第四篇 并行程序设计

第十二章 并行程序设计基础

第十三章 共享存储系统并行编程

第十四章 分布存储系统并行编程

第十五章 并行程序设计环境与工具

共享存储编程环境

- 纯共享存储环境
  - SMP、PVP
  - UMA
  - 集中的、公共的共享存储器
  - 全局统一编址
  - ANSI X3H5、Pthreads、OpenMP等
- 虚拟共享存储环境
  - DSM
  - NUMA
  - 分布的局部存储器组成系统的全局共享虚拟存储器
  - 全局统一编址
  - Linda、Jiajia

共享存储并行编程的基本问题

- 任务划分
  - 域分解（数据并行）
  - 功能分析（控制并行）
- 任务调度
  - 静态调度（确定、非确定）
  - 动态调度
- 任务同步
  - 同步方式：锁、路障、事件、信号灯、管程等
- 任务通信
  - 读写共享变量

Shared Memory Programming Models

ANSI X3H5 Shared Memory Model

- Standardized 1993
- Never became a de facto standard
  however, many vendors borrowed concepts of X3H5
- X3H5 defines a conceptional programming model
- X3H5 defines 3 language bindings:
  C, Fortran 77, Fortran 90

http://pvs.informatik.uni-heidelberg.de/Teaching/PHP-SS00/HTML/img69.htm

Programming for High Performance - An Introduction © Thomas Ludwig, 2000 (71/146)

Shared Memory Programming Models

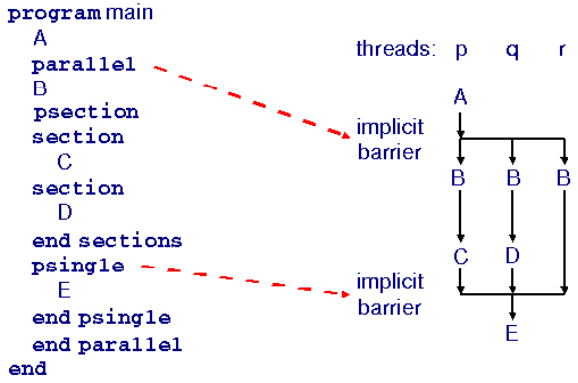
X3H5 Constructs for Parallelism

- X3H5 uses a set of specific language constructs
- No explicit definition of the number of threads that execute
- Programs start in sequential mode with a master thread
- Constructs are `parallel psection psingle pdo`

Programming for High Performance - An Introduction © Thomas Ludwig, 2000 (72/146)

Shared Memory Programming Models

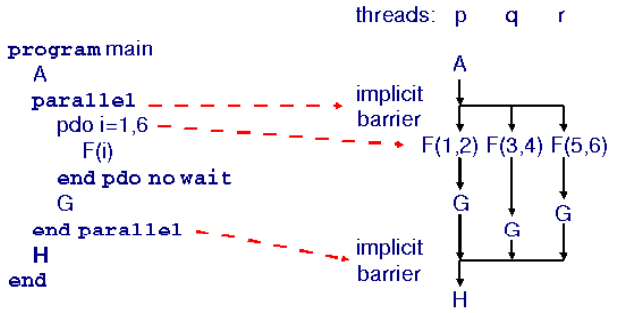
X3H5 Example 1



Programming for High Performance - An Introduction © Thomas Ludwig, 2000 (73/146)

Shared Memory Programming Models

X3H5 Example 2



Programming for High Performance - An Introduction © Thomas Ludwig, 2000 (74/146)

Shared Memory Programming Models

X3H5 Semantics

`parallel`

- a number of threads are started
- change to parallel execution mode
- further constructs are executed in parallel

Worksharing constructs

- `psection`: multiple code multiple data parallelism corresponds to code partitioning
- `psingle`: only one thread executes (e.g. input/output)
- `pdo`: single program multiple data parallelism

Programming for High Performance - An Introduction © Thomas Ludwig, 2000 (75/146)

X3H5 Semantics (2)

