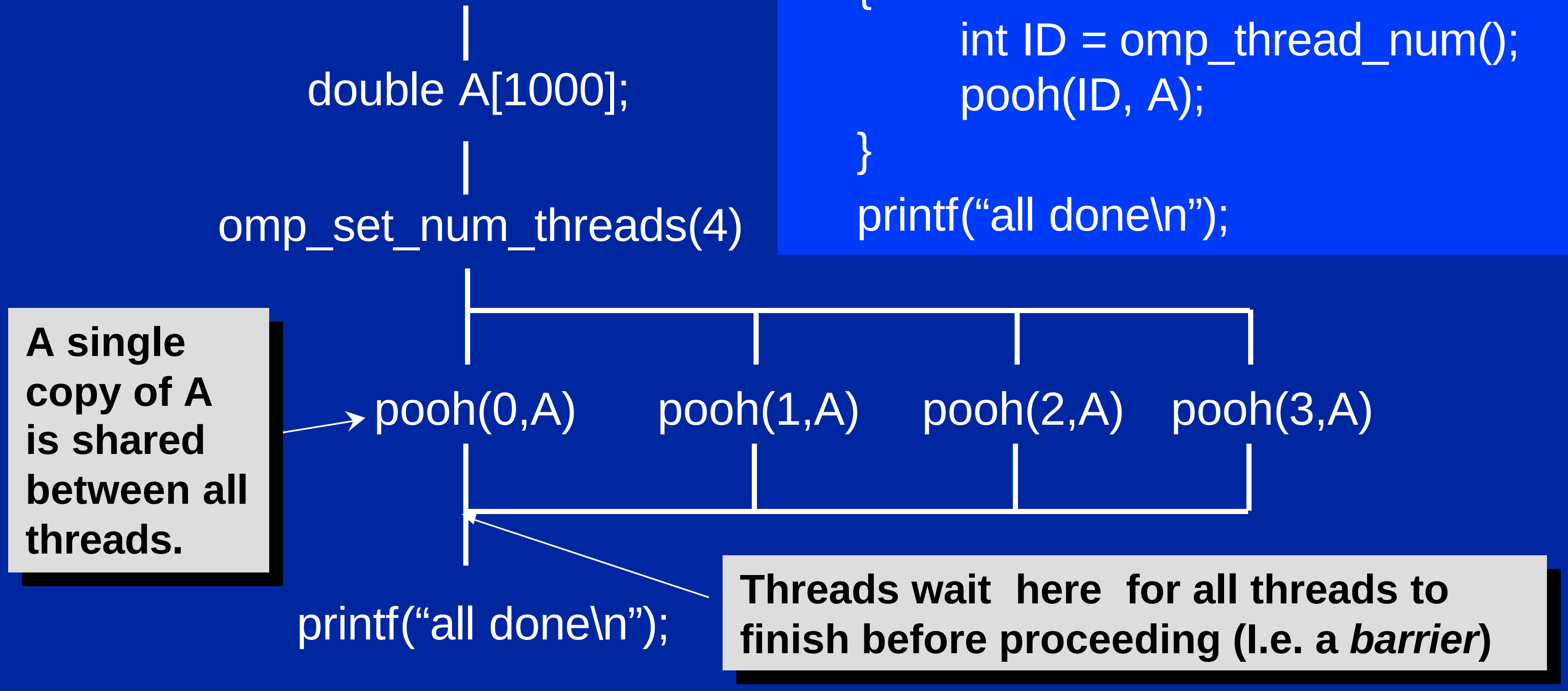
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        pooh(ID,A);
}
```

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

OpenMP: Parallel Regions

- **Each thread executes the same code redundantly.**

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_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*)

Exercise 1:
A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints a simple message (such as "hello world").**
- **Use two separate printf statements and include the thread ID:**

```
int ID = omp_get_thread_num();
printf(" hello(%d) ", ID);
printf(" world(%d) ", ID);
```

- **What do the results tell you about I/O with multiple threads?**

OpenMP: Some subtle details (don't worry about these at first)

- Dynamic mode (the default mode):
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
- Static mode:
  - The number of threads is fixed and controlled by the programmer.
- OpenMP lets you nest parallel regions, but…
  - A compiler can choose to *serialize* the nested parallel region (i.e. use a team with only one thread).

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

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.

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 constuct:
The schedule clause

- **The schedule clause effects 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.

OpenMP: Work-Sharing Constructs

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

```
#pragma omp parallel
#pragma omp sections
{
        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.

OpenMP: Combined Parallel Work-Sharing Constructs

- A short hand notation that combines the Parallel and work-sharing construct.

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

- There’s also a “parallel sections” construct.

Exercise 2:
A multi-threaded “pi” program

- On the following slide, you’ll see a sequential program that uses numerical integration to compute an estimate of PI.
- Parallelize this program using OpenMP. There are several options (do them all if you have time):
  - Do it as an SPMD program using a parallel region only.
  - Do it with a work sharing construct.
- Remember, you’ll need to make sure multiple threads don’t overwrite each other’s variables.

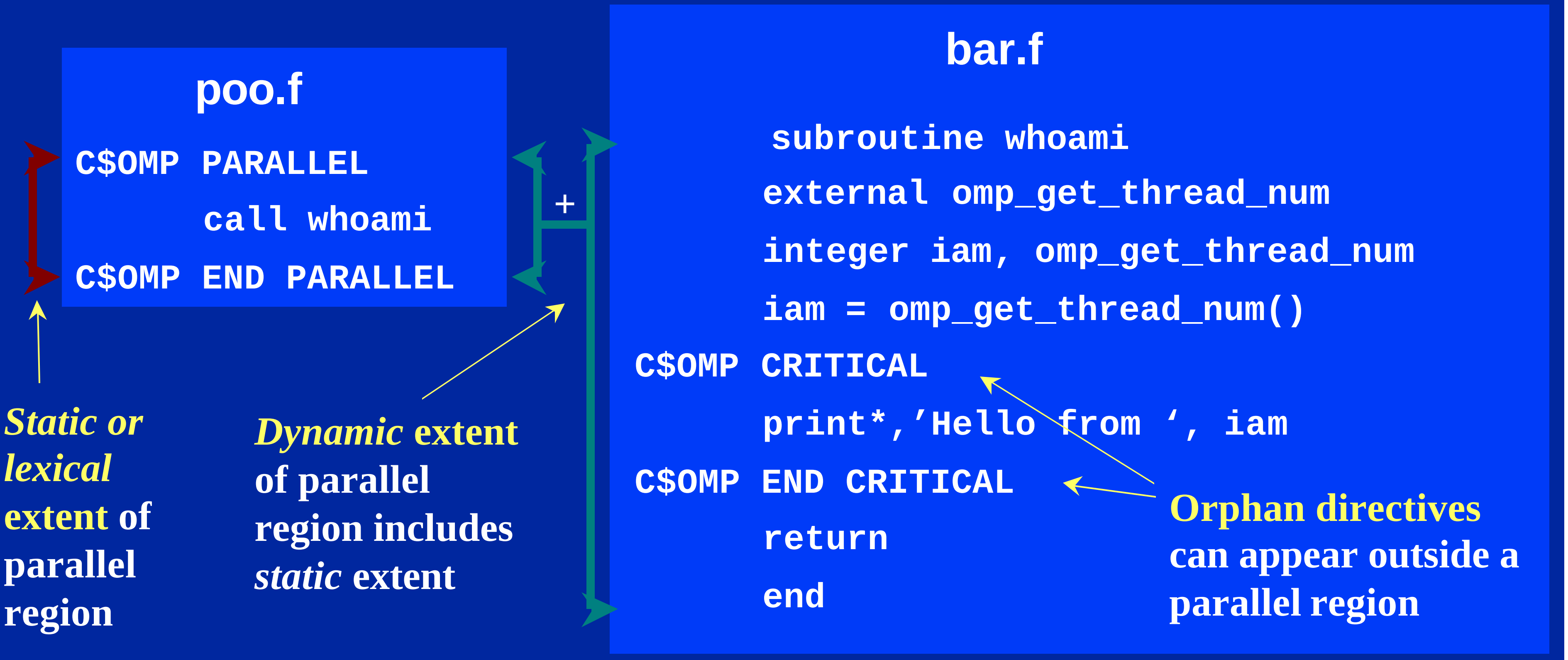
PI Program:
The sequential program

```
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=1;i<= num_steps; i++){
                     x = (i-0.5)*step;
                     sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP:

More details: Scope of OpenMP constructs

OpenMP constructs can span multiple source files.



poo.f

