


Shared Memory Programming with OpenMP

Stéphane Ethier
Computational Plasma Physics Group
Princeton Plasma Physics Laboratory

Picasso program tutorial
March 9, 2004

Outline

- Introduction 
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Cache contention effects
- Final remarks

Why parallel computing?

- The performance of a serial code is limited by the processor speed and the rate at which instructions and data are transmitted to it.
 - In current computers, memory access speed is the limiting factor.
 - To minimize the impact of that slow memory, several levels of fast but expensive caches are used.
- But we want to decrease our time-to-solution or run bigger problem sizes within the same time scale!
- Solution: use several processors to work on different parts of the problem at the same time.
 - Although this is easier said than done, in general...

Historical remarks

- The idea of parallel programming has been around for a long time.
 - Flynn's taxonomy in use since 1966 (SISD, SIMD, MISD, MIMD)
- Shared memory multi-processor computers were common in the early 90's (CRAY-XMP, SGI Challenge).
 - At the time, each vendor would supply its own similar, directive-based, Fortran programming extension.
 - The user would insert directives to indicate which loops to parallelize.
 - The compiler was responsible for parallelizing those loops across the processors.
 - Some compilers would detect the parallelism in a loop and automatically parallelize that loop (earliest example is CRAY autotasking).

What are directives?

- In C or C++, preprocessor statements ARE directives. They “direct” the preprocessing stage.
- Parallelization directives tell the compiler to add some machine code so that the next set of instructions will be distributed to several processors and run in parallel.
- In FORTRAN, directives are special purpose comments inserted right before the loop or region to parallelize.

C:

```
#pragma omp parallel for  
for (idx=n; idx; idx--) {  
    a[idx] = b[idx] + c[idx];  
}
```

Fortran:

```
!$omp parallel do  
do idx=1,n  
    a(idx) = b(idx) + c(idx)  
enddo
```

Why directives and auto-parallelization?

Hardware



SHORT LIFE CYCLE

Scientific software



LONG LIFE CYCLE

- Scientist do not want to rewrite their codes every 2 or 3 years because of new hardware or new message passing implementation.
- In most cases, software is the real capital investment.
- Directives allow an “incremental” parallelization, which means that you can start from your serial code and make it parallel without major restructuring. Using Message Passing cannot make the same claim...
- Auto-parallelization is really nice but does not generally give good results, although it can be used as an indicator.

Towards standardization

- At the beginning, the vendor’s directives had very similar functionality but were non-compatible and were diverging.
- A first attempt at a standard was the draft for ANSI X3H5 in 1994, but it was never adopted.
 - The distributed memory machines were becoming popular and the Message Passing Interface standard was getting a lot of attention.
- In the spring of 1997, the OpenMP standard specification took over where the ANSI X3H5 had left off, due to a renewed interest in shared-memory architecture.
 - Silicon Graphics had introduced its new ccNUMA Origin 2000 computer, which could scale to a much larger number of processors than the previous shared-memory machines.

Partners in the OpenMP specification

OpenMP Architecture Review Board

- Compaq / Digital
- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Kuck & Associates, Inc. (KAI)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc.
- U.S. Department of Energy ASCI program

Endorsing software vendors:

- Absoft Corporation
- Edinburgh Portable Compilers
- GENIAS Software GmbH
- Myrias Computer Technologies, Inc.
- The Portland Group, Inc. (PGI)

Endorsing application developers:

- ADINA R&D, Inc.
- ANSYS, Inc.
- Dash Associates
- Fluent, Inc.
- ILOG CPLEX Division
- Livermore Software Technology Corporation (LSTC)
- MECALOG SARL
- Oxford Molecular Group PLC
- The Numerical Algorithms Group Ltd.(NAG)

Release History

- October 1997: Fortran version 1.0
- Late 1998: C/C++ version 1.0
- June 2000: Fortran version 2.0
- April 2002: C/C++ version 2.0

*Source: <http://www.openmp.org>

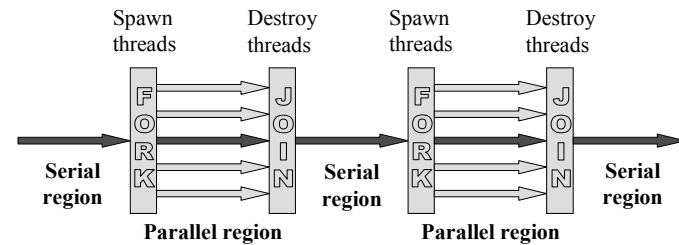


Outline

- Introduction
- Shared memory parallel model ←
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Cache contention effects
- Final remarks

Shared memory parallelism

- Multi-threaded parallelism (parallelism-on-demand)
- Fork-and-Join Model (although we say “spawn” for threads and “fork” for processes).



