OpenMP Guide

The purpose of this guide is to discuss a number of topics related to OpenMP. The topics include:

- Portability issues
- Discussion of thread scheduling and the effect of scheduling on timings
- Information on compiler invocation
- A collection of links to additional information
- Potential problems from the inmaturity of compilers

This guide is organized as a collection of articles. The titles are shown below. The collection of links is just what it says, a collection of web links to useful pages dealing with OpenMP. These include documentation, user groups, tutorials, and vendor information. The compiler invocation section discussed how OpenMP is invoked and controlled on various platforms.

The material included under "Examples and more obscure usages" is somewhat varied. It includes discussions on thread scheduling in do/for loops, some advanced threadprivate code, coarse grained parallelism not using do/for loops, and some examples that are included simply because they broke many early implementations of OpenMP.

The examples all have source code associated with them. Some of the articles have sections of the code imbedded. For consistency, there is a link to the complete source at the beginning of each article.

Collection of links about OpenMP

Main OpenMP web pages. Contains links to many other OpenMP sites.

OpenMP Architecture Review Board
    http://www.openmp.org/

Official OpenMP Specifications
    http://www.openmp.org/specs/

User Group

cOMPunity, a community of OpenMP researchers and developers in academia and industry
    http://www.compunity.org/

Good general link

Parallel Computing Group at La Laguna University
    http://nereida.deioc.ull.es/html/openmp.html

Tutorials

Tutorials and a lot of other good information from La Laguna University
    http://nereida.deioc.ull.es/html/openmpindex.html#tutorials

Nice short tutorial from NERSC
    http://hpcf.nersc.gov/training/tutorials/openmp/

LLNL
    http://www.llnl.gov/computing/tutorials/workshops/workshop/openMP/MAIN.html

Free implementations of OpenMP

Intone OpenMP C/C++ compiler
  http://odinmp.imit.kth.se/

Omni OpenMP Compiler Project
  http://phase.etl.go.jp/Omni/

OpenMP benchmarks

EPCC microbenchmarks
  http://www.epcc.ed.ac.uk/research/openmpbench/

EPCC microbenchmarks in Fortran90
  http://coherentcognition.com/~tkaiser/articles/openmp/bull/port.html

NAS Parallel Benchmarks
  http://phase.etl.go.jp/Omni/benchmarks/NPB/index.html

Vendor information

HP-KAP

Fortan
  http://nf.apac.edu.au/facilities/sc/compaq_mirror3/progtool/kapc/kapc.html

C
  http://nf.apac.edu.au/facilities/sc/compaq_mirror3/progtool/kapf/kapfu.htm

IBM AIX

Fortran
  http://www-3.ibm.com/software/awdtools/fortran/xlfortran/support/ Then click on Product information

C

http://www-3.ibm.com/software/awdtools/caix/support/ Then click on Product information

SUN

Fortran users Guide
ftp://docs-pdf.sun.com/806-7988/806-7988.pdf
Fortran Programmers Guide
ftp://docs-pdf.sun.com/805-4940/805-4940.pdf
For other Sun documents see
http://docs.sun.com/

Intel

Compiler pages
http://developer.intel.com/software/products/compilers
Fortran User guide
http://www.intel.com/software/products/compilers/techtopics/for_ug_lnx.htm
Fortran Reference
http://www.intel.com/software/products/compilers/techtopics/for_prg.htm
Intel C++ Compiler User's Guide
http://developer.intel.com/software/products/compilers/techtopics/c_ug_lnx.pdf
Intel C++ Compiler 7.0 for Linux* Getting Started Guide
ftp://download.intel.com/software/products/compilers/techtopics/C_Getting_Started_Guide1.pdf
For additional Intel documentation
Do an anonymous ftp to download.intel.com and look in the directory /software/products/compilers/techtopics

Compiler invocation, information and environment variables

Cray SV1

To Enable OpenMP in the compiler

Fortran

OpenMP is on by default to turn it **off** specify

```
f90 -xOMP myprogram.f90
```

Notes on environment variables:

OMP_DYNAMIC
: OMP_DYNAMIC is ignored. The dynamic adjustment of threads is always enabled and cannot be disabled.

OMP_NESTED
: OMP_NESTED is ignored. Nested parallelism is not supported.

OMP_SCHEDULE
: The default schedule type is DYNAMIC.

OMP_NUM_THREADS
: Caution, the value of the NCPUS environment variable overrides the value of the OMP_NUM_THREADS environment variable. That is, if NCPUS is set then it takes precedence over OMP_NUM_THREADS. If neither are defined then the default is 4 or the number of CPUs on the system, whichever is less.

NCPUS
: See OMP_NUM_THREADS

Other notes:

IBM AIX

To Enable OpenMP in the compiler

Fortran

Use the compilers xlf_r, xlf90_r, or xlf95_r and specify the -qsmp=omp compiler option. The "_r" extension in the compiler name supplies thread safe library routines.

C

Use the compilers xlc_r or cc_r and specify the -qsmp=omp compiler option. The "_r" extension in the compiler name supplies thread safe library routines.

You can also specify the schedule type for threads on the compile line using the syntax

```
xlf90_r myprogram.f -qsmp=omp,schedule=static
```

Where the schedule suboption takes subsuboptions.

- affinity[=n]
- dynamic[=n]
- guided[=n]
- runtime
- static[=n]
- threshold[=n]

For descriptions please see below or the Fortran User's Guide.

Notes on environment variables:

OMP_DYNAMIC
: Defalut value is TRUE

OMP_NESTED
: Defalut value is FALSE

OMP_SCHEDULE
: The default schedule type is STATIC.

OMP_NUM_THREADS
: Defalut value is the number of processors on a node

XLSMPOPTS
: Can be used instead of the normal OMP environment varaialbes. See below. If you specify both the XLSMPOPTS environment variable and an OpenMP environment variable, the OpenMP environment variable takes precedence.

Other notes:

If there is a routine say, my_fft, where my_fft does not contain OpenMP and it is called in a loop like:

```
!$omp parallel do
    do i=1,n
       call my_fft(x(i))
    enddo
```

Try compiling my_fft separately with OpenMP turned off and link it with the rest of your program. Compiling with OpenMP limits some optimizations. Compiling separately might improve performance.

From the IBM XLF Users guide: **The XLSMPOPTS Environment Variable**

The XLSMPOPTS environment variable allows you to specify options that affect SMP execution. You can declare XLSMPOPTS by using the following ksh command format:

```
XLSMPOPTS= runtime_option_name = option_setting
```

You can specify option names and settings in uppercase or lowercase. You can add blanks before and after the colons and equal signs to improve readability. However, if the XLSMPOPTS option string contains imbedded blanks, you must enclose the entire option string in double quotation marks ("). You can specify the following run-time options with the XLSMPOPTS environment variable:

schedule

Selects the scheduling type and chunk size to be used as the default at run time. The scheduling type that you specify will only be used for loops that were not already marked with a scheduling type at compilation time. Work is assigned to threads in a different manner, depending on the scheduling type and chunk size used. A brief description of the scheduling types and their influence on how work is assigned follows:

dynamic or guided
: The run-time library dynamically schedules parallel work for threads on a "first-come, first-do" basis. "Chunks" of the remaining work are assigned to available threads until all work has been assigned. Work is not assigned to threads that are asleep.

static
: Chunks of work are assigned to the threads in a "round-robin" fashion. Work is assigned to all threads, both active and asleep. The system must activate sleeping threads in order for them to complete their assigned work.

affinity
: The run-time library performs an initial division of the iterations into number_of_threads partitions. The number of iterations that these partitions contain is:
: CEILING(number_of_iterations / number_of_threads)
: These partitions are then assigned to each of the threads. It is these partitions that are then subdivided into chunks of iterations. If a thread is asleep, the threads that are active will complete their assigned partition of work.

Choosing chunking granularity is a tradeoff between overhead and load balancing. The syntax for this option is schedule=suboption, where the suboptions are defined as follows:

affinity[=n]
: As described previously, the iterations of a loop are initially divided into partitions, which are then preassigned to the threads. Each of these partitions is then further subdivided into chunks that contain n iterations. If you have not specified n, a chunk consists of

CEILING(number_of_iterations_remaining_in_local_partition / 2)
loop iterations. When a thread becomes available, it takes the next chunk from its preassigned partition. If there are no more chunks in that partition, the thread takes the next available chunk from a partition preassigned to another thread.

dynamic[=n]
: The iterations of a loop are divided into chunks that contain n iterations each. If you have not specified n,a chunk consists of
CEILING(number_of_iterations / number_of_threads)
iterations.

guided[=n]
: The iterations of a loop are divided into progressively smaller chunks until a minimum chunk size of n loop iterations is reached. If you have not specified n, the default value for n is 1 iteration. The first chunk contains
CEILING(number_of_iterations / number_of_threads)
iterations. Subsequent chunks consist of
CEILING(number_of_iterations_remaining / number_of_threads)
iterations.

static[=n]
: The iterations of a loop are divided into chunks that contain n iterations. Threads are assigned chunks in a "round-robin" fashion. This is known as block cyclic scheduling. If the value of n is 1, the scheduling type is specifically referred to as cyclic scheduling. If you have not specified n, the chunks will contain
CEILING(number_of_iterations / number_of_threads)
iterations. Each thread is assigned one of these chunks. This is known as block scheduling. If you have not specified schedule, the default is set to schedule=static, resulting in block scheduling.