```
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
```

Static or lexical extent of parallel region

Dynamic extent of parallel region includes *static* extent

Orphan directives can appear outside a parallel region

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - ◆Parallel Regions
  - ◆Worksharing
  - ➡ ◆Data Environment
  - ◆Synchronization
  - ◆Runtime functions/environment variables

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.

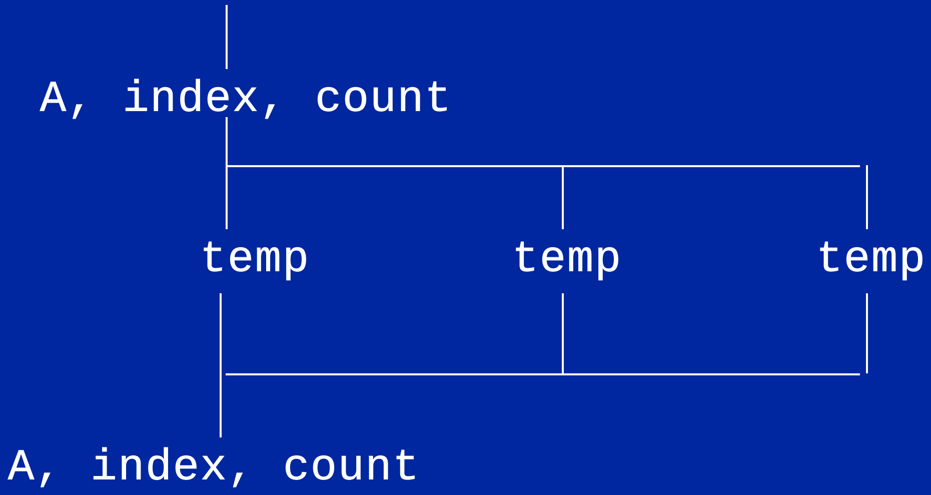
Data Environment:
Example storage attributes

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

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

A, index and count are shared by all threads.

temp is local to each thread



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 only apply to the *lexical extent* of the OpenMP construct.

- 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.

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

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

IS was not initialized

Regardless of initialization, IS is undefined at this point

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

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 iteration (I.e. for J=1000)

OpenMP:
Another data environment example

- Here’s an example of PRIVATE and FIRSTPRIVATE

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

- 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.

OpenMP:
Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to specify)
- 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).

OpenMP:
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
```

These two codes are equivalent

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.

A threadprivate example

Consider two different routines called within a parallel region.

```
      subroutine poo
      parameter (N=1000)
      common/buf/A(N),B(N)
C$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)
C$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/.

OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - reduction (op : list)
- The variables in “list” must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - pair wise “op” is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

OpenMP:
Reduction example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
        int i;
        double ZZ, func(), res=0.0;
        omp_set_num_threads(NUM_THREADS)
#pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++){
              ZZ = func(I);
              res = res + ZZ;
        }
}
```

Exercise 3:
A multi-threaded "pi" program

- Return to your "pi" program and this time, use private, reduction and a worksharing construct to parallelize it.
- See how similar you can make it to the original sequential program.

OpenMP: Some subtle details (don't worry about these at first)

- The data scope clauses take a list argument
  - The list can be a common block name with is a short hand for listing all the variables in the common block.
- Default private for some loop indices:
  - Fortran: loop indices are private even if they are specified as shared.
  - C: Loop indices on "work-shared loops" are private when they otherwise would be shared.
- Not all privates are undefined
  - Allocatable arrays in Fortran
  - Class type (I.e. non-POD) variables in C++.

See the OpenMP spec. for more details.

OpenMP: More subtle details (don’t worry about these at first)

- Variables privitized in a parallel region can not be reprivitzied on an enclosed worksharing directive.
- Assumed size and assumed shape arrays can not be privitized.
- Fortran pointers or allocatable arrays can be private or shared but not lastprivate or firstprivate.
- When a common block is listed in a private, firstprivate, or lastprivate clause, its constituent elements can’t appear in other data scope clauses.
- If an element of a shared common block is privitized, it is no longer storage associated with the common block.

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - atomic
  - critical section
  - barrier
  - flush
  - ordered
  - single
  - master

We discuss this here, but it really isn't a synchronization construct. It's a work-sharing construct that includes synchronization.

We discus this here, but it really isn't a synchronization construct.

OpenMP: Synchronization

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

```
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
```

OpenMP: Synchronization

- Atomic is a special case of a critical section that can be used for certain simple statements.
- It applies only to the update of a memory location (the update of X in the following example)

```
C$OMP PARALLEL PRIVATE(B)
          B =  DOIT(I)
C$OMP ATOMIC
          X = X + B

C$OMP END PARALLEL
```

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 construct enforces the sequential order for a block.

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

OpenMP: Synchronization

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

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

OpenMP: Synchronization

- **The single construct denotes a block of code that is executed by only one thread.**
- **A barrier and a flush are 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();
}
```

OpenMP: Synchronization

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - 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.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

This is a confusing construct and we won’t say much about it. To learn more, consult the OpenMP specifications.

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.

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 critical
end single (except when nowiat is used)
```

- Flush is implied on the following OpenMP constructs:

```
barrier
critical, end critical
end do
end parallel
```

```
end sections
end single
ordered, end ordered
```

OpenMP: Some subtle details on directive nesting

- *For*, *sections* and *single* directives binding to the same parallel region can't be nested.
- Critical sections with the same name can't be nested.
- *For*, *sections*, and *single* can not appear in the dynamic extent of *critical*, *ordered* or *master*.
- *Barrier* can not appear in the dynamic extent of *for*, *ordered*, *sections*, *single*., *master* or *critical*
- *Master* can not appear in the dynamic extent of *for*, *sections* and *single*.
- *Ordered* are not allowed inside *critical*
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.

OpenMP: Contents

- OpenMP’s constructs fall into 5 categories:
  - ◆Parallel Regions
  - ◆Worksharing
  - ◆Data Environment
  - ◆Synchronization
  - ➡ ◆Runtime functions/environment variables

OpenMP: Library routines

- Lock routines
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
- 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()
    - Turn on/off nesting and dynamic mode
        - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
    - Are we in a parallel region?
        - omp_in_parallel()
    - How many processors in the system?
        - omp_num_procs()

OpenMP: Library Routines

- Protect resources with locks.

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

OpenMP: Library Routines

- To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```
#include <omp.h>
void main()
{  omp_set_dynamic(0);
   omp_set_num_threads(4);
#pragma omp parallel
   {  int id=omp_get_thread_num();
      do_lots_of_stuff(id);  }
}
```

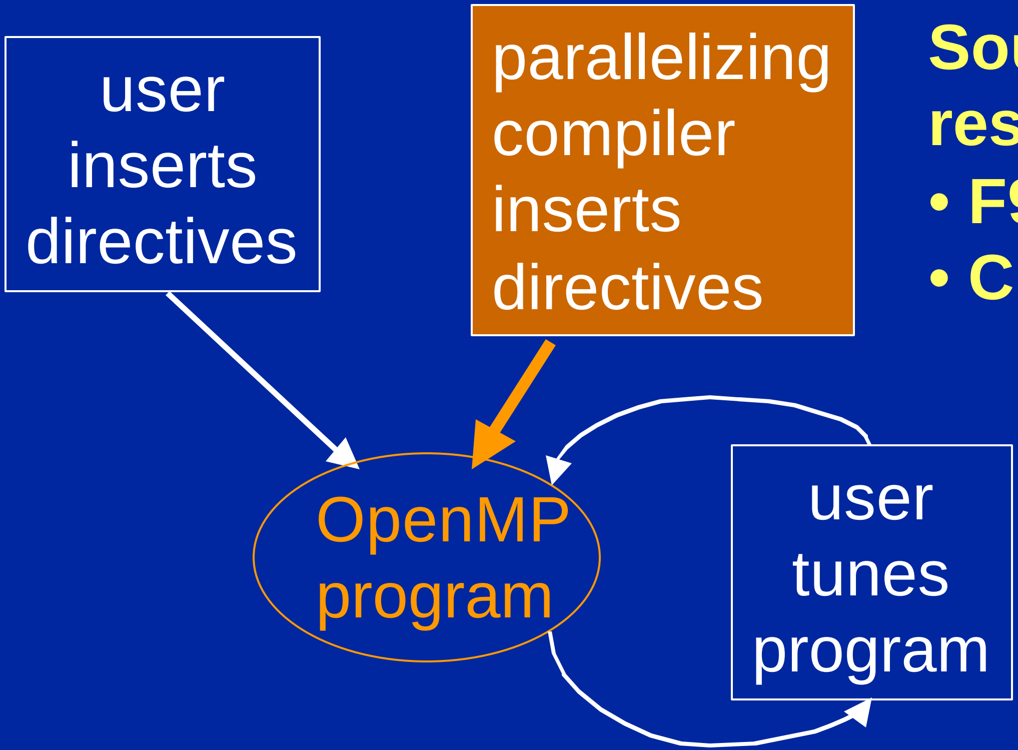
OpenMP: Environment Variables

- 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*
- Can the program use a different number of threads in each parallel region?
    - OMP_DYNAMIC TRUE || FALSE
- Will nested parallel regions create new teams of threads, or will they be serialized?
    - OMP_NESTED TRUE || FALSE

SC’99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- ➡ Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

Generating OpenMP Programs Automatically



Source-to-source restructurers:
- **F90 to F90/OpenMP**
- **C   to C/OpenMP**

Examples:
- SGI F77 compiler (-apo -mplist option)
- Polaris compiler

The Basics About Parallelizing Compilers

- **Loops are the primary source of parallelism in scientific and engineering applications.**
- **Compilers detect loops that have independent iterations.**