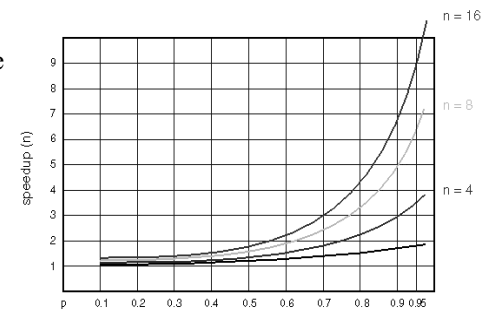
Process and thread: what's the difference?

- You need an existing process to create a thread.
- Each process has at least one thread of execution.
- A process has its own virtual memory space that cannot be accessed by other processes running on the same or on a different processor.
- All threads created by a process share the virtual address space of that process. They read and write to the same address space in memory. They also share the same process and user ids, file descriptors, and signal handlers. However, they have their own program counter value and stack pointer, and can run independently on several processors.

What about Amdahl's law?

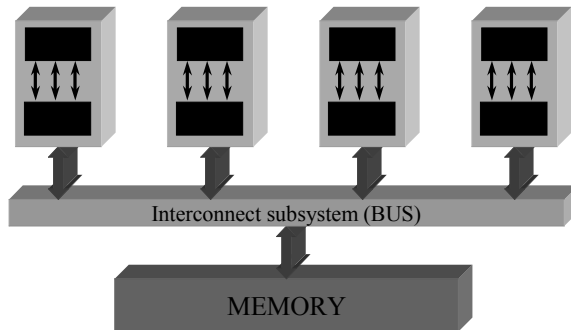
- $\text{Speedup}(n) = t_1/t_n = 1/[(p/n) + (1-p)]$ where n is the number of processors and p the fraction of parallel work

- The goal is to minimize the time spent in the serial regions.



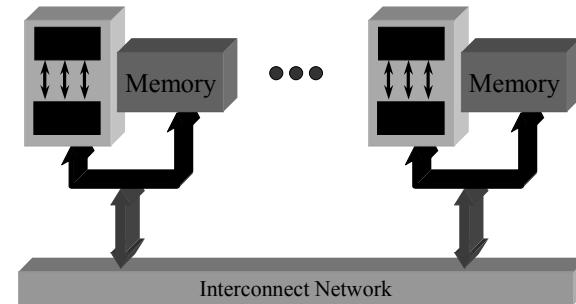
Shared memory architectures: SMP (or UMA)

- SMP – Symmetric Multi-Processing, UMA – Uniform Memory Access.
- All processors have equal (symmetric) access to memory.
- All processors share the same bus to access the memory.



Shared memory architectures: ccNUMA

- ccNUMA – cache-coherent Non-Uniform Memory Access
- Example of ccNUMA machine: SGI Origin series




Scalability of shared memory

- Shared memory parallelism with a flat or uniform memory model, SMP, does not scale to large number of processors because of limited memory bandwidth and the need to maintain cache coherency,:
 - done via broadcasting of all transactions over the shared bus (snooping).
- Non-uniform memory access (NUMA) pushes the scalability to a higher number of processors by using directory-based cache coherency method:
 - Each processor has a special cache (directory) to keep a list of which processors have the same cache lines. If change is made to the cache line, only the processors in the list need to know about it.

How data move from memory to cpu

- The memory structure is complex
 - Multiple-levels (increasing)
 - Caches are invaluable, essential and difficult
- Some difficulties with data access
 - Single processor effects
 - Cache misses and thrashing
 - TLB misses (TLB = Table Look-aside Buffer)
 - Multi-processor effects
 - False Sharing
 - Required synchronizations
- NUMA Systems
 - Data allocation and distribution
 - Page misses and migration

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description 
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Cache contention effects
- Final remarks

Goals of OpenMP

- Provide a standard among a variety of shared memory architectures/platforms
- Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism.
 - Coarse-grain = domain decomposition.
 - Fine-grain = loop-level parallelism.
- Supports Fortran (77, 90, and 95), C, and C++
- Public forum for API and membership

What is OpenMP?