Related Information: For more information, see the description of the SCHEDULE directive in the XL Fortran for AIX Language Reference. 74 XL Fortran for AIX: User's Guide

Parallel execution options

The three parallel execution options, parthds, usrthds, and stack,areas follows:

parthds=num
: Specifies the number of threads (num)to be used for parallel execution of code that you compiled with the -qsmp option. By default, this is equal to the number of online processors. There are some applications that cannot use more than some maximum

number of processors. There are also some applications that can achieve performance gains if they use more threads than there are processors. This option allows you full control over the number of execution threads. The default value for num is 1 if you did not specify -qsmp. Otherwise, it is the number of online processors on the machine. For more information, see the NUM_PARTHDS intrinsic function in the XL Fortran for AIX Language Reference.

usrthds=num
: Specifies the maximum number of threads (num) that you expect your code will explicitly create if the code does explicit thread creation. The default value for num is 0. For more information, see the NUM_USRTHDS intrinsic function in the XL Fortran for AIX Language Reference.

stack=num
: Specifies the largest amount of space in bytes (num) that a thread's stack will need. The default value for num is 4194304.

SGI

Fortran

To Enable OpenMP in the compiler

```
f90   -mp -MP:open_mp=ON myprog.f90
```

C

```
cc   -mp -MP:open_mp=ON myprog.c
```

Notes on environment variables:

OMP_SCHEDULE
: The default value for this environment variable is STATIC.

OMP_NUM_THREADS

The default value is the minimum of 8 and the number of CPUs on the system.

OMP_DYNAMIC

The default value is TRUE.

OMP_NESTED

The default is FALSE

Other notes:

You can specify default scheduling on the compile line. From the f90 man page we find:

```
     -mp_schedtype=mode
               Specifies a default mode for scheduling work among the
               participating tasks in loops.  This option must be specified
               in conjunction with -mp.  Specifying this option has the
               same effect as putting a !$MP_SCHEDTYPE=mode directive at
               the beginning of the file.  Specify one of the following for
               mode:

               mode           Action

               DYNAMIC        Breaks the iterations into pieces, the size
                              of which is specified by the -chunk=integer
                              option.  As each process finishes a piece, it
                              enters a critical section and obtains the
                              next available piece.  For more information,
                              see the -chunk=integer option.

               GSS            Schedules pieces according to the sizes of
                              the pieces awaiting execution.

               INTERLEAVE     Breaks the iterations into pieces, this size
                              of which is specified by the -chunk=integer
                              option.  Execution of the pieces is
```

	interleaved among the processes. For more information, see the -chunk=integer option.
RUNTIME	Schedules pieces according to information contained in the MP_SCHEDTYPE environment variable.
SIMPLE	Divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process. Default.

Additional information can be obtain by running the following commands:

```
man 5 pe_environ
```

```
relnotes ftn90_fe
```

Sun Microsystems

To Enable OpenMP in the compiler

Fortran

```
f90 -explicitpar -stackvar -mp=openmp myprog.f90
```

or

```
f90   -openmp
```

This option is a macro for the combination of options:

```
             -mp=openmp -explicitpar -stackvar -D_OPENMP=2000011
```

-D_OPENMP=2000011 specifies the November 2000 version of OpenMP, that is version 2.0

```
f90   -xopenmp
```

-xopenmp is a synonym for -openmp

C

```
cc -xopenmp=parallel myprog.c
```

Notes on environment variables:

OMP_SCHEDULE
: Default value of STATIC is used

OMP_NUM_THREADS
: Default of 1 is used.

OMP_DYNAMIC
: A default value of TRUE is used.

OMP_NESTED
: Ignored. Nested parallelism is not supported.

Other notes:

Environment variables not part of the OpenMP Fortran API:

SUNW_MP_THR_IDLE

Controls the end-of-task status of each thread executing the parallel part of a program. You can set the value to spin, sleep ns, or sleep nms. The default is SPIN: a thread should spin (or busy-wait) after completing a parallel task, until a new parallel task arrives. Choosing SLEEP time specifies the amount of time a thread should spin-wait after completing a parallel task. If, while a thread is spinning, a new task arrives for the thread, the tread executes the new task immediately. Otherwise, the thread goes to sleep and is awakened when a new task arrives. time may be specified in seconds, (ns), or just (n), or milliseconds, (nms). SLEEP with no argument puts the thread to sleep immediately after completing a parallel task. SLEEP, SLEEP (0), SLEEP (0s), and SLEEP (0ms) are all equivalent. Example: setenv SUNW_MP_THR_IDLE (50ms)

HP Tru64 UNIX and Linux Alpha Systems

To Enable OpenMP in the KAP compiler

Fortran

```
kf90 -fkapargs='-noconc' myprogram.f -omp -pthread -call_shared
```

C

```
kcc  -ckapargs='-noconc'  myprogram.c -omp  -pthread  -call_shared
```

The option -fkapargs='-noconc' turns off automatic parallelization.

OMP_SCHEDULE
    (static,dynamic,guided,runtime)
OMP_DYNAMIC
    Default is false.
OMP_NESTED
    Default is false.

OMP_NUM_THREADS
Default value is the number of processors on the current system.

Other notes:

IBM and HP_UX /Intel IA-32 and Itanium

To Enable OpenMP

Fortran

For IA-32

```
ifc -openmp   myprogram.f90
```

For Itanium

```
efc -openmp  myprogram.f90
```

Note: Specifing -openmp implies the -fpp option.

C

For IA-32

```
icc -openmp   myprogram.c
```

For Itanium

```
ecc -openmp  myprogram.c
```

OMP_SCHEDULE
    Defalut static,no chunk sizespecified
OMP_NUM_THREADS
    Defalut Number of processors
OMP_DYNAMIC
    Default .false.
OMP_NESTED Enables
    Default .false.

Other notes:

The user can specify the detail level of the report on OpenMP parallelization:

-openmp_report{0|1|2}
    Controls the OpenMP parallelizers diagnostic levels. The default is 1

Portability and thread scheduling

Link to example source.

There is a common misconception about OpenMP. Programmers often assume that there is a guarantee that a particular thread, or collection of threads, will execute some block of code. In some cases there is no such guarantee provided by OpenMP. Programs that are written assuming such a guarantee might produce different results on different machines and thus are not portable. They may also produce different results on subsequent runs on the same machine.

We will look at two examples. The first deals with do loops the second with parallel regions outside of do loops. Assume we are running the examples using 4 threads.

Programmers will often write something like

```
!$OMP parallel do
do n=1,4
  if(omp_get_thread_num() .eq. 2)then
    call some_func(n)
  else
	call another_func(n)
  endif
enddo
```

A programmer might write this block of code assuming that some_func will be called with n=2. There are three possibilities for this program: some_func is called with n=2, some_func is called with n not equal to 2, and sume_func is not called at all.

In the case given above, the programmer has not specified a schedule type for the do loop. Thus the implementors of the OpenMP compiler is free to chose a schedule for this loop. In an exstream, the schedule could be such that all work is given to the first thread and none to the others. (This would actually make sense if the overhead of handling threads was high and the amount of work for each iteration was small.) If all of the work was given to the first thread then the function some_func would never be called.

Programmer writers are free to specify schedules. For example, you could specify a STATIC schedule with a chunk size of 1, like

```
!$OMP parallel do schedule(static,1)
```

The static schedule has the following meaning:

STATIC
: When schedule (static, chunk_size) is specified, iterations are divided into chunks of size specified by chunk_size. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.

Will this guarantee that some_func is called with n=4. No. We read in the OpenMP standard at the end of section 2.4.1. "An OpenMP-compliant program should not rely on a particular schedule for correct execution. A program should not rely on a schedule kind conforming precisely to the description given above, because it is possible to have variations in the implementation of the same schedule kind across different compilers. The description can be used to select the schedule that is appropriate for a particular situation."

In other words, programmers should use the schedule clause to **suggest** to the compiler how the work should be distributed. For some reason, maybe known only to them, the compiler writer is free to, ignore the schedule clause.

Here is a slight variation on the example given above.

```
!$OMP end parallel
!$OMP parallel do
  do n=1,8
  if(n .eq. 1)then
    call my_wait(x)
  endif
  write(*,*)n,omp_get_thread_num()
  enddo
```

Here we specify that a statement, the call to my_wait, be executed with n=1. The function my_wait is designed to pause for a given number of seconds. In this case, the value of x can effect the thread that executes the write statement. For a machine that uses dynamic scheduling as the default, you might get the following if x=0.

```
 1,  0
 2,  1
 3,  2
 4,  3
 5,  0
 6,  1
 7,  2
 8,  3
```

Or,if x=1 you could get

```
 2,  1
 3,  1
 4,  2
 5,  3
 6,  1
 7,  2
 8,  3
 1,  0
```

so that thread 0 only gets one iteration of the do loop. Or we could even get

```
 2,  1
 3,  1
 4,  1
 5,  1
 6,  1
 7,  1
 8,  1
 1,  0
```

Let's look at another example.

```
!$OMP parallel
        myt=omp_get_thread_num()
        write(*,*)"thread= ",myt," of ",OMP_GET_NUM_THREADS()
!$OMP end parallel
```

Clearly, the ordering of the output from the write statement is non deterministic so the programmer might expect something like

```
 thread=  2 of 4
 thread=  1 of 4
 thread=  0 of 4
 thread=  3 of 4
```

But you might get

```
 thread=  3  of  4
 thread=  3  of  4
 thread=  3  of  4
 thread=  3  of  4
```

Both of these are actual results produced on different machines. Both sets of results are legal. From the OpenMP version 2.0 specification, page 3 line 224 we have: "When a parallel construct is encountered, the master thread creates a team of threads, and the master thread becomes the master of the team. The statements in the program that are enclosed by the parallel construct, including routines called from within the enclosed statements, are executed in parallel by each thread in the team."

For this simple section of code, every thread executes every statement but the ordering of the statement execution, is not fixed. In the first case, thread 2 set the value of myt to 2 and then printed the value. Then thread 1 did the same and so on. In the second case all of the threads set the value of myt. Thread 3 was the final thread to set the value. Thus, when each thread took a turn at printing the value of myt it printed the final value 3.