```
DO I=1,N
    A(expression1) = …
        … = A(expression2)
ENDDO
```

The loop is independent if, for different iterations, *expression1* is always different from *expression2*

Basic Compiler Transformations

Data privatization:

```
DO i=1,n
     work(1:n) = ….
      .
      .
      .
      …  =  work(1:n)
ENDDO
```

$\rightarrow$

```
C$OMP PARALLEL DO
C$OMP+ PRIVATE (work)
DO i=1,n
     work(1:n) = ….
      .
      .
      .
      …  =  work(1:n)
ENDDO
```

Each processor is given a separate version of the private data, so there is no sharing conflict

Basic Compiler Transformations

Reduction recognition:

```
DO i=1,n
      ...
  sum = sum + a(i)
      …
 ENDDO
```

$\longrightarrow$

```
C$OMP PARALLEL DO
C$OMP+ REDUCTION (+:sum)
DO i=1,n
     ...
     sum = sum + a(i)
      …
ENDDO
```

Each processor will accumulate partial sums, followed by a combination of these parts at the end of the loop.

Basic Compiler Transformations

Induction variable substitution:

```
i1 = 0
i2 = 0
DO I =1,n
    i1 = i1 + 1
    B(i1) = ...

    i2 = i2 + i
    A(i2) = …

 ENDDO
```

$\rightarrow$

```
C$OMP PARALLEL DO
DO I =1,n

    B(i) = ...

    A((i**2 + i)/2) = …

ENDDO
```

The original loop contains data dependences: each processor modifies the shared variables *i1*, and *i2*.

Compiler Options

Examples of options from the KAP parallelizing compiler (KAP includes some 60 options)

- **optimization levels**
  - **optimize :** simple analysis, advanced analysis, loop interchanging, array expansion
  - **aggressive:** pad common blocks, adjust data layout
- **subroutine inline expansion**
  - inline all, specific routines, how to deal with libraries
- **try specific optimizations**
  - e.g., recurrence and reduction recognition, loop fusion

  (These transformations may degrade performance**)**

More About Compiler Options

- **Limits on amount of optimization:**
  - e.g., size of optimization data structures, number of optimization variants tried
- **Make certain assumptions:**
  - e.g., array bounds are not violated, arrays are not aliased
- **Machine parameters:**
  - e.g., cache size, line size, mapping
- **Listing control**

Note, compiler options can be a substitute for advanced compiler strategies. If the compiler has limited information, the user can help out.

Inspecting the Translated Program

- Source-to-source restructurers:
  - transformed source code is the actual output
  - Example: KAP
- Code-generating compilers:
  - typically have an option for viewing the translated (parallel) code
  - Example: SGI f77 -apo -mplist

This can be the starting point for code tuning

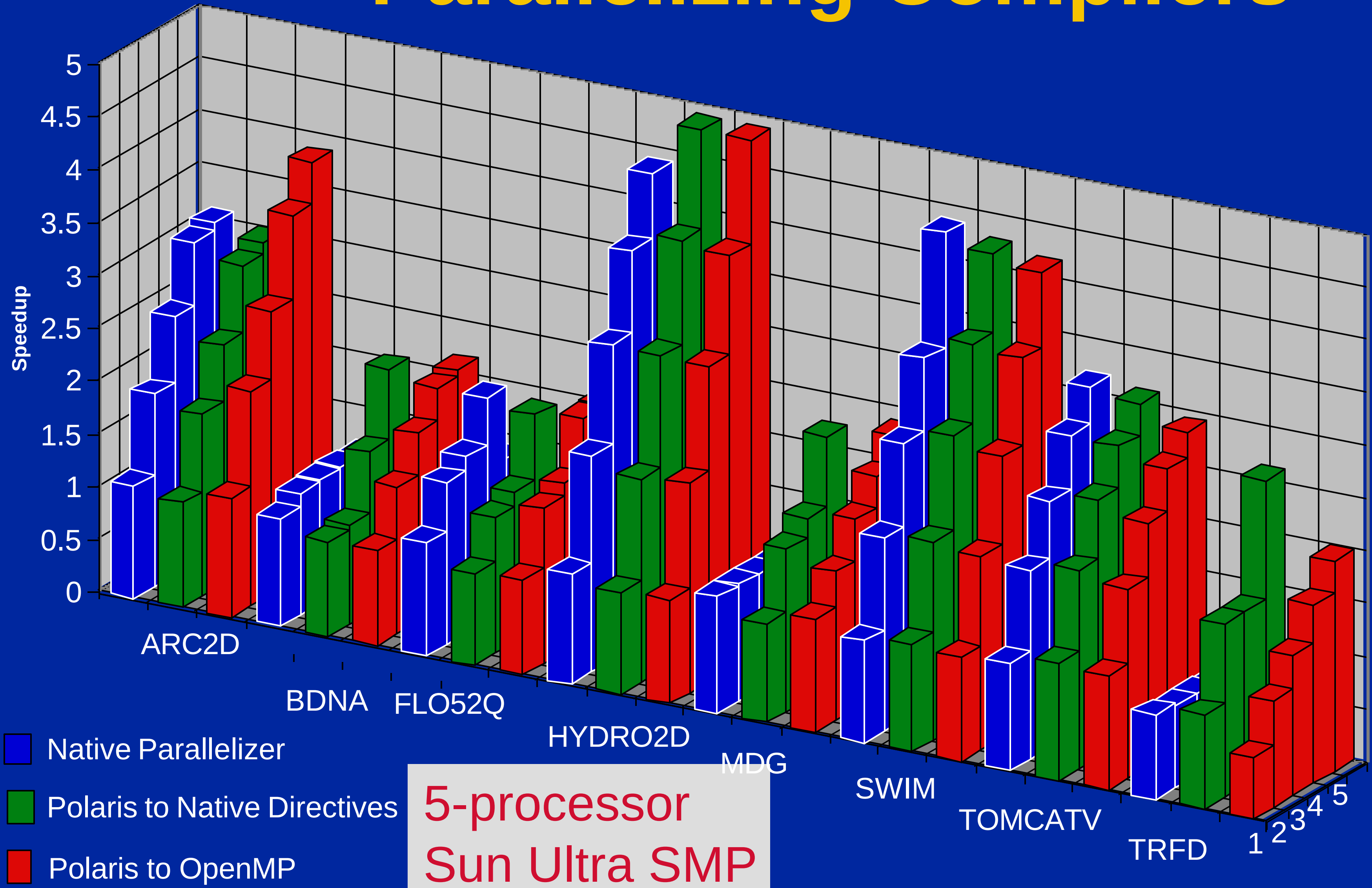
Compiler Listing

The listing gives many useful clues for improving the performance:

- Loop optimization tables
- Reports about data dependences
- Explanations about applied transformations
- The annotated, transformed code
- Calling tree
- Performance statistics

The type of reports to be included in the listing can be set through compiler options.

Performance of Parallelizing Compilers



5-processor
Sun Ultra SMP

Tuning Automatically-Parallelized Code

- This task is similar to explicit parallel programming (will be discussed later)
- Two important differences :
  - The compiler gives hints in its listing, which may tell you where to focus attention. E.g., which variables have data dependences.
  - You don’t need to perform all transformations by hand. If you expose the right information to the compiler, it will do the translation for you.

    (E.g., `C$assert independent`)

Why Tuning Automatically-Parallelized Code?

Hand improvements can pay off because

- compiler techniques are limited
  - E.g., array reductions are parallelized by only few compilers
- compilers may have insufficient information
  - E.g.,
    - loop iteration range may be input data
    - variables are defined in other subroutines (no interprocedural analysis)

SC'99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- ➡ OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

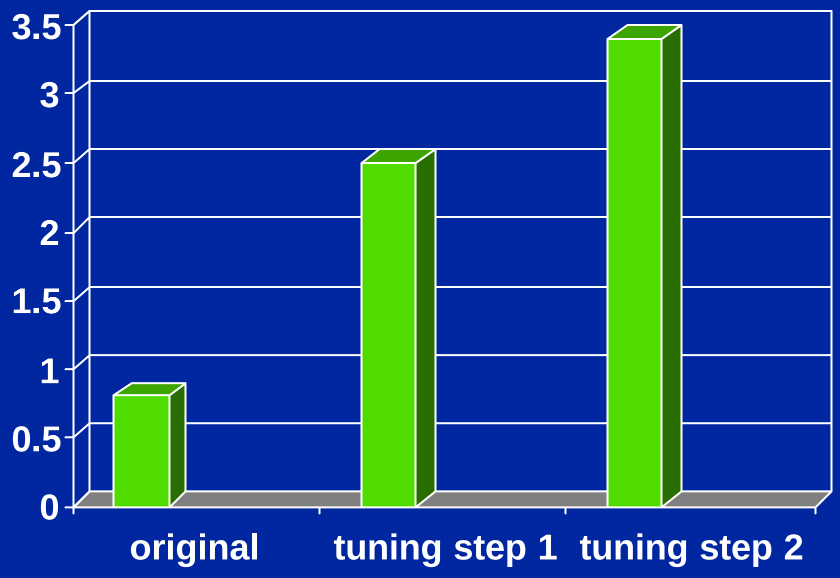
Performance Tuning and Case Studies with Realistic Applications

1. Performance tuning of several benchmarks
2. Case study of a large-scale application

Performance Tuning Example 1: MDG

- MDG: A Fortran code of the “Perfect Benchmarks”.
- Automatic parallelization does not improve this code.

These performance improvements were achieved through manual tuning on a 4-processor Sun Ultra:



MDG: Tuning Steps

Step 1: **parallelize the most time-consuming loop. It consumes 95% of the serial execution time. This takes:**

- **array privatization**
- **reduction parallelization**

Step 2: **balancing the iteration space of this loop.**

- **Loop is “triangular”. By default unbalanced assignment of iterations to processors.**

MDG Code Sample

Structure of the most time-consuming loop in MDG:

Original