- OpenMP Is:
 - An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
 - Comprised of three primary API components:
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
 - Portable:
 - The API is specified for C/C++ and Fortran
 - Multiple platforms have been implemented including most Unix platforms and Windows NT
 - Standardized:
 - Jointly defined and endorsed by a group of major computer hardware and software vendors
 - Expected to become an ANSI standard later?

What it is not...

- OpenMP Is Not:
 - Meant for distributed memory parallel systems (by itself)
 - Necessarily implemented identically by all vendors
 - Guaranteed to make the most efficient use of shared memory (currently there are no data locality constructs)

Example of OpenMP code structure

In FORTRAN:

```
PROGRAM HELLO
  INTEGER VAR1, VAR2, VAR3
  Serial code . . .
  Beginning of parallel section. Fork a team of threads.
  Specify variable scoping
  !$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
  Parallel section executed by all threads . . .
  All threads join master thread and disband
  !$OMP END PARALLEL
  Resume serial code . . .
END
```

Example of code structure in C

In C:

```
#include <omp.h>
main () {
  int var1, var2, var3;
  Serial code . . .
  Beginning of parallel section. Fork a team of threads.
  Specify variable scoping
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    Parallel section executed by all threads . . .
    All threads join master thread and disband
  }
  Resume serial code . . .
}
```

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives ←
 - Runtime library routines
 - Environment variables
- Cache contention effects
- Final remarks

Directives format in Fortran

```
!$OMP DIRECTIVE-NAME [CLAUSE, ...]
```

sentinel directive-name [clause...]

- All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are: **!\$OMP**, **C\$OMP**, ***\$OMP**
- Just use **!\$OMP** and you will be fine...
- The sentinel must be followed by a valid directive name.
- Clauses are optional and can be in any order, and repeated as necessary unless otherwise restricted.
- All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line

Fortran fixed form source

- Fixed Form Source:
 - !\$OMP C\$OMP *\$OMP are accepted sentinels and must start in column 1
 - All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line
 - Initial directive lines must have a space/zero in column 6.
 - Continuation lines must have a non-space/zero in column 6.

The following formats are equivalent:

```
!234567
!$OMP PARALLEL DO SHARED(A,B,C)
C$OMP PARALLEL DO
C$OMP+SHARED(A,B,C)
```

Fortran free form source

- Free Form Source:
 - !\$OMP is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
 - All Fortran free form rules for line length, white space, continuation and comment columns apply for the entire directive line
 - Initial directive lines must have a space after the sentinel.
 - Continuation lines must have an ampersand as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

```
!23456789
!$OMP PARALLEL DO &
!$OMP SHARED(A,B,C)
!$OMP PARALLEL &
!$OMP&DO SHARED(A,B,C)
```

General rules for Fortran interface

- Comments cannot appear on the same line as a directive if using OpenMP version 1.0 or 1.1. This is permitted in v.2.0
- Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.
 - SGI IRIX: f90 -mp
 - IBM AIX: xlf90 -qsmp=omp
- Several Fortran OpenMP directives come in pairs and have the form:

```
!$OMP directive
[ structured block of code ]
!$OMP end directive
```

C / C++ Directives Format

```
#pragma omp directive-name [clause,...] newline
```

- **#pragma omp**
 - Required for all OpenMP C/C++ directives.
- **directive-name**
 - A valid OpenMP directive. Must appear after the pragma and before any clauses.
- **[clause, ...]**
 - This is optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- **newline**
 - Required. Precedes the structured block which is enclosed by this directive.