With a slight modification of this program, it will always produce an enumeration of all of the threads.

```
!$OMP parallel
!$OMP critical
        myt=omp_get_thread_num()
        write(*,*)"critical thread= ",myt
!$OMP end critical
!$OMP end parallel
```

In this case, the critical directive causes each thread to call omp_get_thread_num then to do the write before another thread can change the value. The ordering of the output from each thread is still non deterministic so we might get

```
 critical thread=  0
 critical thread=  2
 critical thread=  3
 critical thread=  1
```

The program combines these examples. It was run twice on a collection of machines with the results reported below.

Program listing

```
program ompf
    integer OMP_GET_THREAD_NUM,OMP_GET_NUM_THREADS
    logical wt(0:3)
!$OMP parallel
        myt=omp_get_thread_num()
        write(*,*)"myt= ",myt," of ",OMP_GET_NUM_THREADS()
!$OMP end parallel
!$OMP parallel
!$OMP critical
```

```
        myt=omp_get_thread_num()
        write(*,*)"inside critical myt= ",myt
!$OMP end critical
!$OMP end parallel
    x=0.0
!$OMP parallel do
    do n=1,8
       if(n .eq. 1)then
          call my_wait(x)
       endif
!$OMP critical
       write(*,*)n,omp_get_thread_num()
!$OMP end critical
    enddo
    write (*,*)"*********"
    x=10.0
!$OMP parallel do
    do n=1,8
       if(n .eq. 1)then
          call my_wait(x)
       endif
!$OMP critical
       write(*,*)n,omp_get_thread_num()
!$OMP end critical
    enddo
end
subroutine my_wait(x)
    real x
    integer ct1,ct2,cr,cm,cend
    if(x .le. 0.0)return
    call system_clock(ct1,cr,cm)
    cend=ct1+nint(x*float(cr))
    do
      call system_clock(ct2)
      if(ct2 .lt. ct1)ct2=ct1+cm
      if(ct2 .ge. cend)return
    enddo
end subroutine
```

Results

IBM SP xlf90 version 7.x

Run 1

```
% ./a.out
 myt=  0  of  4
 myt=  0  of  4
 myt=  0  of  4
 myt=  0  of  4
 inside critical myt=  1
 inside critical myt=  3
 inside critical myt=  0
 inside critical myt=  2
 3 1
 4 1
 5 2
 6 2
 7 3
 8 3
 1 0
```

Run 2

```
% ./a.out
 myt=  2  of  4
 myt=  2  of  4
 myt=  2  of  4
 myt=  2  of  4
 inside critical myt=  0
 inside critical myt=  1
 inside critical myt=  3
 inside critical myt=  2
 7 3
 8 3
 3 1
 4 1
 5 2
 6 2
 1 0
 2 0
 *********
 3 1
 4 1
 5 2
 6 2
 7 3
 8 3
 1 0
 2 0
%
```

```
 2 0
 *********
 7 3
 8 3
 5 2
 6 2
 3 1
 4 1
 1 0
 2 0
%
```

Cray Sv1 Fortran: Version 3.5.0.4

Run 1

```
% ./a.out
 myt=  3  of  4
 myt=  3  of  4
 myt=  3  of  4
 myt=  3  of  4
 inside critical myt=  0
 inside critical myt=  1
 inside critical myt=  2
 inside critical myt=  3
 1,  0
 2,  1
 3,  2
 4,  0
 5,  3
```

Run 2

```
% ./a.out
 myt=  3  of  4
 myt=  3  of  4
 myt=  3  of  4
 myt=  3  of  4
 inside critical myt=  0
 inside critical myt=  1
 inside critical myt=  2
 inside critical myt=  3
 1,  0
 2,  1
 3,  2
 4,  3
 5,  0
 6,  1
 7,  2
 8,  0
 *********
 2,  3
 2*3
 4,  1
 5,  2
 6,  3
 7,  1
 8,  2
 1,  0
%
```

```
 6,  1
 7,  2
 8,  0
 *********
 2,  3
 2*3
 4,  3
 5,  3
 6,  3
 7,  3
 8,  3
 1,  0
%
```

Intel Fortran Itanium(R) compiler version 7.0

Run 1

```
% ./a.out
 myt=           0  of            4
 myt=           2  of            4
 myt=           3  of            4
 myt=           1  of            4
 inside critical myt=           0
 inside critical myt=           1
 inside critical myt=           2
 inside critical myt=           3
          1           0
          5           2
          7           3
```

Run 2

```
%./a.out
 myt=           1  of            4
 myt=           0  of            4
 myt=           2  of            4
 myt=           3  of            4
 inside critical myt=           1
 inside critical myt=           2
 inside critical myt=           0
 inside critical myt=           3
          5           2
          3           1
          7           3
          2          0
          6          2
          8          3
          3          1
          4          1
 *********
          3          1
          5          2
          4          1
          6          2
          7          3
          8          3
          1          0
          2          0
%
```

```
          1          0
          6          2
          4          1
          8          3
          2          0
 *********
          3          1
          5          2
          4          1
          6          2
          7          3
          8          3
          1          0
          2          0
%
```

```
program ompf
    integer OMP_GET_THREAD_NUM,OMP_GET_NUM_THREADS
    logical wt(0:3)
!$OMP parallel
        myt=omp_get_thread_num()
        write(*,*)"myt= ",myt," of ",OMP_GET_NUM_THREADS()
!$OMP end parallel
!$OMP parallel
!$OMP critical
        myt=omp_get_thread_num()
        write(*,*)"inside critical myt= ",myt
!$OMP end critical
!$OMP end parallel
    x=0.0
!$OMP parallel do
    do n=1,8
       if(n .eq. 1)then
          call my_wait(x)
       endif
!$OMP critical
       write(*,*)n,omp_get_thread_num()
!$OMP end critical
    enddo
    write (*,*)"*********"
    x=10.0
!$OMP parallel do
    do n=1,8
       if(n .eq. 1)then
          call my_wait(x)
       endif
!$OMP critical
       write(*,*)n,omp_get_thread_num()
!$OMP end critical
    enddo
end
subroutine my_wait(x)
    real x
    integer ct1,ct2,cr,cm,cend
    if(x .le. 0.0)return
    call system_clock(ct1,cr,cm)
    cend=ct1+nint(x*float(cr))
    do
      call system_clock(ct2)
      if(ct2 .lt. ct1)ct2=ct1+cm
      if(ct2 .ge. cend)return
    enddo
```

```
end subroutine
```

Effects of schedule types

Link to example source.

The purpose of this article is to show that loop schedule type can dramatically effect run time. We present three simple routines. These routines are called in a loop using various schedules. The run times vary depending on schedule type and in some cases the results are not what is expected. In particular, the thread that performs a specific iteration count of a loop is not always the one implied by the schedule.

Consider the following three routines

1. **all_fast**

```
void all_fast() {
  int k;
  k=omp_get_thread_num();
  dist[k]++;
}
```

2. **zero_slow**

```
void zero_slow() {
  int k;
  FLT x,y;
  k=omp_get_thread_num();
  dist[k]++;
  if(k == 0) {
        x=system_clock((FLT*)0);
        y=x+1;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
}
```

3. **imbalance**

```
void imbalance (int i) {
  int k;
  FLT x,y;
  k=omp_get_thread_num();
  dist[k]++;
  if(i == 1) {
    idid=k;
        x=system_clock((FLT*)0);
        y=x+1;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
  else {
        x=system_clock((FLT*)0);
        y=x+0.01;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
}
```

The first routine, all_fast, does no real work. It just increments a counter dist[k], where k is the thread number calling the routine.

The second routine addes a block of code that spins for one second if the thread calling the routine is thread 0.

The third routine, imbalance, runs for 1 second if the input value is 1 and 0.01 seconds for other input values.

We are interested in seeing what will happen if these routines are called in a for loop using different types of scheduling. Why? Even if you manually specify scheduling, some implementations of openmp have slightly differnt scheduling algorithms. That is, different threads than would be expected from the specified scheduling, run individual iterations of a for loop. This illustrates that uses sould not rely on specific threads running specific iterations of a loop. We also see that different scheduling algorithms can lead to dramaticily different run times.

We call these routine in a wrapper program that reports the type of scheduling, the routine called, the total number of iterations, and the precent of the iterations that are performmed by each thread. For the imbalance routine we also report which thread ran the slow iteration. We compare the results to what might be expected.

Output	Expected

`******  default scheduling`
`******  for a subroutine with little work`
`0  25.00 %`
`1  25.00 %`
`2  25.00 %`
`3  25.00 %`
` total iterations: 400 time       0.00`
	Each thread does an equal number of iterations as expected. The total time is less than 0.01 seconds.

Output	Expected

```
******  default scheduling
******  for a subroutine with thread 0 given 1 second of work
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 16 time       4.00
```

It looks like this machine is doing static scheduling. It is giving thread 0 4 iterations each taking 1 second. It would be faster if thread 0 was not given as much work, say using dynamic scheduling.

Output | Expected

```
******  schedule(static,1)
******  for a subroutine with thread 0 given 1 second of work
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 16 time       4.00
```

The results are 25% for each thread as expected. It would be faster if thread 0 was not given as much work, say using dynamic scheduling.

Output | Expected

```
******  schedule(static,2)
******  for a subroutine with thread 0 given 1 second of work
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 16 time       4.00
```

The results are 25% for each thread as expected. It would be faster if thread 0 was not given as much work, say using dynamic scheduling.

Output

Expected

```
******  schedule(dynamic,1)
******  for a subroutine with thread 0 given 1 second of work
0   6.25 %
1  87.50 %
2   0.00 %
3   6.25 %
 total iterations: 16 time       1.00
```

Dynamic scheduling allows thread 0 to be given a single iteration. The time for the loop is still dominated by the time used by thread 0. This is expected. The suprize in this case is that the rest of the iterations are not distributed to the remaining threads evenly.