```
c1  =  x(1)>0

c2  =  x(1:10)>0


DO i=1,n
  DO j=i,n
    IF (c1) THEN rl(1:100) = …
    …
    IF (c2) THEN … = rl(1:100)
    sum(j) = sum(j) + …
  ENDDO
ENDDO
```

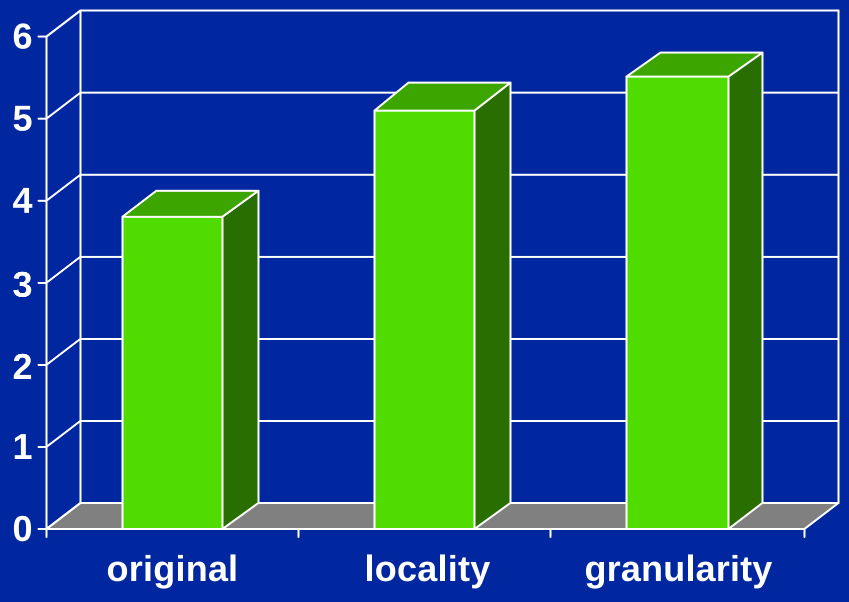
Parallel

```
c1  =  x(1)>0
c2  =  x(1:10)>0
Allocate(xsum(1:#proc,n))

C$OMP PARALLEL DO
C$OMP+ PRIVATE (I,j,rl,id)
C$OMP+ SCHEDULE (STATIC,1)
DO  i=1,n
   id = omp_get_thread_num()
   DO j=i,n
      IF (c1) THEN rl(1:100) = …
      …
      IF (c2) THEN … = rl(1:100)
      xsum(id,j) = xsum(id,j) + …
   ENDDO
ENDDO

C$OMP PARALLEL DO
DO i=1,n
   sum(i)=sum(i)+xsum(1:#proc,i)
ENDDO
```

Performance Tuning Example 2: ARC2D

ARC2D: A Fortran code of the “Perfect Benchmarks”.

ARC2D is parallelized very well by available compilers. However, the mapping of the code to the machine could be improved..



ARC2D: Tuning Steps

- **Step 1:**

  Loop interchanging increases cache locality through stride-1 references.

- **Step 2:**

  Move parallel loops to outer positions

- **Step 3:**

  Move synchronization points outward

- **Step 4:**

  Coalesce loops

ARC2D: Code Samples

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
      DO j = jlow, jup
         DO k = 2, kmax-1
            r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
            r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
           coef(j, k) = ABS(r1/r2)
        ENDDO
      ENDDO
!$OMP END PARALLEL
```

Loop interchanging increases cache locality

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(R1,R2,K,J)
      DO k = 2, kmax-1
         DO j = jlow, jup
            r1 = prss(jminu(j), k) + prss(jplus(j), k) + (-2.)*prss(j, k)
            r2 = prss(jminu(j), k) + prss(jplus(j), k) + 2.*prss(j, k)
            coef(j, k) = ABS(r1/r2)
        ENDDO
      ENDDO
!$OMP END PARALLEL
```

ARC2D: Code Samples

```
!$OMP PARALLEL
!$OMP+PRIVATE(LDI,LD2,LD1,J,LD,K)
    DO k = 2+2, ku-2, 1
!$OMP DO
      DO j = jl, ju
        ld2 = a(j, k)
        ld1 = b(j, k)+(-x(j, k-2))*ld2
        ld = c(j, k)+(-x(j, k-1))*ld1+(-y(j, k-1))*ld2
        ldi = 1./ld
        f(j, k, 1) = ldi*(f(j, k, 1)+(-f(j, k-2, 1))*ld2+(-f(j, k-1, 1))*ld1)
        f(j, k, 2) = ldi*(f(j, k, 2)+(-f(j, k-2, 2))*ld2+(-f(jk-2, 2))*ld1)
        x(j, k) = ldi*(d(j, k)+(-y(j, k-1))*ld1)
        y(j, k) = ldi*e(j, k)*ldi
      ENDDO
!$OMP END DO  NOWAIT
    ENDDO
!$OMP END PARALLEL
```

Increasing parallel loop granularity through NOWAIT clause

ARC2D: Code Samples

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(n, k,j)
      DO  n = 1, 4
       DO k = 2, kmax-1
        DO j = jlow, jup
         q(j, k, n) = q(j, k, n)+s(j, k, n)
         s(j, k, n) = s(j, k, n)*phic
        ENDDO
       ENDDO
      ENDDO
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(nk,n,k,j)
      DO nk = 0,4*(kmax-2)-1
       n = nk/(kmax-2) + 1
       k = MOD(nk,kmax-2)+2
       DO j = jlow, jup
        q(j, k, n) = q(j, k, n)+s(j, k, n)
        s(j, k, n) = s(j, k, n)*phic
       ENDDO
      ENDDO
!$OMP END PARALLEL
```

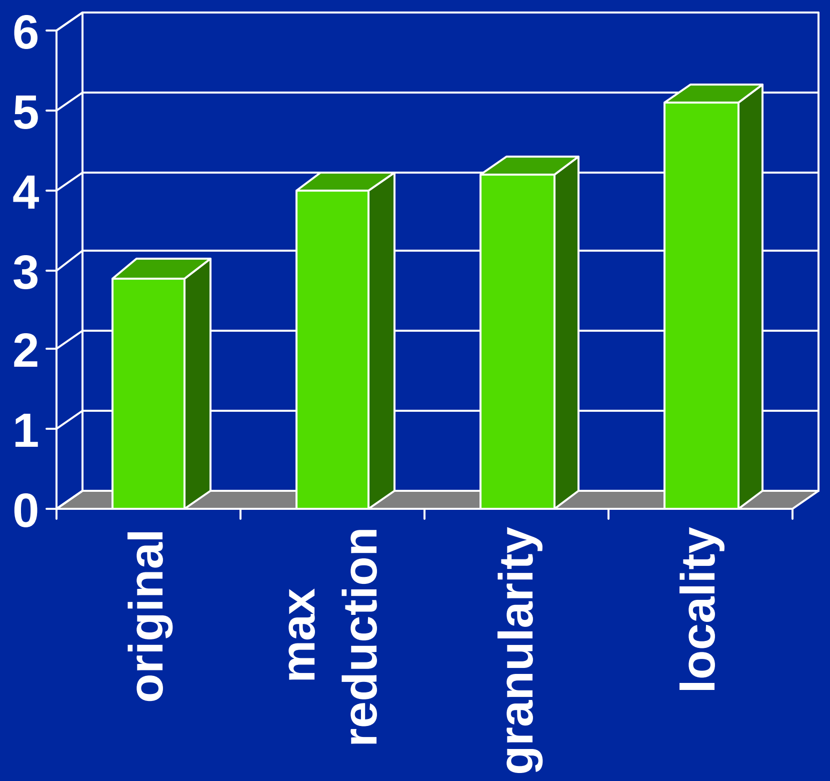
$\longrightarrow$

Increasing parallel loop granularity though loop coalescing

Performance Tuning Example 3: TOMCATV

TOMCATV: A Fortran code of the SPEC 95 benchmarks.