General rules for C/C++ format

- Directives follow conventions of the C/C++ standards for compiler directives.
- Case sensitive.
- Only one directive-name may be specified per directive except in special cases (true with Fortran also).
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

Conditional compilation: _OPENMP

- All OpenMP-compliant implementations define a macro named `_OPENMP` when the OpenMP compiler option is enabled ("`f90 -mp`" on IRIX and "`xlf90 -qsmp=omp`" on AIX).
- This macro can be used to include extra code at the preprocessing stage.
- Valid for both C and Fortran, although one can also use simply `!$` in version 2.0 for Fortran (see specification).

```
#ifdef _OPENMP
    iam = omp_get_thread_num() + index;
#endif
```

PARALLEL Region Construct

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
- When a thread reaches a `PARALLEL` directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

Fortran format of PARALLEL construct

```
#$OMP PARALLEL [ clauses ...
    PRIVATE (list)
    SHARED (list)
    DEFAULT (PRIVATE | SHARED | NONE)
    FIRSTPRIVATE (list)
    REDUCTION ({operator|intrinsic_procedure}: list)
    COPYIN (list)
    IF (scalar_logical_expression)
        NUM_THREADS (scalar_integer_expression)
    ]
    block
!OMP END PARALLEL
```


C/C++ format of parallel construct

```
#pragma omp parallel [ clauses ] new-line  
{ structured-block }
```

Where clauses are:

```
private(list)  
shared(list)  
default(shared | none)  
firstprivate(list)  
reduction(operator : variable-list)  
copyin(list)  
if(scalar_expression)  
num_threads(scalar_integer_expression)
```

Data scope attribute clauses

- An important consideration for OpenMP programming is the understanding and use of data scoping.
- Because OpenMP is based on the shared memory programming model, most variables are shared by default between the threads.
- Global variables include (shared by default):
 - Fortran: COMMON blocks, SAVE variables, MODULE variables
 - C: File scope variables, static
- Private variables include (private by default):
 - Loop index variables
 - Stack variables in subroutines called from parallel regions
 - Fortran: Automatic variables within a statement block

Data scope attributes clauses

- The clauses `private(list)`, `shared(list)`, `default` and `firstprivate(list)` allow the user to control the scope attributes of variables for the duration of the parallel region in which they appear. The variables are listed in brackets right after the clause.
- PRIVATE variables behave as follows:
 - A new object of the same type is declared once for each thread in the team
 - All references to the original object are replaced with references to the new object
 - Variables declared PRIVATE are uninitialized for each thread
- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list. Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

REDUCTION clause

- This clause performs a reduction on the variables that appear in *list*, with the *operator* or the intrinsic procedure specified.
- *Operator* is one of the following:
 - Fortran: +, *, -, .AND., .OR., .EQV., .NEQV.
 - C/C++: +, *, -, &, ^, |, &&, ||
- Intrinsic procedure can be specified only in Fortran and is one of the following: MAX, MIN, IAND, IOR, IEOR
- Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used.

COPYIN clause

- The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.
 - The THREADPRIVATE directive is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.
- List contains the names of variables to copy. In Fortran, the list can contain both the names of common blocks and named variables.
- The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

IF clause

- If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- Only a single IF clause can appear on the directive.

How many threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 1. If the NUM_THREADS clause appears after the directive name, the number of threads specified is used for that parallel region.
 2. Use of the `omp_set_num_threads()` library function
 3. Setting of the `OMP_NUM_THREADS` environment variable
 4. Implementation default
- The threads are numbered from 0 (master thread) to N-1
- By default, a program with multiple parallel regions will use the same number of threads to execute each region. This behavior can be changed to allow the run-time system to dynamically adjust the number of threads that are created for a given parallel section. The `num_threads` clause is an example of this.

Fortran example of PARALLEL construct

```
PROGRAM REDUCTION
  INTEGER tnumber,I,J,K,OMP_GET_THREAD_NUM
  I=0; J=1; K=5
  PRINT *, "Before Par Region: I=",I," J=", J," K=",K

  !$OMP PARALLEL PRIVATE(tnumber) REDUCTION(+:I)&
  !$OMP REDUCTION(*:J) REDUCTION(MAX:K)
    tnumber=OMP_GET_THREAD_NUM()
    I = I + tnumber
    J = J*tnumber
    K = MAX(K,tnumber)
    PRINT *, "Thread ",tnumber, "      I=",I," J=", J," K=",K
  !$OMP END PARALLEL

  PRINT *, ""
  print *, "Operator      +      *      MAX"
  PRINT *, "After Par Region: I=",I," J=", J," K=",K
END PROGRAM REDUCTION
```