Output

Expected

```
******  schedule(dynamic,2)
******  for a subroutine with thread 0 given 1 second of work
0   0.00 %
1   0.00 %
2  12.50 %
3  87.50 %
 total iterations: 16 time       0.00
```

It is expected that thread 0 be given an iteration. It is not so the loop runs very fast. The other suprize in this case is that the rest of the iterations are not distributed to the remaining threads evenly.

Output

Expected

```
******  schedule(dynamic,4)
******  for a subroutine with thread 0 given 1 second of work
0   0.00 %
1 100.00 %
2   0.00 %
3   0.00 %
 total iterations: 16 time       0.00
```

It is expected that thread 0 be given an iteration. It is not so the loop runs very fast. The other suprize in this case is that the rest of the iterations are not distributed to the remaining threads evenly.

Output

Expected

```
******  default scheduling
******  for an imbalanced subroutine
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 400 time       1.99
 thread 0 did the slow iteration
```

As said above, this machine used static scheduling by default. Each thread gets 25% of the iterations. The run time is determined by thread 0 that doing the slow iteration, 1 second, followed by 99 iterations of 0.01 seconds. The other threads are idle while it is doing the 99 iterations.

Output

```
******  schedule(static,1)
******  for an imbalanced subroutine
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 400 time       1.99
 thread 0 did the slow iteration
```

Expected

Each thread gets 25% of the iterations. The run time is determined by thread 0 that doing the slow iteration, 1 second, followed by 99 iterations of 0.01 seconds. The other threads are idle while it is doing the 99 iterations.

Output

Expected

```
******  schedule(static,2)
******  for an imbalanced subroutine
0  25.00 %
1  25.00 %
2  25.00 %
3  25.00 %
 total iterations: 400 time       1.99
 thread 0 did the slow iteration
```

Each thread gets 25% of the iterations. The run time is determined by thread 0 that doing the slow iteration, 1 second, followed by 99 iterations of 0.01 seconds. The other threads are idle while it is doing the 99 iterations.

Output

Expected

```
******  schedule(dynamic,1)
******  for an imbalanced subroutine
0  31.00 %
1  31.25 %
2   6.50 %
3  31.25 %
 total iterations: 400 time       1.25
 thread 2 did the slow iteration
```

It might be expected that thread 1 would get the slow iteration. However, the performance for this loop is good, 1.25 seconds, compared to static scheduling, 1.99 seconds. Thread 2 does the slow iteration, 1 second, while the other threads do 100 iterations in 1 second for a total of 301 iterations. The remaining 99 iterations are distributed amongst the 4 threads for about 0.25 seconds. This is the optimum solution.

Output

Expected

```
******  schedule(dynamic,2)
******  for an imbalanced subroutine
0  31.50 %
1   6.50 %
2  31.00 %
3  31.00 %
 total iterations: 400 time       1.26
 thread 1 did the slow iteration
```

It might be expected that thread 1 would get the slow iteration. However, the performance for this loop is good, 1.25 seconds, compared to static scheduling, 1.99 seconds. Thread 2 does the slow iteration, 1 second, while the other threads do 100 iterations in 1 second for a total of 301 iterations. The remaining 99 iterations are distributed amongst the 4 threads for about 0.25 seconds. This is a near optimum solution.

Output

```
******  schedule(dynamic,4)
******  for an imbalanced subroutine
0   6.50 %
1  31.50 %
2  31.00 %
3  31.00 %
 total iterations: 400 time       1.26
 thread 0 did the slow iteration
```

Expected

It might be expected that thread 1 would get the slow iteration. However, the performance for this loop is good, 1.25 seconds, compared to static scheduling, 1.99 seconds. Thread 2 does the slow iteration, 1 second, while the other threads do 100 iterations in 1 second for a total of 301 iterations. The remaining 99 iterations are distributed amongst the 4 threads for about 0.25 seconds. This is a near optimum solution.

t5.c

```c
/* cc  -lm t4.c -qsmp */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#define FLT double

/* utility routines */
void my_bar();
void explain(char astr[]);
FLT system_clock(FLT *x);
void start_time();
FLT end_time();

/* array used to determine how much work each thread performs */
int *dist,idid;
FLT st;

/* routine to reset dist */
void zero(int j);

/* work routines */
void all_fast();
void zero_slow();
void a_slow(int i);


void all_fast() {
  int k;
  k=omp_get_thread_num();
  dist[k]++;
}

void zero_slow() {
  int k;
  FLT x,y;
  k=omp_get_thread_num();
  dist[k]++;
  if(k == 0) {
        x=system_clock((FLT*)0);
        y=x+1;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
```

```
}

void imbalance (int i) {
  int k;
  FLT x,y;
  k=omp_get_thread_num();
  dist[k]++;
  if(i == 1) {
    idid=k;
        x=system_clock((FLT*)0);
        y=x+1;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
  else {
        x=system_clock((FLT*)0);
        y=x+0.01;
        while(x < y) {
                x=system_clock((FLT*)0);
        }
  }
}

main() {
int  i,k,max_threads,total;
max_threads=omp_get_max_threads();
printf("max threads = %d\n",max_threads);
dist=(int*)malloc(max_threads*sizeof(int));
zero(max_threads);
total=0;
explain("report the % of iterations for each thread");
explain("for a set of loops");
explain("******");
explain("default scheduling");
explain("for a subroutine with little work");
k=max_threads*100;
start_time();
#pragma omp parallel for
    for( i=1;i<=k;i++) {
                all_fast();
    }
my_bar();
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);
explain("default scheduling");
explain("for a subroutine with thread 0 given 1 second of work");
k=max_threads*4;
start_time();
#pragma omp parallel for
    for( i=1;i<=k;i++) {
                zero_slow();
    }
my_bar();
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);
explain("schedule(static,1)");
explain("for a subroutine with thread 0 given 1 second of work");
start_time();
#pragma omp parallel for schedule(static,1)
    for( i=1;i<=k;i++) {
                zero_slow();
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);
explain("schedule(static,2)");
explain("for a subroutine with thread 0 given 1 second of work");
start_time();
#pragma omp parallel for schedule(static,2)
    for( i=1;i<=k;i++) {
                zero_slow();
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
```

```
total=0;
zero(max_threads);
explain("schedule(dynamic,1)");
explain("for a subroutine with thread 0 given 1 second of work");
start_time();
#pragma omp parallel for schedule(dynamic,1)
    for( i=1;i<=k;i++) {
                zero_slow();
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);
explain("schedule(dynamic,2)");
explain("for a subroutine with thread 0 given 1 second of work");
start_time();
#pragma omp parallel for schedule(dynamic,2)
    for( i=1;i<=k;i++) {
                zero_slow();
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);
explain("schedule(dynamic,4)");
explain("for a subroutine with thread 0 given 1 second of work");
start_time();
#pragma omp parallel for schedule(dynamic,2)
    for( i=1;i<=k;i++) {
                zero_slow();
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n\n",total, end_time());
total=0;
zero(max_threads);

explain("default scheduling");
explain("for an imbalanced subroutine");
k=max_threads*100;
start_time();
#pragma omp parallel for
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
my_bar();
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
zero(max_threads);
explain("default scheduling");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
my_bar();
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
zero(max_threads);
explain("schedule(static,1)");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for schedule(static,1)
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
```

```
total=0;
zero(max_threads);
explain("schedule(static,2)");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for schedule(static,2)
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
zero(max_threads);
explain("schedule(dynamic,1)");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for schedule(dynamic,1)
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
zero(max_threads);
explain("schedule(dynamic,2)");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for schedule(dynamic,2)
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
```

```
zero(max_threads);
explain("schedule(dynamic,4)");
explain("for an imbalanced subroutine");
start_time();
#pragma omp parallel for schedule(dynamic,2)
    for( i=1;i<=k;i++) {
                imbalance(i);
    }
for( i=0;i<max_threads;i++) {
                printf("%d %6.2f %%\n",i,100.0*(FLT)dist[i]/((FLT)k));
                total=total+dist[i];
    }
printf(" total iterations: %d time %10.2f\n",total, end_time());
printf(" thread %d did the slow iteration\n\n",idid);
total=0;
my_bar();
}


void my_bar() {
#pragma omp barrier
        fflush(stdout);
#pragma omp barrier
}

void explain(char astr[]){
        printf("******  %s\n",astr);
}

FLT system_clock(FLT *x) {
        FLT t;
        FLT six=1.0e-6;
        struct timeval tb;
        struct timezone tz;
        gettimeofday(&tb,&tz);
        t=(FLT)tb.tv_sec+((FLT)tb.tv_usec)*six;
        if(x){
                *x=t;
        }
        return(t);
}

void zero(int j) {
int i;
    for( i=0;i<j;i++) {
                dist[i]=0;
    }
}

void start_time() {
        st=system_clock((FLT*)0);
}

FLT end_time() {
        return (system_clock((FLT*)0)-st);
}
```

Parallel Sections

Link to example source.

Link to example source in Fortran.

The emphasis in most OpenMP examples and descriptions is on loop level parallelism. However, OpenMP also has a more coarse grained parallelism construct, the parallel section. The syntax is

```
#pragma omp parallel sections
 {
#pragma omp section
         {
/*fist block of code */
         }

#pragma omp section
         {
/*second block of code */
         }

#pragma omp section
         {
/*third block of code */
         }

#pragma omp section
         {
/*fourth block of code */
         }
 }
```

There can be an arbitrary number of code blocks or sections. The requirement is that the individual sections be independent. Since the sections are independent they can be run in parallel. So if you have 4 sections and are running using 4 threads each thread should run a block of code in a section in parallel with the others. If you have more threads than sections then some threads will be idle.