TOMCATV is parallelized very well by available compilers. However, like in ARC2D, the mapping of the code to the machine could be improved.



TOMCATV: Tuning Steps

- **Step1:**

  Parallelizing MAX reduction

- **Step2:**

  Cache optimization and granularity increase through loop interchange and array transpose

TOMCATV Code Samples

Parallelizing MAX reduction:

```
DO j = 2, n
  DO i = 2, n
    rxm= MAX(rxm,  ABS(rx(i, j)))
    rym= MAX(rym,  ABS(ry(i, j)))
  ENDDO
ENDDO
```

```
C$OMP PARALLEL DO
C$OMP+ REDUCTION (MAX:rxm,rxy)
DO j = 2, n
  DO i = 2, n
    rxm= MAX(rxm, ABS(rx(i, j)))
    rym = MAX(rym, ABS(ry(i, j)))
  ENDDO
END DO
```

TOMCATV Code Samples

Better cache locality through array transposition:

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

What Tools Did We Use for Performance Analysis and Tuning?

- Compilers
  - the starting point for our performance tuning was always the compiler-parallelized program.
  - It reports: parallelized loops, data dependences
- Subroutine and loop profilers
  - focusing attention on the most time-consuming loops is absolutely essential.
- Performance tables:
  - typically comparing performance differences at the loop level.

Guidelines for Fixing “Performance Bugs”

- The methodology that worked for us:
  - Use compiler-parallelized code as a starting point
  - Get loop profile and compiler listing
  - Inspect time-consuming loops (biggest potential for improvement)
    - Case 1. Check for parallelism where the compiler could not find it
    - Case 2. Improve parallel loops where the speedup is limited

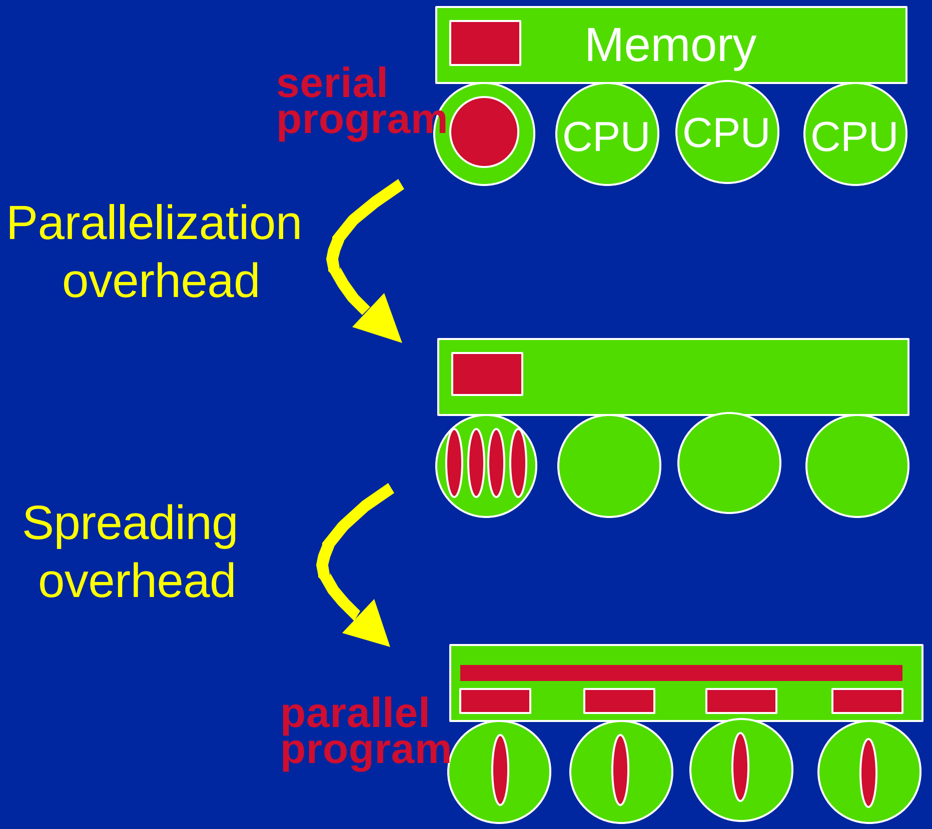
Performance Tuning

Case 1: if the loop is not parallelized automatically, do this:

- Check for parallelism:
  - read the compiler explanation
  - a variable may be independent even if the compiler detects dependences (compilers are conservative)
  - check if conflicting array is privatizable (compilers don’t perform array privatization well)
- If you find parallelism, add OpenMP parallel directives, or make the information explicit for the parallelizer

Performance Tuning

Case 2: if the loop is parallel but does not perform well, consider several optimization factors:



High overheads are caused by:

- parallel startup cost
- small loops
- additional parallel code
- over-optimized inner loops
- less optimization for parallel code

- load imbalance
- synchronized section
- non-stride-1 references
- many shared references
- low cache affinity

Case Study of a Large-Scale Application

Converting a Seismic Processing Application to OpenMP

- Overview of the Application
- Basic use of OpenMP
- OpenMP Issues Encountered
- Performance Results

Overview of Seismic

- Representative of modern seismic processing programs used in the search for oil and gas.
- 20,000 lines of Fortran. C subroutines interface with the operating system.
- Available in a serial and a parallel variant.
- Parallel code is available in a message-passing and an OpenMP form.
- Is part of the SPEChpc benchmark suite. Includes 4 data sets: small to x-large.

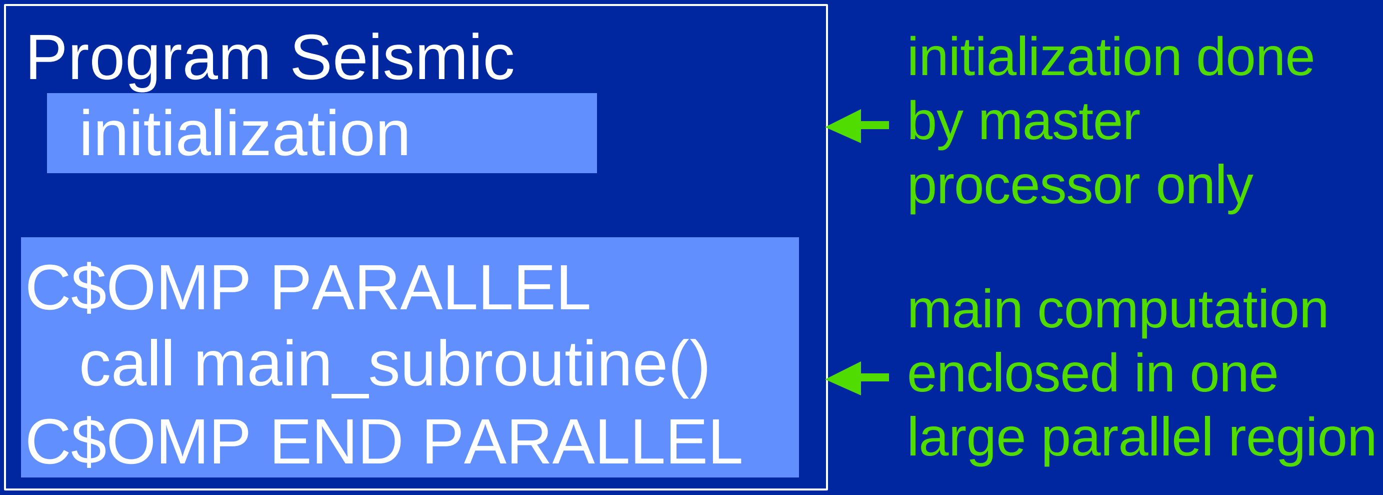
Seismic:
Basic Characteristics

- Program structure:
  - ◆240 Fortran and 119 C subroutines.
- Main algorithms:
  - ◆ FFT, finite difference solvers
- Running time of Seismic (@ 500MFlops):
  - ◆small data set: 0.1 hours
  - ◆x-large data set: 48 hours
- IO requirement:
  - ◆small data set: 110 MB
  - ◆x-large data set: 93 GB