C example of PARALLEL construct

```
#include <omp.h>
main () {
    int nthreads, tid; /* Fork a team of threads giving
                        them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0){
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
}
```

Work-Sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.
- Types of Work-Sharing Constructs:
 - **DO / for** - shares iterations of a loop across the team. Represents a type of "data parallelism".
 - **SECTIONS** - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".
 - **SINGLE** - serializes a section of code

Work-Sharing Constructs Restrictions

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.
- Successive work-sharing constructs must be encountered in the same order by all members of a team.
- New in Fortran OpenMP specifications 2.0, the **WORKSHARE** directive.

DO/for directive

- Purpose:
 - The **DO / for** directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.
- Restrictions:
 - The **DO** loop can not be a **DO WHILE** loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
 - Program correctness must not depend upon which thread executes a particular iteration.
 - It is illegal to branch out of a loop associated with a **DO/for** directive.

Format of DO construct

```
!$OMP DO [clause ...  
    SCHEDULE (type [,chunk])  
    ORDERED  
    PRIVATE (list)  
    FIRSTPRIVATE (list)  
    LASTPRIVATE (list)  
    SHARED (list)  
    REDUCTION (operator / intrinsic : list)  
    ]  
do_loop  
!$OMP END DO [ NOWAIT ]
```

(C/C++) Format of “for” construct

```
#pragma omp for [clause ...] newline  
{ for-loop }
```

Where clauses are:

```
    schedule (type [,chunk])  
    ordered  
    private(list)  
    firstprivate(list)  
    lastprivate(list)  
    shared(list)  
    reduction(operator : variable-list)  
    nowait
```

SCHEDULE clause

- Describes how iterations of the loop are divided among the threads in the team. For both C/C++ and Fortran.
- STATIC:
 - Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- DYNAMIC:
 - Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

SCHEDULE clause

- GUIDED:
 - The chunk size is exponentially reduced with each dispatched piece of the iteration space. The chunk size specifies the minimum number of iterations to dispatch each time.. The default chunk size is 1.
- RUNTIME:
 - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- The default schedule is implementation dependent. Implementation may also vary slightly in the way the various schedules are implemented.

DO/for directive clauses

- ORDERED:
 - Must be present when ORDERED directives are enclosed within the DO/for directive.
- LASTPRIVATE(*list*)
 - The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.
 - The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct. For example, the team member which executes the final iteration for a DO section, or the team member which does the last SECTION of a SECTIONS context performs the copy with its own values.

NOWAIT clause

- If specified, then the threads do not synchronize at the end of the parallel loop. Threads proceed directly to the next statements after the loop.
- In C/C++, must be in lowercase: `nowait`
- For Fortran, the END DO directive is optional at the end of the loop.

Fortran example DO directive

```
PROGRAM VEC_ADD_DO
  INTEGER N, CHUNKSIZE, CHUNK, I
  PARAMETER (N=1000)
  PARAMETER (CHUNKSIZE=100)
  REAL A(N), B(N), C(N)
  ! Some initializations
  DO I = 1, N
    A(I) = I * 1.0
    B(I) = A(I)
  ENDDO
  CHUNK = CHUNKSIZE
  !$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)
  !$OMP DO SCHEDULE(DYNAMIC,CHUNK)
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO
  !$OMP END DO NOWAIT
  !$OMP END PARALLEL
END
```

C/C++ example of “for” directive

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main()
{
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;

  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  } /* end of parallel section */
}
```

SECTIONS directive

- The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections will be executed by different threads.

Fortran format of SECTIONS construct

```
!$OMP SECTIONS [clause ...
    PRIVATE (list)
    FIRSTPRIVATE (list)
    LASTPRIVATE (list)
    REDUCTION (operator / intrinsic : list)
]
[!$OMP SECTION]
    block

[!$OMP SECTION]
    block ]
...
!$OMP END SECTIONS [ NOWAIT ]
```

(C/C++) Format of sections construct

```
#pragma omp sections [clause ...] newline
{
    [#pragma omp section newline]
    structured-block
    [#pragma omp section newline]
    structured-block ]
...
}
```

Where clauses are:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator : variable-list)
nowait
```