We give as an example a set of matrix inversions performed on four different matrices. We do the inversions twice and compare to the original matrices. We have a routine mset that puts values in the matrices, a routine over that does an inversion and a routine mcheck that checks the results and returns a difference form the original matrix. The routine system_clock is for timing. The routine mset is called outside of the parallel section. Our parallel section then looks like:

```
#pragma omp section
         {
           system_clock(&t1_start);
           over(m1,n);
           over(m1,n);
           system_clock(&t1_end);
           e1=mcheck(m1,n,1);
           t1_start=t1_start-t0_start;
           t1_end=t1_end-t0_start;
}
```

After doing the parallel sections we print the time spent in each section.

The program run using 4 threads returned the following

```
[geight]% setenv OMP_NUM_THREADS 4
[geight]% ./a.out
section 1 start time= 0.00056601   end time=     2.6892  error= 3.43807e-07
section 2 start time=   0.011296   end time=     2.9498  error= 6.04424e-07
section 3 start time=  0.0071419   end time=     2.9925  error= 3.67327e-06
section 4 start time= 0.00054705   end time=     2.9233  error= 3.42406e-06
[geight] %
```

This shows that the 4 threads did the matrix inversions in parallel.

Using two threads we get the following:

```
[geight]% setenv OMP_NUM_THREADS 2
[geight]% ./a.out
section 1 start time= 0.00039494   end time=     1.3827  error= 3.43807e-07
section 2 start time= 0.00038493   end time=     1.5283  error= 6.04424e-07
section 3 start time=     1.3862   end time=     2.8165  error= 3.67327e-06
section 4 start time=     1.5319   end time=     3.0124  error= 3.42406e-06
[geight]%
```

We have the first and section sections run in parallel and then the third and fourth sections run in parallel.

```
/* cc  -lm t4.c -qsmp */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#define FLT double

/* utility routines */
FLT system_clock(FLT *x);
FLT **matrix(int nrl,int nrh,int ncl,int nch);

/* work routines */
void mset(FLT **m, int n, int in);
FLT mcheck(FLT **m, int n, int in);
void over(FLT ** mat,int size);

main() {
    FLT **m1,**m2,**m3,**m4;
    FLT t0_start;
    FLT t1_start,t1_end,e1;
    FLT t2_start,t2_end,e2;
    FLT t3_start,t3_end,e3;
    FLT t4_start,t4_end,e4;
    int n;
    n=200;
    m1=matrix(1,n,1,n);
    m2=matrix(1,n,1,n);
    m3=matrix(1,n,1,n);
    m4=matrix(1,n,1,n);
        mset(m1,n,1);
    mset(m2,n,2);
    mset(m3,n,3);
    mset(m4,n,4);

    system_clock(&t0_start);

#pragma omp parallel sections
 {
#pragma omp section
         {
                        system_clock(&t1_start);
                        over(m1,n);
                        over(m1,n);
                        system_clock(&t1_end);
                        e1=mcheck(m1,n,1);
                        t1_start=t1_start-t0_start;
                        t1_end=t1_end-t0_start;
         }
#pragma omp section
         {
                        system_clock(&t2_start);
                        over(m2,n);
                        over(m2,n);
                        system_clock(&t2_end);
                        e2=mcheck(m2,n,2);
                        t2_start=t2_start-t0_start;
                        t2_end=t2_end-t0_start;
         }
#pragma omp section
         {
           system_clock(&t3_start);
           over(m3,n);
           over(m3,n);
           system_clock(&t3_end);
           e3=mcheck(m3,n,3);
           t3_start=t3_start-t0_start;
           t3_end=t3_end-t0_start;
         }
#pragma omp section
         {
           system_clock(&t4_start);
           over(m4,n);
           over(m4,n);
           system_clock(&t4_end);
           e4=mcheck(m4,n,4);
           t4_start=t4_start-t0_start;
           t4_end=t4_end-t0_start;
         }
 }
 printf("section 1 start time= %10.5g   end time= %10.5g  error= %g\n",t1_start,t1_end,e1);
 printf("section 2 start time= %10.5g   end time= %10.5g  error= %g\n",t2_start,t2_end,e2);
 printf("section 3 start time= %10.5g   end time= %10.5g  error= %g\n",t3_start,t3_end,e3);
 printf("section 4 start time= %10.5g   end time= %10.5g  error= %g\n",t4_start,t4_end,e4);
}

void mset(FLT **m, int n, int in) {
        int i,j;
    for(i=1;i<=n;i++)
       for(j=1;j<=n;j++) {
           if(i == j) {
               m[i][j]=in;
           } else {
               m[i][j]=i+j;
           }
       }

}

FLT mcheck(FLT **m, int n, int in) {
        int i,j;
        FLT x;
    x=0.0;
    for(i=1;i<=n;i++)
       for(j=1;j<=n;j++) {
           if(i == j) {
               x=x+fabs(m[i][j]-in);
           } else {
               x=x+fabs(m[i][j]-(i+j));
           }
       }
   return x;
}

void over(FLT ** mat,int size)
{
        int k, jj, kp1, i, j, l, krow, irow;
        FLT pivot, temp;
        FLT sw[2000][2];
        for (k = 1 ;k<= size ; k++)
        {
                jj = k;
                if (k != size)
                {
                        kp1 = k + 1;
                        pivot = fabs(mat[k][k]);
                        for( i = kp1;i<= size ;i++)
                        {
                                temp = fabs(mat[i][k]);
                                if (pivot < temp)
                                {
                                        pivot = temp;
                                        jj = i;
                                }
                        }
                }
                sw[k][0] =k;
                sw[k][1] = jj;
                if (jj != k)
                        for (j = 1 ;j<= size; j++)
                        {
                                temp = mat[jj][j];
                                mat[jj][j] = mat[k][ j];
                                mat[k][j] = temp;
                        }
                for (j = 1 ;j<= size; j++)
                        if (j != k)
                                mat[k][j] = mat[k][j] / mat[k][k];
                mat[k][k] = 1.0 / mat[k][k];
                for (i = 1; i<=size; i++)
                        if (i != k)
                                for (j = 1;j<=size; j++)
                                        if (j != k)
                                                mat[i][j] = mat[i][j] - mat[k][j] * mat[i][k];
                for (i = 1;i<=size;i++)
                        if (i != k)
                                mat[i][k] = -mat[i][k] * mat[k][k];
        }
        for (l = 1; l<=size; ++l)
        {
                k = size - l + 1;
                krow = sw[k][0];
                irow = sw[k][1];
                if (krow != irow)
                        for (i = 1; i<= size; ++i)
                        {
                                temp = mat[i][krow];
                                mat[i][krow] = mat[i][irow];
                                mat[i][irow] = temp;
                        }
        }
}

/*
The routine matrix was  adapted from
Numerical Recipes in C The Art of Scientific Computing
Press, Flannery, Teukolsky, Vetting
Cambridge University Press, 1988.
*/
FLT **matrix(int nrl,int nrh,int ncl,int nch)
{
    int i;
        FLT **m;
        m=(FLT **) malloc((unsigned) (nrh-nrl+1)*sizeof(FLT*));
        if (!m){
             printf("allocation failure 1 in matrix()\n");
             exit(1);
        }
        m -= nrl;
        for(i=nrl;i<=nrh;i++) {
            if(i == nrl){
                    m[i]=(FLT *) malloc((unsigned) (nrh-nrl+1)*(nch-ncl+1)*sizeof(FLT));
                    if (!m[i]){
                         printf("allocation failure 2 in matrix()\n");
                         exit(1);
                    }
                    m[i] -= ncl;
            }
            else {
                m[i]=m[i-1]+(nch-ncl+1);
            }
        }
        return m;
}

FLT system_clock(FLT *x) {
        FLT t;
        FLT six=1.0e-6;
        struct timeval tb;
        struct timezone tz;
        gettimeofday(&tb,&tz);
        t=(FLT)tb.tv_sec+((FLT)tb.tv_usec)*six;
        if(x){
                *x=t;
        }
        return(t);
}
```

```
module ccm_numz
! basic real types
    integer, parameter:: b8 = selected_real_kind(10)
contains
     function ccm_time()
        implicit none
        integer i
        integer :: ccm_start_time(8) = (/(-100,i=1,8)/)
        real(b8) :: ccm_time,tmp
        integer,parameter :: norm(13)=(/  &
               0, 2678400, 5097600, 7776000,10368000,13046400,&
        15638400,18316800,20995200,23587200,26265600,28857600,31536000/)
        integer,parameter :: leap(13)=(/  &
               0, 2678400, 5184000, 7862400,10454400,13132800,&
        15724800,18403200,21081600,23673600,26352000,28944000,31622400/)
        integer :: values(8),m,sec
        save
        call date_and_time(values=values)
        if(mod(values(1),4) .eq. 0)then
           m=leap(values(2))
        else
           m=norm(values(2))
        endif
        sec=((values(3)*24+values(5))*60+values(6))*60+values(7)
        tmp=real(m,b8)+real(sec,b8)+real(values(8),b8)/1000.0_b8
        !write(*,*)"vals ",values
        if(values(1) .ne. ccm_start_time(1))then
            if(mod(ccm_start_time(1),4) .eq. 0)then
                tmp=tmp+real(leap(13),b8)
            else
                tmp=tmp+real(norm(13),b8)
            endif
        endif
        ccm_time=tmp
    end function

subroutine invert (matrix,size)
        implicit none
        real(b8) matrix(:,:)
        integer size
        integer switch,k, jj, kp1, i, j, l, krow, irow,nmax
        parameter (nmax=1000)
        dimension switch(nmax,2)
        real(b8) pivot,temp
        do  k = 1,size
                jj = k
                if (k .ne. size) then
                        kp1 = k + 1
                        pivot = (matrix(k, k))
                        do i = kp1,size
                                temp = (matrix(i, k))
                                if ( abs(pivot) .lt.  abs(temp)) then
                                        pivot = temp
                                        jj = i
                                endif
                        enddo
                endif
                switch(k, 1) = k
                switch(k, 2) = jj
                if (jj .ne. k) then
                        do  j = 1 ,size
                                temp = matrix(jj, j)
                                matrix(jj, j) = matrix(k, j)
                                matrix(k, j) = temp
                        enddo
                endif
                do j = 1,size
                        if (j .ne. k)matrix(k, j) = matrix(k, j) / matrix(k, k)
                enddo
                matrix(k, k) = 1.0_b8 / matrix(k, k)
                do  i = 1,size
                        if (i.ne.k) then
                                do  j = 1,size
                                        if(j.ne.k)matrix(i,j)=matrix(i,j)-matrix(k,j)*matrix(i,k)
                                enddo
                        endif
                enddo
                do i = 1, size
                        if (i .ne. k)matrix(i, k) = -matrix(i, k) * matrix(k, k)
                enddo
        enddo
        do  l = 1,size
                k = size - l + 1
                krow = switch(k, 1)
                irow = switch(k, 2)
                if (krow .ne. irow) then
                        do  i = 1,size
                                temp = matrix(i, krow)
                                matrix(i, krow) = matrix(i, irow)
                                matrix(i, irow) = temp
                        enddo
                endif
        enddo
end subroutine

subroutine mset(m,  n,  in)
        real(b8) :: m(:,:)
        integer n,in
        integer i,j
        do i=1,n
                do j=1,n
                        if( i .eq. j)then
                                m(i,j)=in
                        else
                                m(i,j)=i+j
                        endif
                enddo
        enddo
end subroutine

function mcheck(m,  n,  in)
        real(b8) :: m(:,:)
        real(b8) mcheck,x
        integer n,in
        integer i,j
        x=0
        do i=1,n
                do j=1,n
                        if( i .eq. j)then
                                x=x+abs(m(i,j)-in)
                        else
                                x=x+abs(m(i,J)-(i+j))
                        endif
                enddo
        enddo
        mcheck=x
end function
end module ccm_numz

program tover
        use ccm_numz
        real(b8),allocatable :: m1(:,:),m2(:,:),m3(:,:),m4(:,:)
        integer n
        real(b8) t0_start;
    real(b8) t1_start,t1_end,e1;
    real(b8) t2_start,t2_end,e2;
    real(b8) t3_start,t3_end,e3;
    real(b8) t4_start,t4_end,e4;
```

```
        n=200
        allocate(m1(n,n),m2(n,n),m3(n,n),m4(n,n))
        call mset(m1,n,1)
        call mset(m2,n,2)
        call mset(m3,n,3)
        call mset(m4,n,4)
        t0_start=ccm_time()
!$pragma omp parallel sections

!$pragma omp section
        t1_start=ccm_time()
        call invert(m1,n)
        call invert(m1,n)
        t1_end=ccm_time()
        e1=mcheck(m1,n,1)
        t1_start=t1_start-t0_start
        t1_end=t1_end-t0_start
!$pragma omp end section

!$pragma omp section
        t2_start=ccm_time()
        call invert(m2,n)
        call invert(m2,n)
        t2_end=ccm_time()
        e2=mcheck(m2,n,2)
        t2_start=t2_start-t0_start
        t2_end=t2_end-t0_start
!$pragma omp end section

!$pragma omp section
        t3_start=ccm_time()
        call invert(m3,n)
        call invert(m3,n)
        t3_end=ccm_time()
        e3=mcheck(m3,n,3)
        t3_start=t3_start-t0_start
        t3_end=t3_end-t0_start
!$pragma omp end section

!$pragma omp section
        t4_start=ccm_time()
        call invert(m4,n)
        call invert(m4,n)
        t4_end=ccm_time()
        e4=mcheck(m4,n,4)
        t4_start=t4_start-t0_start
        t4_end=t4_end-t0_start
!$pragma omp end section

!$pragma omp end parallel sections

 write(*,1)1,t1_start,t1_end,e1
 write(*,1)2,t2_start,t2_end,e2
 write(*,1)3,t3_start,t3_end,e3
 write(*,1)4,t4_start,t4_end,e4
 1 format("section ",i4," start time= ",g10.5," end time= ",g10.5," error=",g10.5)
 end program
```