Worksharing constructs support load balancing

- e.g. with `pdo` threads execute indices according to availability

Implicit barriers

- `parallel`, `end parallel`, `end psections`, `end pdo`, `end psingle`
- enforces memory consistency

Explicit barriers

- `no wait`- construct; faster, more error prone

Threads Programming Model

- Manual thread creation
  - `pthread_create` (... (* myroutine) ...)
  - low abstraction level
- Synchronisation
  - Mutex variable (mutual exclusion)
  - Condition variable
- No high level abstraction constructs
  - no `parallel`, `psection`, `pdo` or similiar

The POSIX Thread Model

- IEEE standardisation 1995
- Slightly different implementations in different operating systems
- Similiar to threads in Solaris
- Designed for SMP systems with low number of processors
- Pure library approach
- No compiler support necessary/available

The OpenMP Standard

- By vendors like DEC, Intel, IBM, Potland Group, and others
- OpenMP initially for Fortran in 1997
- OpenMP has compiler directives, libraries, and environment variables
- OpenMP defines API for shared memory programming under Unix and WindowsNT
- Similiar to X3H5 but more constructs

Shared Memory Programming Models

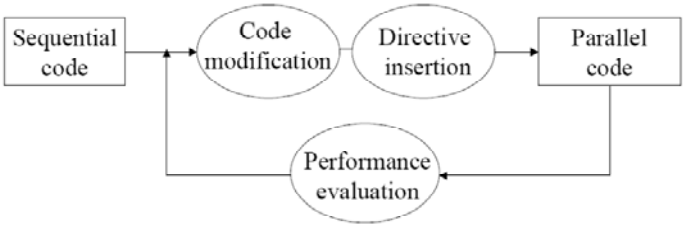
A Comparison of Concepts

	X3H5	pthreads	OpenMP
scalability	no	maybe	yes
Fortran binding	yes	no	yes
C binding	yes	yes	planned
high abstraction	yes	no	yes
performance oriented	no	no	yes
data parallelism	yes	no	yes
portability	yes	yes	yes
incremental parallelization	yes	no	yes
vendor support	no	UNIX/SMP	starting

Pthreads

- ⌘ POSIX standard shared-memory multithreading interface.
- ⌘ Not just for parallel programming, but for general multithreading programming.
- ⌘ Provides primitives for thread management and synchronization.

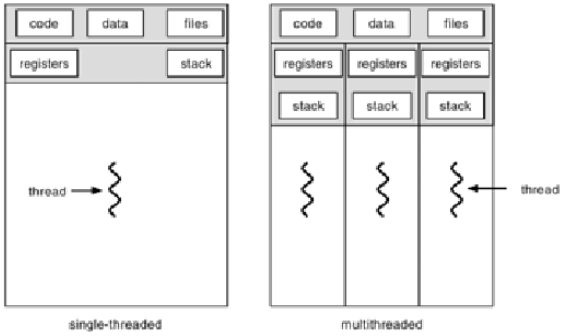
Incremental parallelization

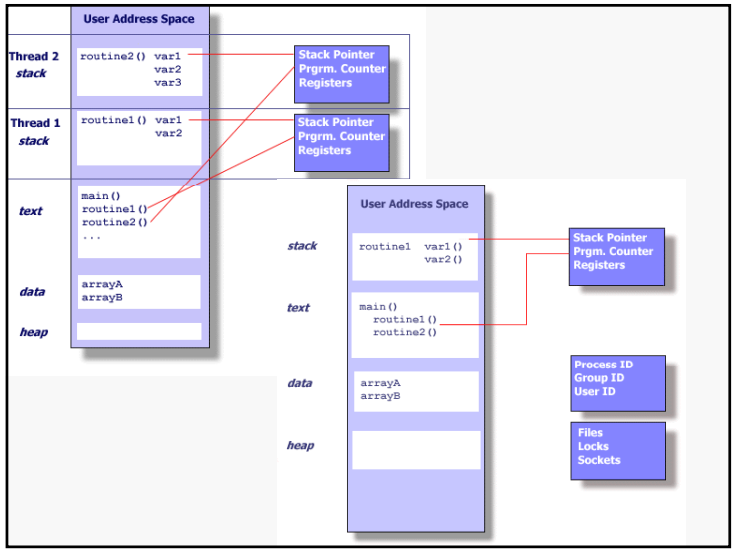


Work can be done incrementally!!!!!!!