Fortran example SECTIONS directive

```
PROGRAM VEC_ADD_SECTIONS
  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N)
  ! Some initializations
  DO I = 1, N
    A(I) = I * 1.0
    B(I) = A(I)
  ENDDO
  !$OMP PARALLEL SHARED(A,B,C), PRIVATE(I)
  !$OMP SECTIONS
  !$OMP SECTION
    DO I = 1, N/2
      C(I) = A(I) + B(I)
    ENDDO
  !$OMP SECTION
    DO I = 1+N/2, N
      C(I) = A(I) + B(I)
    ENDDO
  !$OMP END SECTIONS NOWAIT
  !$OMP END PARALLEL
END
```

C/C++ example of “sections” directive

```
#include <omp.h>
#define N 1000
main()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    #pragma omp parallel shared(a,b,c) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N/2; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=N/2; i < N; i++)
                c[i] = a[i] + b[i];
        } /* end of sections */
    } /* end of parallel section */
}
```

SINGLE directive

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O).
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a nowait (C/C++) or NOWAIT (Fortran) clause is specified.
- Format:
 - Fortran: !\$OMP SINGLE [clause...] ... !\$OMP END SINGLE
 - C/C++: #pragma omp single [clause ...] newline
 - Clauses: private(list), firstprivate(list), nowait

Combined Parallel Work-Sharing Constructs: PARALLEL DO

- This is one of the simplest and most useful constructs for fine-grain parallelism.

```
PROGRAM VEC_ADD_DO
  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N)
  ! Some initializations
  !$OMP PARALLEL DO           !By default, the static schedule
  DO I = 1, N                 !will be used and the loop will
    A(I) = I * 1.0           !be divided in equal chunks
    B(I) = A(I)
  ENDDO                       ! No need to put the END DO directive here

  !$OMP PARALLEL DO SHARED(A,B,C) PRIVATE(I)
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO
END
```

Combined Parallel Work-Sharing Constructs: “parallel for”

```
#include <omp.h>
#define N 1000
main()
{
    int i;
    float a[N], b[N], c[N];

    /* Some parallel initialization */
    #pragma omp parallel for
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;

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

Synchronization Constructs

- Let two threads on two different processors both trying to increment a variable x at the same time (assume x is initially 0):

```
THREAD 1:
increment(x)
{ x = x + 1; }

THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)
```

```
THREAD 2:
increment(x)
{ x = x + 1; }

THREAD 2:
10 LOAD B, (x address)
20 ADD B, 1
30 STORE B, (x address)
```

One possible execution sequence:

- Thread 1 loads the value of x into register A.
 - Thread 2 loads the value of x into register B.
 - Thread 1 adds 1 to register A
 - Thread 2 adds 1 to register B
 - Thread 1 stores register A at location x
 - Thread 2 stores register B at location x
- The resultant value of x will be 1, not 2 as it should be.

Synchronization Constructs

- To avoid the situation shown on the previous slide, the increment of x must be synchronized between the two threads to insure that the correct result is produced.
- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads:
 - OMP MASTER
 - OMP CRITICAL
 - OMP BARRIER
 - OMP ATOMIC
 - OMP FLUSH
 - OMP ORDERED

Synchronization Constructs

- The C/C++ versions of the following directives is constructed by replacing `!$OMP` by `#pragma omp` and using lowercase.
- `!$OMP MASTER ... !$OMP END MASTER`
 - Specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. No implied barrier.
- `!$OMP CRITICAL [name]... !$OMP END CRITICAL`
 - Specifies a region of code that must be executed by only one thread at a time. The optional name enables multiple different CRITICAL regions to exist.
- `!$OMP BARRIER`
 - When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier. For C/C++, the smallest statement that contains a barrier must be a structured block.

Synchronization Constructs

- `!$OMP ATOMIC`
 - The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section. The directive applies only to a single, immediately following statement.
- `!$OMP FLUSH (list)`
 - Identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point. The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, note that the pointer itself is flushed, not the object it points to.
- `!$OMP ORDERED`
 - specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

THREADPRIVATE directive

- The THREADPRIVATE directive is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.
- The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.
- Format:
 - Fortran: !OMP THREADPRIVATE (/cb/, ...)
 - C/C++: #pragma omp threadprivate (list)

THREADPRIVATE directive

- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive.
- THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.
- Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined (although usually "static" in practice).

Fortran example of THREADPRIVATE