Complex usages of Threadprivate

Link to example source.

There are two purposes for these examples. First, they illustrate a more complex usages of thread private. Also, the code is these examples broke many early implementations of OpenMP. Thus, these examples provide a test of for compilers.

One common usages of derived types in scientific computing is to create a type that represents a vector. In 2d this corresponds to a complex number. We could define such a type as:

```
#define FLT double
struct real_img {
    FLT xpart;
    FLT ypart;
};
```

Next we create an array of this data type and declare the array threadprivate.

```
struct real_img itype[9];
#pragma omp threadprivate(itype)
```

Recall that making a variable thread private implies that each thread gets its own copy of the variable.

In our main program we fill the array with values. If each element of the array is a point, then the collection of points describes an octagon.

```
        itype[0].ypart=0;
        itype[0].xpart=1;
        pi4=pi/4.0;
        for(i=1;i<=8;i++) {
                yvect=sin(i*pi4);
                xvect=cos(i*pi4);
                itype[i].ypart=yvect-itype[i-1].ypart;
                itype[i].xpart=xvect-itype[i-1].xpart;
        }
```

Now we call a subroutine, sub_1, that will work with the array of structures.

```
#pragma omp parallel for copyin(itype) schedule(static,1)
        for(i=1;i<=max_threads;i++) {
                sub_1();
        }
```

The copyin clause ensures that each thread gets a copy of the values entered into the array.

The routine sub_1 multiplies the values in itype by the (thread id + 1) and sums the values. Summing the values should bring you back to the point x=(thread id + 1),y=0. The final sum is printed and compared to the thread id. The routine also checks to see that itype has been allocated and it checks to see that values have been copied into the array.

The output from the program should look something like:

```
a complex test of thread private with a do loop
we pass in the array of structures
the structure is a 2d vector. we sum of all
the vectors to get us back to the real axis
we multiply the vectors times (thread+1)
from sub_1 thread 2, magnitude is thread+1= 3.00, angle is zero=0.00000000
from sub_1 thread 0, magnitude is thread+1= 1.00, angle is zero=0.00000000
from sub_1 thread 1, magnitude is thread+1= 2.00, angle is zero=0.00000000
from sub_1 thread 3, magnitude is thread+1= 4.00, angle is zero=0.00000000
```

Another reason to make data global is to have an easy way to pass it between subroutines. In an openmp environment you may want to make such data threadprivate. Can arrays be allocated inside of a subroutine for use in routines that are called by the subroutine? OpenMP should be able to handle but this broke early compilers.

We start with an integer pointer that we will use as a vector.

```
int *ray2;
#pragma omp threadprivate(ray2)
```

We nullify ray2 and then call the routine sub_2 inside a for loop.

```
        j=max_threads*2;
        k=j/2;
        ray2=0;
#pragma omp parallel for copyin(ray2) schedule(static,1)
        for(i=1;i<=j;i++) {
                sub_2(i,k);
        }
```

Note that we expect that each thread will call sub_2 two times, one time with i <= k and the second time with i > k. The routine sub_2 was written for illustration purposes only. Its output will vary depending on thread scheduling. If each thread does not call sub_2 as expected the routine will return different results than what are reported here. For a "real" program routines should not, in general, be written so that the correct results depend on scheduling. Scheduling directives are only suggestions to the compiler. Most compilers, in most instances, will follow the suggestions but there is no guarantee provided by the standard.

The routine sub_2 is shown below. If first checks to see if ray2 is allocated. If ray2 is null, it is allocated. Values are stuffed into the array and then sub_3 is called and returns x. The routine sub_3 sums the elements of ray2 and returns the sum as x. Finally, x is printed.

If sub_2 is called with n <= nhalf then ray2 is set to the thread id +1. If n > nhalf then ray2 is incremented by nhalf.

Both routines, sub_2 and sub_3 access ray2 as a global. This illustrates that we can use global variables to pass values while also using

OpenMP.

The first print statement in sub_2 shows that a thread is allocating ray2. The next print gives the address of ray2. Each thread should have its own copy of ray2 so each thread should print a different address. The final print gives the thread number and the values of n and x.

The address of ray2 and the values held in the array ray2 and should be preserved between invocations of sub_2. Once a thread assigns an address to ray2 that thread should have the same address. So each time a thread calls sub_2 it should print the same address. Also, the values stored in the array should be preserved.

If sub_2 is called as suggested by the scheduling directive,

schedule(static,1)

then each thread will call sub_2 twice. The first time with the value of n <=nhalf. The second time with n > nhalf. If the values are preserved between invocations, sub_2 will print the values 1 <= n <= max_threads*2 and x = n*10.

```
void sub_2(int n,int nhalf) {
    int i,k;
    FLT x;
    k=omp_get_thread_num();
    if(ray2 == 0) {
        ray2=(int *)malloc(10*sizeof(int));

                {
                printf("(a) thread %d allocating ray2\n",k);
                }
        }

        {
    printf("(b) thread %d ray2 is at %d\n",k,ray2);
    }
```

```
        if(n <= nhalf) {
                for(i=0;i<10;i++) {
                        ray2[i]=k+1;
                }
        }
        else {
                for(i=0;i<10;i++) {
                    if(ray2[i] != k+1) {
                        printf("value not preserved\n");
                    }
                        ray2[i]=ray2[i]+nhalf;
                }
        }

    x=sub_3(10);

        {
    printf("(c) thread %d ",k);
    printf("n= %d x= %g\n",n,x);
    }
}
```

Note that the print statements are given their own structured block. This is done so we could wrap them with a #pragma omp critical directive and add a statement to flush the output. The critical directives give us cleaner output. They are there to prevent multiple threads from trying to print at the same time. They do not effect the "real" output of this program, the value of x.