Threads (lightweight processes)

- ⌘ A thread is basic unit of CPU utilization





Thread Creation

```
int pthread_create
  (pthread_t *new_id,
  const pthread_attr_t *attr,
  void *(*func) (void *),
  void *arg)
```

- new_id: thread's unique identifier
- attr: ignore for now
- func: function to be run in parallel
- arg: arguments for function func

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

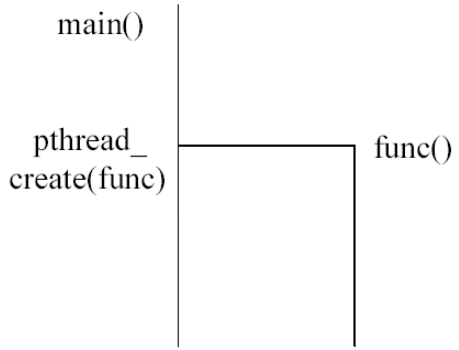
http://www.llnl.gov/computing/tutorials/pthreads/#Overview

Example of Thread Creation

```
void *func(void *arg) {
  int       *i=arg;
  .....
}

void main()
{
  int X;
  pthread_t       id;
  ....
  pthread_create(&id, NULL, func, &X);
  ...
}
```

Example of Thread Creation (contd.)



Thread Joining

```
int pthread_join(
pthread_t new_id,
void **status)
```

⌘ Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().

⌘ Status receives the return value or the value given as argument to pthread_exit().

Pthread Termination

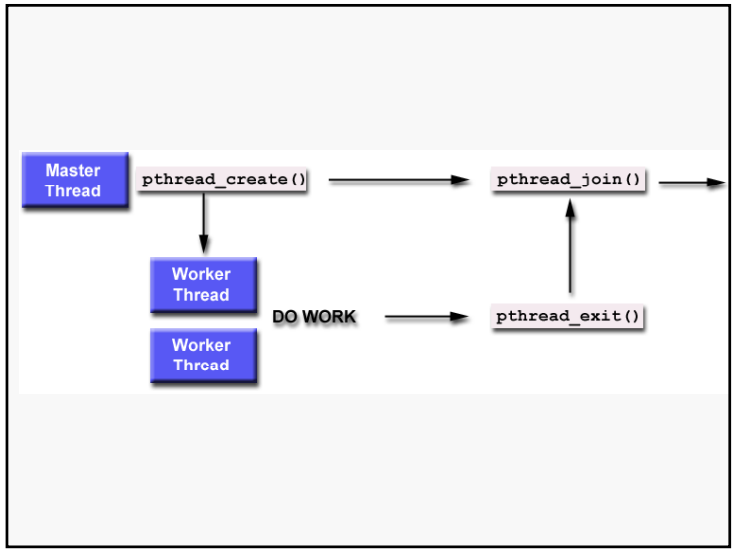
```
void pthread_exit(void *status)
```

⌘ Terminates the currently running thread.

⌘ Is implicit when the function called in pthread_create returns.

Thread Joining Example

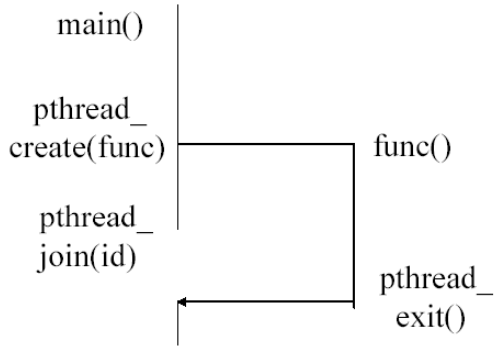
```
void *func(void *) { ..... }
pthread_t     id;  int X;
pthread_create(&id, NULL, func, &X);
.....
pthread_join(id, NULL);
.....
```



Matrix Multiplication

```
for( i=0; i<n; i++ )
  for( j=0; j<n; j++ ) {
        c[i][j] = 0.0;
        for( k=0; k<n; k++ )
              c[i][j] += a[i][k]*b[k][j];
  }
```

Example of Thread Creation (contd.)