```
PROGRAM THREADPRIV
  INTEGER ALPHA(10), BETA(10), I
  COMMON /A/ ALPHA
  !OMP THREADPRIVATE(/A/)
  C Explicitly turn off dynamic threads
  CALL OMP_SET_DYNAMIC(.FALSE.)
  C First parallel region
  !OMP PARALLEL PRIVATE(BETA, I)
  DO I=1,10
    ALPHA(I) = I
    BETA(I) = I
  END DO
  !OMP END PARALLEL
  C Second parallel region
  !OMP PARALLEL
  PRINT *, 'ALPHA(3)=',ALPHA(3), ' BETA(3)=',BETA(3)
  !OMP END PARALLEL
END
```

C/C++ example of threadprivate construct

```
#include <omp.h>

int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)

main() {
  /* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);

  /* First parallel region */
  #pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++)
    alpha[i] = beta[i] = i;

  /* First parallel region */
  #pragma omp parallel
  printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);
}
```

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines ←
 - Environment variables
- Cache contention effects
- Final remarks

OpenMP library routines

- The OpenMP standard defines an API for library calls that perform a variety of functions:
 - Query the number of threads/processors, set number of threads to use
 - General purpose locking routines (semaphores)
 - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
- For C/C++, it may be necessary to specify the include file "omp.h".
- For the Lock routines/functions:
 - The lock variable must be accessed only through the locking routines
 - For Fortran, the lock variable should be of type integer and of a kind large enough to hold an address.
 - For C/C++, the lock variable must have type `omp_lock_t` or type `omp_nest_lock_t`, depending on the function being used.

OMP_SET_NUM_THREADS

- Sets the number of threads that will be used in the next parallel region.
- Format:
 - Fortran
 - SUBROUTINE OMP_SET_NUM_THREADS(*scalar_integer_expression*)
 - C/C++
 - `void omp_set_num_threads(int num_threads)`
- Notes & Restrictions:
- The dynamic threads mechanism modifies the effect of this routine.
 - Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
 - Disabled: specifies exact number of threads to use until next call to this routine.
- This routine can only be called from the serial portions of the code
- This call has precedence over the OMP_NUM_THREADS environment variable

OMP_GET_NUM_THREADS

- Purpose:
 - Returns the number of threads that are currently in the team executing the parallel region from which it is called.
- Format:
 - Fortran
 - INTEGER FUNCTION OMP_GET_NUM_THREADS()
 - C/C++
 - `int omp_get_num_threads(void)`
- Notes & Restrictions:
 - If this call is made from a serial portion of the program, or a nested parallel region that is serialized, it will return 1.
 - The default number of threads is implementation dependent.

OMP_GET_THREAD_NUM

- Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0
- Format:
 - Fortran
 - INTEGER FUNCTION OMP_GET_THREAD_NUM()
 - C/C++
 - int omp_get_thread_num(void)
- Notes & Restrictions:
 - If called from a nested parallel region, or a serial region, this function will return 0.

Example of omp_get_thread_num

CORRECT:

```
PROGRAM HELLO

  INTEGER TID, OMP_GET_THREAD_NUM

  !$OMP PARALLEL PRIVATE(TID)
    TID = OMP_GET_THREAD_NUM()
    PRINT *, 'Hello World from thread = ', TID
  ...
  !$OMP END PARALLEL

END
```

INCORRECT:

- TID must be PRIVATE

```
PROGRAM HELLO

  INTEGER TID, OMP_GET_THREAD_NUM

  !$OMP PARALLEL
    TID = OMP_GET_THREAD_NUM()
    PRINT *, 'Hello World from thread = ', TID
  ...
  !$OMP END PARALLEL