If we run using 4 threads and sort the output from sub_2 by thread number we get:

```
(a) thread 0 allocating ray2
(b) thread 0 ray2 is at 323312
(b) thread 0 ray2 is at 323312
(c) thread 0 n= 1 x= 10
```

```
(c) thread 0 n= 5 x= 50
(a) thread 1 allocating ray2
(b) thread 1 ray2 is at 323456
(b) thread 1 ray2 is at 323456
(c) thread 1 n= 2 x= 20
(c) thread 1 n= 6 x= 60
(a) thread 2 allocating ray2
(b) thread 2 ray2 is at 323408
(b) thread 2 ray2 is at 323408
(c) thread 2 n= 3 x= 30
(c) thread 2 n= 7 x= 70
(a) thread 3 allocating ray2
(b) thread 3 ray2 is at 323360
(b) thread 3 ray2 is at 323360
(c) thread 3 n= 4 x= 40
(c) thread 3 n= 8 x= 80
```

Note that the address for ray2 is consistent in time for each thread but each thread has a different value and the values for x are correct. This indicates that each thread had its own version of ray2 and its copy was preserved between thread invocations.

OpenMP also requires that thread private globals be preserved between parallel regions. This can be seen if we add another parallel region that access ray2. We can add the following block of code to print the address of ray2 before deallocating it

```
#pragma omp parallel
        {
                if(ray2) {
                        printf("thread %d deallocating ray2 at %d with
value=%d\n",omp_get_thread_num(),ray2,ray2[0]);
                        free(ray2);
                        ray2=0;
                }
        }
```

```
}
```

The output, sorted by thread number, shows that ray2 is preserved between parallel regions.

```
thread 0 deallocating ray2 at 323312 with value=5
thread 1 deallocating ray2 at 323456 with value=6
thread 2 deallocating ray2 at 323408 with value=7
thread 3 deallocating ray2 at 323360 with value=8
```

Care should be taken when passing pointers to parallel regions. Consider a slight variation to the example given above. What would happen if ray2 was allocated before the for loop? We could have something like:

```
        ray2=(int *)malloc(10*sizeof(int));
#pragma omp parallel for copyin(ray2) schedule(static,1)
        for(i=1;i<=j;i++) {
                sub_2(i,k);
        }
```

Ray2 is not allocated in sub_2 since we check for null before doing the allocation. In this case each thread has the same nonnull value for ray2 so ray2 points to the same block of memory for each thread.

This is confusing because there a several levels of indirection. Ray2 is a pointer that points to the block of memory allocated by the malloc. The pointer itself is held in some memory location. What gets duplicated by using the copyin clause is a pointer. That memory location for the pointer is different for each thread but the pointer still points to the same location in memory.

Hopefully another example will clear this up. Consider

```
        ray2=(int *)malloc(10*sizeof(int));
#pragma omp parallel for copyin(ray2) schedule(static,1)
        for(i=1;i<=j;i++) {
                sub_4();
        }
```

Where sub_4 will print

- The address of ray2:
  - the address of a pointer
- The value held by ray2:
  - address of where the pointer points to, obtained from the malloc
- ray2[0]:
  - the value of the first element of an array

Sub_4 is:

```
void sub_4() {
        printf("address of pointer ray2 %d, address held in ray2 %d, value held in 
ray2[0] %d\n",
  &ray2,ray2,ray2[0]);
}
```

The output from this block of code is:

```
address of pointer ray2 322592, address held in ray2 323312, value held in ray2[0] 
1234
address of pointer ray2  39168, address held in ray2 323312, value held in ray2[0] 
1234
address of pointer ray2 323296, address held in ray2 323312, value held in ray2[0] 
1234
address of pointer ray2 323280, address held in ray2 323312, value held in ray2[0] 
1234
```

Notice that for each thread ray2 points to the same memory location and thus ray2[0] is the same but each thread has a different address for the pointer.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>

#define FLT double
struct real_img {
    FLT xpart;
    FLT ypart;
};

struct real_img itype[9];
#pragma omp threadprivate(itype)

int *ray2;
#pragma omp threadprivate(ray2)

#define pi 3.1415926535897932384626433833
void sub_1();
void sub_2(int in, int in2);
FLT sub_3(int in);
void sub_4(int in, int in2);

void explain(char astr[]);



void sub_1() {
    int  i,j;
    FLT r,theta,x,y;
    r=0;
    theta=0;
    j=omp_get_thread_num();
    if(! itype) {
                printf("itype not set for thread %d\n",j);
    }
    for(i=0;i<=8;i++) {
        if(itype[i].xpart == 0 && itype[i].ypart == 0) {
                printf("itype not copied for thread %d\n",j);
        }
        itype[i].xpart=itype[i].xpart*(j+1);
        itype[i].ypart=itype[i].ypart*(j+1);
    }
    x=itype[0].xpart;
    y=itype[0].ypart;
    for(i=1;i<=8;i++) {
        x=x+itype[i].xpart;
        y=y+itype[i].ypart;
    }
    theta=atan2(y,x);
    r=sqrt(x*x+y*y);
#pragma omp critical
        {
    fflush(stdout);
    printf("from sub_1 thread %d, magnitude is thread+1= %4.2f, angle is zero=%10.8f\n",j,r,abs(theta));
    fflush(stdout);
    }
}


void sub_2(int n,int nhalf) {
    int i,k;
    FLT x;
    k=omp_get_thread_num();
    if(ray2 == 0) {
        ray2=(int *)malloc(10*sizeof(int));
#pragma omp critical
                {
                fflush(stdout);
                printf("(a) thread %d allocating ray2\n",k);
                }
        }
#pragma omp critical
        {
    fflush(stdout);
    printf("(b) thread %d ray2 is at %d\n",k,ray2);
    }

        if(n <= nhalf) {
                for(i=0;i<10;i++) {
                        ray2[i]=k+1;
                }
        }
        else {
                for(i=0;i<10;i++) {
                    if(ray2[i] != k+1) {
                        printf("value not preserved\n");
                    }
                        ray2[i]=ray2[i]+nhalf;
                }
        }
    x=sub_3(10);

#pragma omp critical
        {
    fflush(stdout);
    printf("(c) thread %d ",k);
    printf("n= %d x= %g\n",n,x);
    }
}

FLT sub_3(int in) {
        int i;
        FLT x;
        x=0.0;
        for(i=0;i<in;i++) {
                x=x+ray2[i];
        }
        return (x);
}

void sub_4() {
#pragma omp critical
        {
        fflush(stdout);
        printf("address of pointer ray2 %d, address held in ray2 %d, value held in ray2[0] %d\n",
               &ray2,ray2,ray2[0]);
        }
}
main() {
        int i,j,k,max_threads;
        FLT pi4,xvect,yvect;

        max_threads=omp_get_max_threads();

        explain("a complex test of thread private with a do loop");
        explain("we pass in the array of structures ");
        explain("the structure is a 2d vector. we sum of all ");
        explain("the vectors to get us back to the real axis ");
        explain("we multiply the vectors times (thread+1) ");
        j=-1;
        itype[0].ypart=0;
        itype[0].xpart=1;
        pi4=pi/4.0;
        for(i=1;i<=8;i++) {
                yvect=sin(i*pi4);
                xvect=cos(i*pi4);
                itype[i].ypart=yvect-itype[i-1].ypart;
                itype[i].xpart=xvect-itype[i-1].xpart;
        }

#pragma omp parallel for copyin(itype) schedule(static,1)
        for(i=1;i<=max_threads;i++) {
                sub_1();
        }


        explain("a complex test of thread private with a do loop");
        explain("we access a threadprivate pointer allocated");
        explain("inside of the thread");

        j=max_threads*2;
        k=j/2;
        ray2=0;
#pragma omp parallel for copyin(ray2) schedule(static,1)
        for(i=1;i<=j;i++) {
                sub_2(i,k);
        }
#pragma omp parallel
        {
                if(ray2) {
                        printf("thread %d deallocating ray2 at %d with value=%d\n",omp_get_thread_num(),ray2,ray2[0]);
                        free(ray2);
                        ray2=0;
                }
        }
        ray2=(int *)malloc(10*sizeof(int));
        ray2[0]=1234;
#pragma omp parallel for copyin(ray2) schedule(static,1)
        for(i=1;i<=k;i++) {
                sub_4();
        }


}

void explain(char astr[]){
        printf("%s\n",astr);
}
```

Single and subsections

Link to example source.

Link to example source in Fortran.

This is a relatively simple example that shows how you might use OpenMP so that each thread operates on a subsection of an array without using a for loop. It also shows a use for the single directive.

This prorgam allocates an array and calls two subroutines. The first subroutine puts values into the array and the second checks the values. The array is allocated and values are put into the array inside of a parallel region. The subroutine that checks the values is called from a serial region. The number of values in the array is npoints=2*3*4*5*7=850.

The allocation of the array is done inside a region protected by a single directive. This is done so that the allocation is only done one time.

```
main () {
      int i,iam,np,npoints,ipoints;
      float *x;
      x=0;
#pragma omp parallel shared(x,npoints,np) default(none) private(iam,ipoints)
      {
              npoints=2*3*4*5*7;
              iam = omp_get_thread_num();
              np = omp_get_num_threads();
#pragma omp single
              {
                      if(x !=0)printf("single fails\n");
                      x=(float *)malloc((unsigned)npoints*sizeof(float));
                      x--; /* this line is used to set the starting point for our array to x[1] */
15               }
16 #pragma omp barrier
17               ipoints = npoints/np;
18               subdomain(x,iam,ipoints);
19       }
20
21       printf("outside of the parallel region\n");
22      for(i=0;i< np;i++) {
23         ipoints = npoints/np;
24         pdomain(x,i,ipoints);
25      }
26 }
```

The parallel region is from lines 7 to 18. X is null coming into this region. The allocation is done on line 13 and should only be done one time because of the single directive. If for some reason, the allocation is done a second time, the test on line 12 will fail and an error message will be printed. This would only happen if the single directive is not working correctly.

The barrier is used on line 16 to ensure that the array is allocated before it is used.

Line 17 gives the number of points that each thread will set in the routine subdomain. Subdomain is shown next.