Parallel Matrix Multiplication

- ⌘ All i- or j-iterations can be run in parallel.
- ⌘ If we have p processors, n/p rows to each processor.
- ⌘ Corresponds to partitioning i-loop.

Matrix Multiply: Parallel Part

```
void mmult(void* s)
{
  int slice = (int) s;
  int from = (slice*n)/p;
  int to = ((slice+1)*n)/p;
  for(i=from; i<to; i++)
        for(j=0; j<n; j++) {
                c[i][j] = 0.0;
                for(k=0; k<n; k++)
                        c[i][j] += a[i][k]*b[k][j];
  }
}
```

More about Pthreads?

⌘ Critical sections ---- mutex

⌘ Thread synchronization ---- condition variables

⌘ For complete information, many good references exist:
- Multithreaded Programming With Pthreads, Bil Lewis, Daniel J. Berg.
- Pthreads Programming, Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell, Jackie Farrell

Matrix Multiplication: Main

```
int main()
{
  pthread_t thrd[p];

  for( i=0; i<p; i++ )
    pthread_create(&thrd[i], NULL, mmult,(void*) i);

  for( i=0; i<p; i++ )
    pthread_join(thrd[i], NULL);
}
```

OpenMP编程简介

- **起源于ANSI X3H5标准**
- **简单、移植性好和可扩展**
- **工业标准**
  - **DEC、Intel、IBM、HP、Sun、SGI等公司支持**
  - **包括UNIX和NT等多种操作系统平台**
- **将已有的串行程序逐步并行化**
- **Fortran77、Fortran90、C、C++语言的实现规范已经完成，**
- **http://www.openmp.org/**

OpenMP: Supporters*

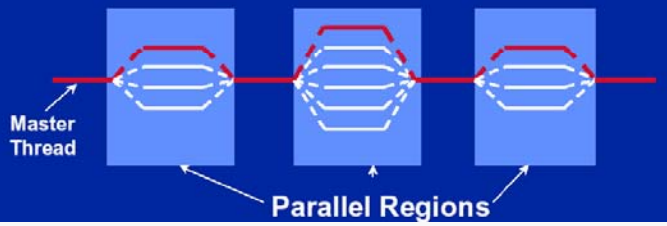
- Hardware vendors
  - Intel, HP, SGI, IBM, SUN, Compaq
- Software tools vendors
  - KAI, PGI, PSR, APR, Absoft
- Applications vendors
  - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, and many others

*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: 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
```

Not A structured block

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.

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`

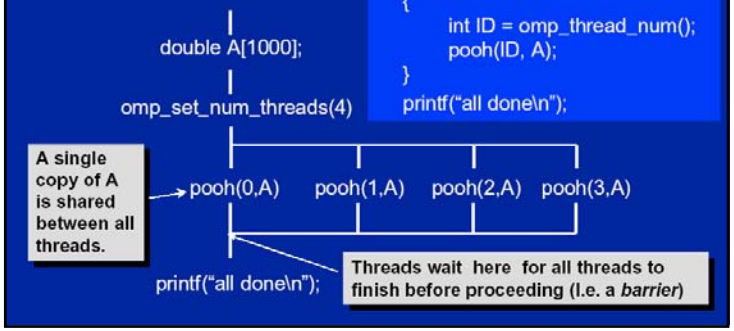
OpenMP: Contents

- 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

- 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");
```



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.

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.

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: 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.