Basic OpenMP Use: Parallelization Scheme

- **Split into *p* parallel tasks**

  (*p* = number of processors)



```
Program Seismic
  initialization

C$OMP PARALLEL
  call main_subroutine()
C$OMP END PARALLEL
```

initialization done by master processor only
main computation enclosed in one large parallel region

$\rightarrow$ SPMD execution scheme

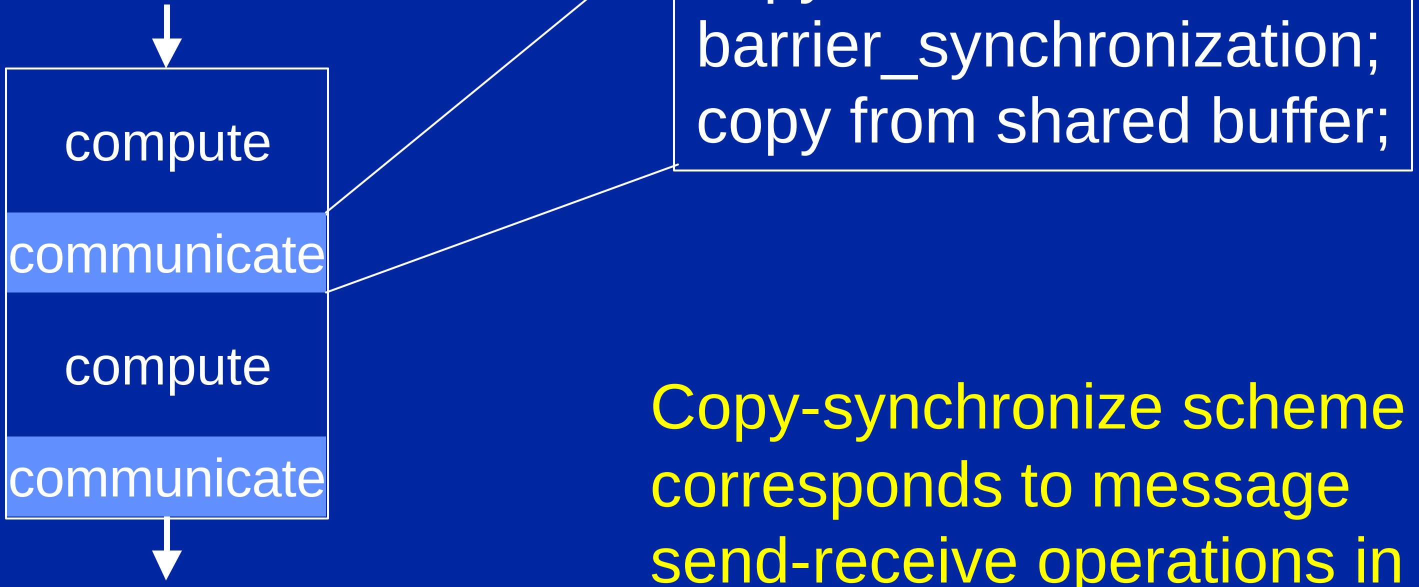
Basic OpenMP Use:
Data Privatization

- Most data structures are ***private***,
  i.e., Each thread has its own copy.
- Syntactic forms:

```
Program Seismic
...
C$OMP PARALLEL
C$OMP+PRIVATE(a)
   a = "local computation"
   call x()
C$END PARALLEL
```

```
Subroutine x()
common /cc/ d
c$omp threadprivate (/cc/)
real b(100)
...
b() = "local computation"
d = "local computation"
...
```

Basic OpenMP Use:
Synchronization and Communication



copy to shared buffer;
barrier_synchronization;
copy from shared buffer;

Copy-synchronize scheme corresponds to message send-receive operations in MPI programs

OpenMP Issues: Mixing Fortran and C

- **Bulk of computation is done in Fortran**
- **Utility routines are in C:**
  - **IO operations**
  - **data partitioning routines**
  - **communication/synchronization operations**
- **OpenMP-related issues:**
  - **IF C/OpenMP compiler is not available, data privatization must be done through “expansion”.**
  - **Mix of Fortran and C is implementation dependent**

Data privatization in OpenMP/C

```
#pragma omp thread private (item)
float item;
void x(){
  ... = item;
}
```

Data expansion in absence of OpenMP/C

```
float item[num_proc];
void x(){
  int thread;
  thread = omp_get_thread_num();
  ... = item[thread];
}
```

OpenMP Issues: Broadcast Common Blocks

```
common /cc/ cdata
common /dd/ ddata
c initialization
   cdata = ...
   ddata = ...

C$OMP PARALEL
C$OMP+COPYIN(/cc/, /dd/)
   call main_subroutine()
C$END PARALLEL
```

Issues in Seismic:

- At the start of the parallel region is it not yet known which common blocks need to be copied in.

Solution:

- copy-in all common blocks
  $\rightarrow$ overhead

OpenMP Issues:
Multithreading IO and malloc

IO routines and memory allocation are called within parallel threads, inside C utility routines.

- OpenMP requires all standard libraries and instrinsics to be thread-save. However the implementations are not always compliant.

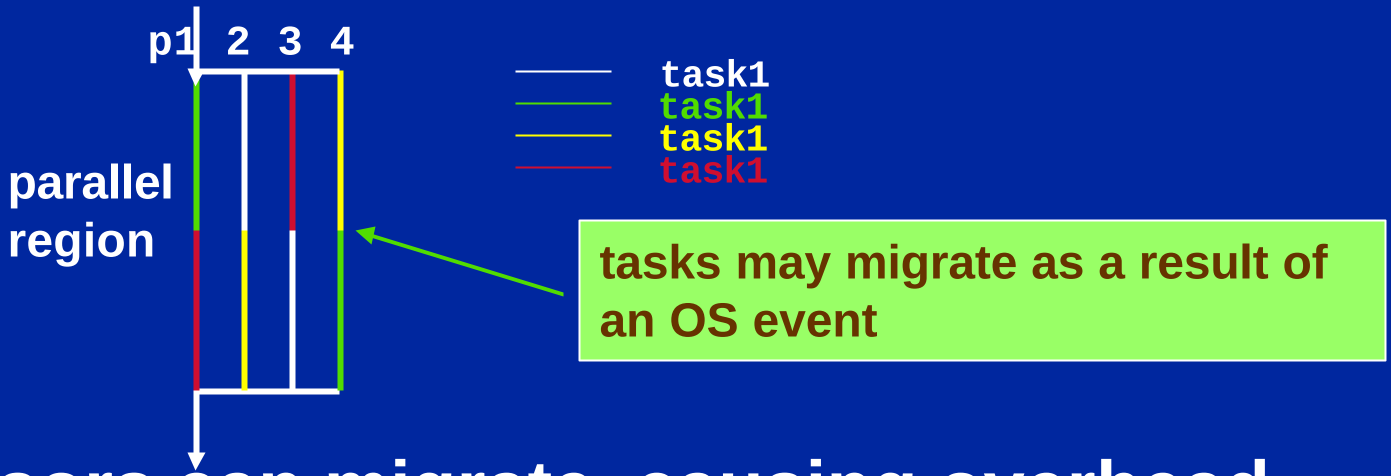
  ® system-dependent solutions need to be found

- The same issue arises if standard C routines are called inside a parallel Fortran region or in non-standard syntax.

  Standard C compilers do not know anything about OpenMP and the thread-safe requirement.

OpenMP Issues: Processor Affinity

- OpenMP currently does not specify or provide constructs for controlling the binding of threads to processors.

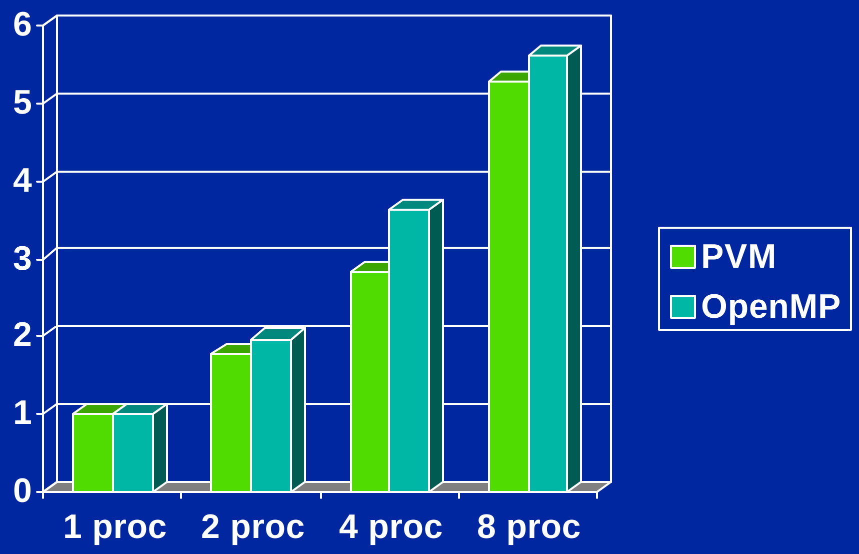


- Processors can migrate, causing overhead. This behavior is system-dependent.