END
```

Other functions and subroutines

- OMP_GET_MAX_THREADS()
 - Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.
- OMP_GET_NUM_PROCS()
 - Returns the number of processors that are available to the program.
- OMP_IN_PARALLEL
 - May be called to determine if the section of code which is executing is parallel or not.

More functions...

- OMP_SET_DYNAMIC()
 - Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.
- OMP_GET_DYNAMIC()
 - Used to determine if dynamic thread adjustment is enabled or not.
- OMP_SET_NESTED()
 - Used to enable or disable nested parallelism.
- OMP_GET_NESTED
 - Used to determine if nested parallelism is enabled or not.


OpenMP v2.0 subroutines and functions

- **OMP_INIT_LOCK()**
 - This subroutine initializes a lock associated with the lock variable.
- **OMP_DESTROY_LOCK()**
 - This subroutine disassociates the given lock variable from any locks.
- **OMP_SET_LOCK()**
 - This subroutine forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.
- **OMP_UNSET_LOCK()**
 - This subroutine releases the lock from the executing subroutine.
- **OMP_TEST_LOCK()**
 - This subroutine attempts to set a lock, but does not block if the lock is unavailable.

OpenMP v2.0 timing functions

- **OMP_GET_WTIME()**
 - This function returns a double precision value equal to the elapsed wallclock time in seconds since some arbitrary time in the past.
 - The times returned are “per-thread times”.
- **OMP_GET_WTICK()**
 - This function returns a double precision value equal to the number of seconds between successive clock ticks.
- Consults the OpenMP specification for more details on all the subroutines and functions:
 - <http://www.openmp.org/specs>

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines
 - Environment variables 
- Cache contention effects
- Final remarks


OpenMP environment variables

- OpenMP provides four environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase. The values assigned to them are not case sensitive.
- **OMP_SCHEDULE**
 - Applies only to DO, PARALLEL DO (Fortran) and for, parallel for (C/C++) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:
 - `setenv OMP_SCHEDULE "guided, 4"`
 - `setenv OMP_SCHEDULE "dynamic"`

OpenMP environment variables...

- **OMP_NUM_THREADS** (the most used...)
 - Sets the maximum number of threads to use during execution. For example: `setenv OMP_NUM_THREADS 8`
- **OMP_DYNAMIC**
 - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. For example: `setenv OMP_DYNAMIC TRUE`
- **OMP_NESTED** (rarely implemented on current systems)
 - Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example: `setenv OMP_NESTED TRUE`

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Cache contention effects 
- Final remarks

Performance issues: false sharing

```
subroutine sum85 (a,s,m,n)
  integer m, n, i, j
  real a(m,n), s(m)
!$omp parallel do private(i,j), shared(s,a)
  do i = 1, m
    s(i) = 0.0
    do j = 1, n
      s(i) = s(i) + a(i,j)
    enddo
  enddo
  return
end
```


- When “m” is small, cache contention leads to false sharing.
- All threads fight for the same cache line.

False sharing: how to avoid it

- The following fix is valid for a 128 kbyte cache line.
- Since a real is 4 bytes, we need to pad the s array with 32 values to clear the cache line size.

```
subroutine sum85 (a,s,m,n)
  integer m, n, i, j
  real a(m,n), s(32,m)
!$omp parallel do private(i,j), shared(s,a)
  do i = 1, m
    s(1,i) = 0.0
    do j = 1, n
      s(1,i) = s(1,i) + a(i,j)
    enddo
  enddo
  return
end
```

Outline

- Introduction
- Shared memory parallel model
- OpenMP
 - Description
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Cache contention effects
- Final remarks 

Final remarks

- If you want to get fancy, use a “look-ahead” approach to minimize the impact of serial sections. Give less work to the master thread and use “nowait” so that it can do some of the serial work while the other threads are still busy.
- Have a look at high-performance SMP libraries such as NAG (http://www.nag.co.uk/numeric/numerical_libraries.asp) and SuperLU (<http://acts.nersc.gov/superlu/main.html>) .
- Mixed-model MPI/OpenMP works well and is an easy way to upgrade your MPI code.

References

- OpenMP specifications: <http://www.openmp.org/specs/>
- OpenMP presentations: <http://www.openmp.org/presentations/>
- SGI techpubs <http://techpubs.sgi.com/> :
 - Book “**Origin 2000 and Onyx2 Performance Tuning and Optimization Guide**”
- <http://www.llnl.gov/computing/tutorials/openMP/>
- EWOMP03 <http://www.rz.rwth-aachen.de/ewomp03/>
- WOMPAT 2003
<http://www.eecg.toronto.edu/wompat2003/>
- Just type “openmp” in Google...