Orphan制导语句

- 为了便于支持粗粒度的任务级并行，OpenMP 提供了Orphan制导语句
- Orphan制导语句是指那些在并行区域(Parallel Region,如PARALLEL)之外的制导语句
- 在OpenMP中提供了一种绑定规则使得这些Orphan制导语句与调用它们的并行区域产生联系，这样大大地增加了程序的模块性。X3H5 中不支持这一特点，所有的同步和控制语句都必须依次出现在并行区域内,无模块性。

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
```

Static or lexical extent of parallel region

Dynamic extent of parallel region includes *static* extent

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
```

Orphan directives can appear outside a parallel region

Data environment: Data scope

- SHARED - variable is shared by all processors
- PRIVATE - each processor has a private copy of a variable

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

- All CPUs have access to the same storage area for A, B, C and N,
- but each loop needs its own private value of the loop index I.

Default Storage Attributes

- Global variables are SHARED among threads
  - FORTRAN: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- Stack variables in sub-programs called from parallel regions are PRIVATE

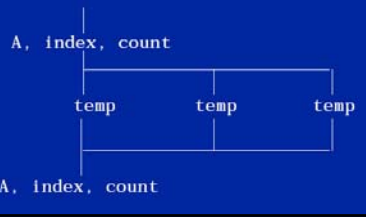
Storage Attributes

- Storage attributes can be changed using the following clauses:
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE
- The value of a PRIVATE variable 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)

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

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$OMP PARALLEL DO PRIVATE(IS)
do j=1,1000
 is=is+j !is is not initialized
end do
C$OMP END PARALLEL DO
print *,is !this is still the original is
```

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

Example

```
   program wrong3
   is=0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP* LASTPRIVATE(IS)
   do j=1,1000
        is=is+j !is is now 0
   end do
C$OMP END PARALLEL DO
   print *,is
```

```
Serial output
junior:~> a.out
    500500

Parallel output:
junior:~> setenv OMP_NUM_THREADS
    2
junior:~> a.out
    375250

junior:~> setenv OMP_NUM_THREADS
    4
junior:~> a.out
    218875
```

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: 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.

```
   program compute_pi
   integer n, i
   double precision w, x, sum, pi, f , a
c function to integrate
   f (a) = 4.d0 / (1.d0 + a*a)
   pr int *, 'Enter number of intervals: '
   read *,n
c calculate the interval size
   w = 1.0d0/n
   sum = 0.0d0
!$OMP PARALLEL DO PRIVATE(x), SHARED(w,n)
!$OMP& REDUCTION(+: sum)
   do i = 1, n
   x = w * (i - 0.5d0)
   sum = sum + f(x)
   end do
!$OMP END PARALLEL DO
   pi = w * sum
   pr int *, 'computed pi = ', pi
   stop
   end
```

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;
      }
}
```

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: 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

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: 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
```

求最大值

```
int i ; int max_num=-1;
#pragma omp parallel for
for (i=0; i<n; i++)
#progama omp critical
    if(ar[i]>max_num)
      max_num=ar[i];
```

求最大值

```
int i ; int max_num=-1;
for (i=0; i<n; i++)
    if(ar[i]>max_num)
      max_num=ar[i];
```

```
int i ; int max_num=-1;
#pragma omp parallel for
for (i=0; i<n; i++)
    if(ar[i]>max_num)
      max_num=ar[i];
```

```
#pragma omp critical [(name)]
```

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

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

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();
}
```

```
#pragma omp parallel for
for( int i = 0; i < 6; ++ i )
 Test( i );
```

```
<T:0> - 0
<T:1> - 3
<T:0> - 1
<T:1> - 4
<T:0> - 2
<T:1> - 5
```

```
#pragma omp parallel for ordered
for( int i = 0; i < 6; ++ i )
{
 #pragma omp ordered
 Test( i );
}
```

```
<T:0> - 0
<T:0> - 1
<T:0> - 2
<T:1> - 3
<T:1> - 4
<T:1> - 5
```

```
#pragma omp parallel for ordered
for( int i = 0; i < 6; ++ i )
{
        Test( i );
 #pragma omp ordered
        Test( 10 + i );
}
```

```
<T:0> - 0
<T:1> - 3
<T:0> - 10
<T:0> - 1
<T:0> - 11
<T:0> - 2
<T:0> - 12
<T:1> - 13
<T:1> - 4
<T:1> - 14
<T:1> - 5
<T:1> - 15
```