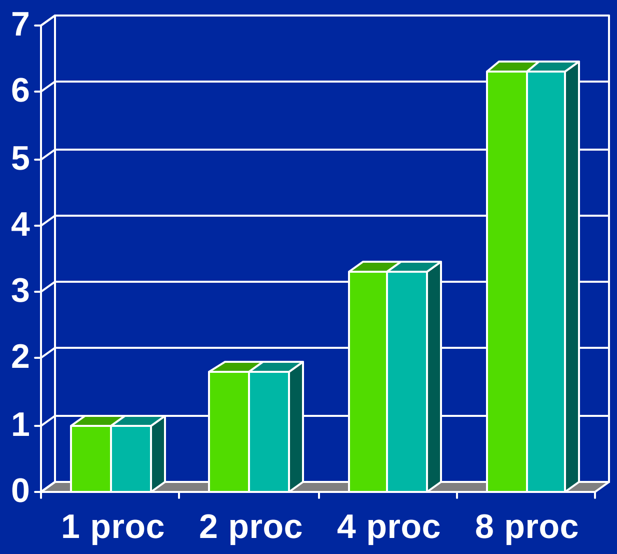
System-dependent solutions may be available.

Performance Results

Speedups of Seismic on an SGI Challenge system



small data set



medium data set

SC’99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- The future of OpenMP

SMP Programming errors

- Shared memory parallel programming is a mixed bag:
  - It saves the programmer from having to map data onto multiple processors. In this sense, its much easier.
  - It opens up a range of new errors coming from unanticipated shared resource conflicts.

2 major SMP errors

- Race Conditions
  - The outcome of a program depends on the detailed timing of the threads in the team.
- Deadlock
  - Threads lock up waiting on a locked resource that will never become free.

Race Conditions

```
C$OMP PARALLEL SECTIONS
      A = B + C
C$OMP SECTION
      B = A + C
C$OMP SECTION
      C = B + A
C$OMP END PARALLEL SECTIONS
```

- **The result varies unpredictably based on detailed order of execution for each section.**
- **Wrong answers produced without warning!**

Race Conditions:
A complicated solution

```
      ICOUNT = 0
C$OMP PARALLEL SECTIONS
      A = B + C
      ICOUNT = 1
C$OMP FLUSH ICOUNT
C$OMP SECTION
1000 CONTINUE
C$OMP FLUSH ICOUNT
      IF(ICOUNT .LT. 1) GO TO 1000
      B = A + C
      ICOUNT = 2
C$OMP FLUSH ICOUNT
C$OMP SECTION
2000  CONTINUE
C$OMP FLUSH ICOUNT
       IF(ICOUNT .LT. 2) GO TO 2000
       C = B + A
C$OMP END PARALLEL SECTIONS
```

- In this example, we choose the assignments to occur in the order A, B, C.
  - ◆ ICOUNT forces this order.
  - ◆ FLUSH so each thread sees updates to ICOUNT - NOTE: you need the flush on each read and each write.

Race Conditions

```
C$OMP PARALLEL SHARED (X)
C$OMP& PRIVATE(TMP)
      ID = OMP_GET_THREAD_NUM()
C$OMP DO REDUCTION(+:X)
      DO 100 I=1,100
         TMP = WORK(I)
         X = X + TMP
100  CONTINUE
C$OMP END DO NOWAIT
      Y(ID) = WORK(X, ID)
C$OMP END PARALLEL
```

- The result varies unpredictably because the value of X isn't dependable until the barrier at the end of the do loop.
- Wrong answers produced without warning!
- Solution: Be careful when you use NOWAIT.

Race Conditions

```
    REAL TMP, X
C$OMP PARALLEL DO REDUCTION(+:X)
      DO 100 I=1,100
          TMP = WORK(I)
          X = X + TMP
100  CONTINUE
C$OMP END DO
       Y(ID) = WORK(X, ID)
C$OMP END PARALLEL
```

- The result varies unpredictably because access to shared variable TMP is not protected.
- Wrong answers produced without warning!
- The user probably wanted to make TMP private.

I lost an afternoon to this bug last year. After spinning my wheels and insisting there was a bug in KAI's compilers, the KAI tool Assure found the problem immediately!

Exercise 4:
Race conditions and the “pi” program

- Return to your “pi” program and this time, drop the private clause on x. In other words, let all threads use the same global variable for x.
  - Does your program still work?
  - Run it many times and see what happens to the answer.
  - Change the number of threads. Does the answer change?

Deadlock

```
      CALL OMP_INIT_LOCK (LCKA)
      CALL OMP_INIT_LOCK (LCKB)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
      CALL OMP_SET_LOCK(LCKA)
      CALL OMP_SET_LOCK(LCKB)
          CALL USE_A_and_B (RES)
      CALL OMP_UNSET_LOCK(LCKB)
      CALL OMP_UNSET_LOCK(LCKA)
C$OMP SECTION
      CALL OMP_SET_LOCK(LCKB)
      CALL OMP_SET_LOCK(LCKA)
          CALL USE_B_and_A (RES)
      CALL OMP_UNSET_LOCK(LCKA)
      CALL OMP_UNSET_LOCK(LCKB)
C$OMP END SECTIONS
```

- This shows a race condition and a deadlock.
- If A is locked by one thread and B by another, you have deadlock.
- If the same thread gets both locks, you get a race condition - i.e. different behavior depending on detailed interleaving of the thread.
- Avoid nesting different locks.

Deadlock

```
      CALL OMP_INIT_LOCK (LCKA)
C$OMP PARALLEL SECTIONS
C$OMP SECTION
      CALL OMP_SET_LOCK(LCKA)
      IVAL = DOWORK()
      IF (IVAL .EQ. TOL) THEN
            CALL OMP_UNSET_LOCK (LCKA)
      ELSE
            CALL ERROR (IVAL)
      ENDIF
C$OMP SECTION
      CALL OMP_SET_LOCK(LCKA)
      CALL USE_B_and_A (RES)
      CALL OMP_UNSET_LOCK(LCKA)
C$OMP END SECTIONS
```

- **This shows a race condition and a deadlock.**
- **If A is locked in the first section and the if statement branches around the unset lock, threads running the other sections deadlock waiting for the lock to be released.**
- **Make sure you release your locks.**

OpenMP death-traps

- ◆Are you using threadsafe libraries?
- ◆I/O inside a parallel region can interleave unpredictably.
- ◆Make sure you understand what your constructors are doing with private objects.
- ◆Private variables can mask globals.
- ◆Understand when shared memory is coherent. When in doubt, use FLUSH.
- ◆ NOWAIT removes implied barriers.

Navigating through the Danger Zones

- Option 1: Analyze your code to make sure every semantically permitted interleaving of the threads yields the correct results.
  - This can be prohibitively difficult due to the explosion of possible interleavings.
  - Tools like KAI’s Assure can help.

Navigating through the Danger Zones

- Option 2: Write SMP code that is portable and equivalent to the sequential form.
  - Use a safe subset of OpenMP.
  - Follow a set of “rules” for Sequential Equivalence.

Portable Sequential Equivalence

- What is Portable Sequential Equivalence (PSE)?
  - A program is sequentially equivalent if its results are the same with one thread and many threads.
  - For a program to be portable (i.e. runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored.

Portable Sequential Equivalence

- Advantages of PSE
  - A PSE program can run on a wide range of hardware and with different compilers - minimizes software development costs.
  - A PSE program can be tested and debugged in serial mode with off the shelf tools - even if they don’t support OpenMP.

2 Forms of Sequential Equivalence

- Two forms of Sequential equivalence based on what you mean by the phrase “equivalent to the single threaded execution”:
  - Strong SE: bitwise identical results.
  - Weak SE: equivalent mathematically but due to quirks of floating point arithmetic, not bitwise identical.

Strong Sequential Equivalence: rules

- Control data scope with the base language
  - Avoid the data scope clauses.
  - Only use private for scratch variables local to a block (eg. temporaries or loop control variables) whose global initialization don’t matter.
- Locate all cases where a shared variable can be written by multiple threads.
  - The access to the variable must be protected.
  - If multiple threads combine results into a single value, enforce sequential order.
  - Do not use the reduction clause.

Strong Sequential Equivalence: example

```
C$OMP PARALLEL  PRIVATE(I, TMP)

C$OMP DO  ORDERED
       DO 100 I=1,NDIM
            TMP =ALG_KERNEL(I)
C$OMP ORDERED
             CALL COMBINE (TMP, RES)
C$OMP END ORDERED
100   CONTINUE

C$OMP END PARALLEL
```

- **Everything is shared except I and TMP. These can be private since they are not initialized and they are unused outside the loop.**
- **The summation into RES occurs in the sequential order so the result from the program is bitwise compatible with the sequential program.**
- **Problem: Can be inefficient if threads finish in an order that's greatly different from the sequential order.**

Weak Sequential equivalence

- For weak sequential equivalence only mathematically valid constraints are enforced.
    - – Floating point arithmetic is not associative and not commutative.
    - – In most cases, no particular grouping of floating point operations is mathematically preferred so why take a performance hit by forcing the sequential order?
  - ◆ In most cases, if you need a particular grouping of floating point operations, you have a bad algorithm.
- How do you write a program that is portable and satisfies weak sequential equivalence?
  - ◆ Follow the same rules as the strong case, but relax sequential ordering constraints.

Weak equivalence: example

```
C$OMP PARALLEL  PRIVATE(I, TMP)
C$OMP DO
      DO 100 I=1,NDIM
          TMP =ALG_KERNEL(I)
C$OMP CRITICAL
           CALL COMBINE (TMP, RES)
C$OMP END CRITICAL
100   CONTINUE

C$OMP END PARALLEL
```

- **The summation into RES occurs one thread at a time, but in any order so the result is not bitwise compatible with the sequential program.**
- **Much more efficient, but some users get upset when low order bits vary between program runs.**

Sequential Equivalence isn’t a Silver Bullet

```
C$OMP PARALLEL
C$OMP& PRIVATE(I, ID, TMP, RVAL)
      ID = OMP_GET_THREAD_NUM()
       N  = OMP_GET_NUM_THREADS()
      RVAL = RAND ( ID )
C$OMP DO
      DO 100 I=1,NDIM
           RVAL = RAND (RVAL)
           TMP =RAND_ALG_KERNEL(RVAL)
C$OMP CRITICAL
            CALL COMBINE (TMP, RES)
C$OMP END CRITICAL
100   CONTINUE
C$OMP END PARALLEL
```

- **This program follows the weak PSE rules, but its still wrong.**
- **In this example, RAND() may not be thread safe. Even if it is, the pseudo-random sequences might overlap thereby throwing off the basic statistics.**

Conclusion

- OpenMP is:
  - A great way to write fast executing code.
  - Your gateway to special, painful errors.
- You can save yourself grief if you consider the possible danger zones as you write your OpenMP programs.
- Tools and/or a discipline of writing portable sequentially equivalent programs can help.

SC'99 Tutorial: Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- Automatic Parallelism and Tools support.
- OpenMP case studies.
  - Including performance tuning.
- Common bugs in OpenMP programs
  - and how to avoid them.
- ➡ The future of OpenMP

OpenMP Futures: The ARB

- The future of OpenMP is in the hands of the OpenMP Architecture Review Board (the ARB)
    - Intel, KAI, IBM, HP, Compaq, Sun, SGI, DOE ASCI
- The ARB resolves interpretation issues and manages the evolution of new OpenMP API's.
- Membership in the ARB is Open to any organization with a stake in OpenMP.
    - Research organization (e.g. DOE ASCI)
    - Hardware vendors (e.g. Intel or HP)
    - Software vendors (e.g. KAI)

ARB: What do we do

- We meet every two to four weeks and work on:
  - Interpretation - resolve ambiguities in the specifications.
  - Answer questions - each member organization takes a one month shift answering questions to the ARB.
  - Develop new specifications - We usually have a specification project underway .
  - Whatever else it takes to further OpenMP's impact - press releases, tutorials, test suites, etc.
- Meetings are held by email and over the phone so travel costs are nil.

Goals of the ARB: Part 1

- To produce API specifications that let programmers write portable, efficient, and well understood parallel programs for shared memory systems.
- To produce specifications that can be readily implemented in robust commercial products.
  - i.e. we want to standardize common or well understood practices, not chart new research agendas.
- To whatever extent makes sense, deliver consistency between programming languages.
  - The specification should map cleanly and predictably between C, Fortran, and C++.

Goals of the ARB: Part 2.

- We want OpenMP to be just large enough to express important, control-parallel, shared memory programs -- but no larger.
  - OpenMP needs to stay "lean and mean".
- Legal programs under an older version of an OpenMP specification should continue to be legal under newer specifications.
- To whatever extent possible, produce specifications that are sequentially consistent.
  - If sequential consistency is violated, there should be documented reasons for doing so.

The Future of OpenMP

- OpenMP is an evolving standard. We will see to it that it is well matched to the changing needs of the shard memory programming community.
- Here's what's coming in the future:
  - OpenMP 1.1 for Fortran:
    - This is a cleaned up version of 1.0 where we have fixed typos and merged the interpretations into the specification.
    - Status: We are in the final proofreading stage. Release: this fall.
  - OpenMP 2.0 for Fortran:
    - This is a major update of OpenMP for Fortran.
    - Status. Under active development. Done sometime in 2000.

OpenMP version 2.0:
Time line.

- Began gathering feedback winter 1999.
- Started regular meetings to create the standard, late spring 1999.
- Freeze list of features to consider, end of September, 1999.
- Completion date? Probably sometime next year but we will not rush it. Its better to be a little late than to do a poor job.

OpenMP 2.0: Goals*

- In addition to the general goals laid out by the ARB, we have some specific goals for OpenMP 2.0:
  - We want to support the Fortran95 language and the programming practices Fortran95 programmers typically use.
  - Fortran77 compilers must be able to conform to OpenMP 2.0.
  - We want to extend the range of applications that can be parallelized with OpenMP.

***This is an unofficial set of goals. We've discussed them, but a final vote has not been taken.**

OpenMP 2.0:
Some key items on "the list"

- Define threadprivate module data.
  - Our number one request from Fortran90 users.
- Define worksharing constructs for Fortran90 array expressions.
- Allow arrays in reductions.
- Cleanup the language
  - allow comments on a directive
  - reprivatization of private variables
  - provide a module defining runtime library interfaces.
  - … and many more subtle changes

OpenMP 2.0:
Some key items dropped from “the list”

- Parallel I/O.
  - This would be a huge addition to the language … violates the “lean and mean” goal.
- Explicit thread groups.
  - Violates the “lean and mean” rule, but also, how to implement and use this construct is a research question.
- Condition variable synchronization.
  - violates the sequential consistency goal. Makes it too easy to write programs that dead-lock under sequential readings.

Summary: OpenMP Benefits*

- ***Get more performance*** from applications running on multiprocessor workstations
- ***Get software to market sooner*** using a simplified programming model
- ***Reduce support costs*** by developing for multiple platforms with a single source code

Learn more at www.openmp.org

*Disclaimer: these benefits depend upon individual circumstances or system configurations and might not always be available.

Extra Slides
A series of parallel pi programs

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=1;i<= num_steps; i++){
                    x = (i-0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

Parallel Pi Program

- Let’s speed up the program with multiple threads.
- Consider the Win32 threads library:
  - Thread management and interaction is explicit.
  - Programmer has full control over the threads

Solution: 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!

Solution: Keep it simple

Threads libraries:

- Pro: Programmer has control over everything
- Con: Programmer must control everything

Full control $\Rightarrow$ Increased complexity $\rightarrow$ Programmers scared away

Sometimes a simple evolutionary approach is better

OpenMP PI Program:
Parallel Region example (SPMD Program)

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

SPMD Programs:

Each thread runs the same code with the thread ID selecting any thread specific behavior.

OpenMP PI Program:

Work sharing construct

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

OpenMP PI Program:
private clause and a critical section

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

Note: We didn’t need to create an array to hold local sums or clutter the code with explicit declarations of “x” and “sum”.

OpenMP PI Program :
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 reduction(+:sum) private(x)
            for (i=1;i<= num_steps; i++){
                       x = (i-0.5)*step;
                       sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP adds 2 to 4 lines of code

Reference Material on OpenMP

OpenMP Homepage www.openmp.org: The primary source of in formation about OpenMP and its development.

Books on OpenMP: Several books are currently being written on the subject but are not yet available by the time of this writing.

Research papers: There is also an increasing number of papers that discuss experiences, performance, proposed extensions etc. of OpenMP. Two examples of such papers are

- Transparent adaptive parallelism on NOWs using OpenMP; Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel; Proceedings of the 7th ACM SIGPLAN Symposium on Principles and practice of parallel programming , 1999, Pages 96 -106

- Parallel Programming with Message Passing and Directives; Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini; SIAM News, Volume 32, No 9, Nov. 1999.