runtime)

guided 的 chunk 切割方法和 static、dynamic 不一樣；他會以「遞減」的數目，來分割出 chunk。而 chunk 的分配方式，則是和 dynamic 一樣是動態的分配。而遞減的方式，大約會以指數的方式遞減到指定的 chunk_size。
如果沒有指定 chunk_size 的話，chunk_size 會被設定為 1。

http://heresy.spaces.live.com/blog/

runtime

原則上，這不是一個方法。設定成 runtime 的話，OpenMP 會在執行到的時候，再由環境變數 OMP_SCHEDULE 來決定要使用的方法。

runtime)

OpenMP 會將 for 循环的所有 iteration 依序以指定 chunk_size 做切割成數個 chunk；然後再用 round-robin fashion 的方法將各個 chunk 指定給 thread 去執行。

如果沒有指定 chunk_size 的話，OpenMP 則會根據 thread 的數目做最平均的分配。

http://heresy.spaces.live.com/blog/

dynamic

和 static 時一樣，OpenMP 會將 for循环的所有 iteration 依序以指定 chunk_size 做切割成數個 chunk。但是 dynamic 時，chunk 的分配方法會是動態的；當 thread 執行完一個 chunk 後，他會在去找別的 chunk 來執行。

如果沒有指定 chunk_size 的話，chunk_size 會被設定為 1。

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: 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: 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: 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: Environment Variables

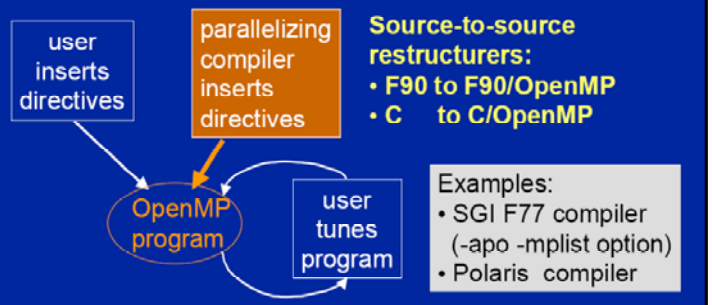
- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - OMP_SCHEDULE "schedule[, chunk_size]"
- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Can the program use a different number of threads in each parallel region?
  - OMP_DYNAMIC TRUE || FALSE
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - OMP_NESTED TRUE || FALSE

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);  }
}
```

Generating OpenMP Programs Automatically



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

Reduction recognition:

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

→

```
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

Data privatization:

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

→

```
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

Induction variable substitution:

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

   i2 = i2 + i
   A(i2) = …

ENDDO
```

→

```
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)

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

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.

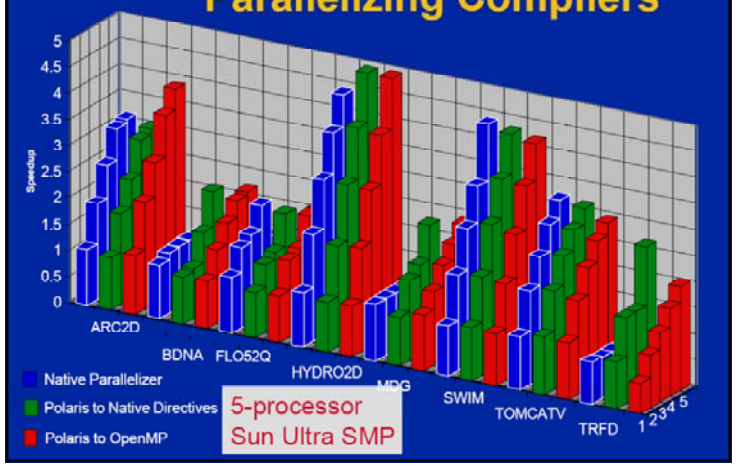
Compiler Listing

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



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)

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`)