```
void subdomain( float *x, int iam,int ipoints) {
    int ibot,itop,i;
    int sum;
    ibot=(iam)*ipoints+1;
    itop=ibot+ipoints-1;
    for(i=ibot;i<=itop;i++)
        x[i]=iam;
    sum=0;
    for(i=ibot;i<=itop;i++)
        sum=sum+x[i];
#pragma omp critical
```

```
    printf(" iam= %d doing %d %d %d \n",iam,ibot,itop,sum/ipoints);
}
```

This routine is called with the array (x) , the thread id (iam), and the number of points a thread is to initialize (ipoints). It first calculates the lower and upper bound of the threads region and then puts the thread id in to its poition of the array. Finally, it prints the average value that is put into the array. This should be the thread id.

Back in the main program we call the routine pdomain in a for loop. Pdomain is called with the for loop counter as input, not thread id. Pdomain checks that all of the elements in a particular subsection contains the same value. The value contained in each subsection is printed. There should be a subsection that contains each thread id.

The following is output from this program run using 4 threads

```
 iam= 0 doing 1 210 0
 iam= 3 doing 631 840 3
 iam= 2 doing 421 630 2
 iam= 1 doing 211 420 1
outside of the parallel region
 section= 0 is from 1 to  210 and contains 0
 section= 1 is from 211 to  420 and contains 1
 section= 2 is from 421 to  630 and contains 2
 section= 3 is from 631 to  840 and contains 3
```

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void subdomain( float *x, int iam,int ipoints);
void   pdomain( float *x, int iam,int ipoints);


void subdomain( float *x, int iam,int ipoints) {
    int ibot,itop,i;
    int sum;
    ibot=(iam)*ipoints+1;
    itop=ibot+ipoints-1;
    for(i=ibot;i<=itop;i++)
        x[i]=iam;
    sum=0;
    for(i=ibot;i<=itop;i++)
        sum=sum+x[i];
#pragma omp critical
    printf(" iam= %d doing %d %d %d \n",iam,ibot,itop,sum/ipoints);
}

void pdomain( float *x, int iam,int ipoints) {
    int ibot,itop,i;
    float y;
    int sum;
    ibot=(iam)*ipoints+1;
    itop=ibot+ipoints-1;
    printf(" section= %d is from %d to  %d",iam,ibot,itop);
    y=x[ibot];
    for(i=ibot;i<=itop;i++)
        if(y != x[i]){
                y=x[i];
        }
    if(y == x[ibot]) {
        printf(" and contains %g\n",y);
    }
    else
        printf(" failed\n");
}

main () {
        int i,iam,np,npoints,ipoints;
        float *x;
        x=0;
#pragma omp parallel shared(x,npoints,np) default(none) private(iam,ipoints)
        {
                npoints=2*3*4*5*7;
                iam = omp_get_thread_num();
                np = omp_get_num_threads();
#pragma omp single
                {
                        if(x !=0)printf("single fails\n");
                        x=(float *)malloc((unsigned)npoints*sizeof(float));
                        x--;
                }
#pragma omp barrier
                ipoints = npoints/np;
                subdomain(x,iam,ipoints);
        }
        printf("outside of the parallel region\n");
    for(i=0;i < np;i++) {
       ipoints = npoints/np;
       pdomain(x,i,ipoints);
    }
}
```

```
module stuff
 contains
    subroutine subdomain( x,  iam, ipoints)
        real x(:)
        integer iam
        integer ipoints
        integer ibot,itop,i
        integer sum
        ibot=(iam)*ipoints+1
        itop=ibot+ipoints-1
        do i=ibot,itop
            x(i)=iam
        enddo
        sum=0
        do i=ibot,itop
            sum=sum+x(i)
        enddo
    !$omp critical
        write(*,*)" iam= ",iam," doing ",ibot,itop,sum/ipoints
    !$omp end critical
    end subroutine

    subroutine pdomain( x,  iam, ipoints)
        real x(:)
        integer iam,ipoints
        integer ibot,itop,i
        real  y
        ibot=(iam)*ipoints+1
        itop=ibot+ipoints-1
        write(*,*)" section= ",iam,"is from ",ibot," to ",itop
        y=x(ibot)
        do i=ibot,itop
            if(y .ne. x(i))y=x(i)
        enddo
        if(y .eq. x(ibot)) then
            write(*,*)" and contains",y
        else
            write(*,*)" failed"
        endif
    end subroutine
end module

program mymain
    use stuff
    integer omp_get_thread_num,omp_get_num_threads
    integer i,iam,np,npoints,ipoints
    real, allocatable :: x(:)
!   x=0
!$omp parallel shared(x,npoints,np) default(none) private(iam,ipoints)
        npoints=2*3*4*5*7
        iam = omp_get_thread_num()
        np = omp_get_num_threads()
!$omp single
            if(allocated(x))write(*,*)"single fails"
            allocate(x(npoints))
!$omp end single
!$omp barrier
        ipoints = npoints/np
        write(*,*)ipoints,iam
        call subdomain(x,iam,ipoints)
!$omp end parallel
    write(*,*)"outside of the parallel region"
    do i=0,np-1
       ipoints = npoints/np
       call pdomain(x,i,ipoints)
    enddo
end program
```

Sorting, threadprivate with pointers to derived types

Link to example source.

This example shows how to do a recursive merge sort using OpenMP. Each thread is assigned a section of an array for which to perform a recursive merge sort. The sorted subsections are then merged. It also shows how threadprivate can be used to pass values to subroutines. This can be important for recursive routines.

The array being sorted is actually the derived type:

```
type thefit
    sequence
    real val
    integer index
end type thefit
```

We are sorting an array of type THEFIT using the key "val." Index will contain the original index of "val" in the array that is being sorted.

The program is given here with line numbers for reference.

Our program starts on line 127. It allocates an array 32 elements long and fills it with random numbers (137-142).

We split the array into sections and use the **!$omp sections** directive to sort the sections (157-166). We then merge the sections and print the final merged list (173-190).

The sorting routine is a standard Merge-Sort (8-126) and is described in in Moret and Shapiro, *Algorithms from P to NP*, *Vol 1*. The routine SORT allocates a temporary work array (18) and there is a pointer to the input array (20). These are used by the routines RecMergeSort and Merge. They are accessible to these routines because they are in common.

At first glance the sorting routine does not look like it contains any OpenMP. But if we look at the definition of "work" and "a" we see that these two pointers to derived types are threadprivate (15,30,46,100). We have used the threadprivate pointer "a" to point to a subsection of an array. Each thread gets its own pointer "a" so that it can access its own subsection of the derived type input array. Each thread allocates its own copy of the work array for use in the rest of the algorithm.

We have shown that we can use threadprivate with pointers to arrays of derived types. These can be used to pass values between subroutines, with each thread having its own copy.

The final output of this program is:

```
 ending list
   20 0.94798243
   16 0.93174280
   30 0.92679580
   28 0.88469052
   12 0.88409471
   31 0.86152124
   18 0.85193380
   25 0.76048082
   23 0.70104372
   11 0.69971883
   29 0.69236451
    1 0.67201113
   26 0.65388730
    3 0.63397812
   17 0.54539450
   13 0.53869861
   22 0.47838681
    7 0.47571510
    9 0.44888480
    5 0.43045502
   15 0.41719050
```

```
  32 0.41321742
  24 0.39106920
  14 0.37801930
  27 0.34327822
   4 0.23692870
  19 0.21749190
  21 0.20226842
   8 0.16754910
   2 0.13908743
   6 0.10045713
  10 0.02444941
```

```
module galapagos
    type thefit
      sequence
      real val
      integer index
    end type thefit
end module

module sort_mod

contains

  subroutine Sort(Ain, n)
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer n
    type(thefit), target:: ain(n)
    allocate(work(n))
    nullify(a)
    a=>ain
    call RecMergeSort(1,n)
    deallocate(work)
    return
  end subroutine Sort

  recursive subroutine RecMergeSort(left, right)
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer,intent(in):: left,right
    integer  middle
    if (left < right) then
        middle = (left + right) / 2
        call RecMergeSort(left,middle)
        call RecMergeSort(middle+1,right)
        call Merge(left,middle-left+1,right-middle)
    endif
    return
  end subroutine RecMergeSort

  subroutine Merge(s, n, m)
```

```
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer s,n,m
    integer i,  j, k, t, u
    k = 1
    t = s + n
    u = t + m
    i = s
    j = t
    if ((i < t) .and. (j < u))then
        do while ((i < t) .and. (j < u))
            if (A(i)%val .ge. A(j)%val)then
                work(k) = A(i)
                i = i + 1
                k = k + 1
            else
                work(k) = A(j)
                j = j + 1
                k = k + 1
            endif
         enddo
    endif
    if(i < t )then
        do while (i < t )
            work(k) = A(i)
            i = i + 1
            k = k + 1
        enddo
    endif
    if(j < u)then
        do while (j < u )
            work(k) = A(j)
            j = j + 1
            k = k + 1
        enddo
    endif
    i = s
! the next line is not in moret & shapiro's book
    k=k-1
    do j = 1 , k
        A(i) = work(j)
        i = i + 1
    enddo
    return
  end subroutine Merge

! this subroutine takes two sorted lists of type(thefit) and merges them
! input d1(n) , d2(m)
! output out(n+m)
subroutine merge2(d1,n,d2,m,out)
  use galapagos
  implicit none
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
  integer n,m
  type(thefit),intent (in):: d1(n),d2(m)
  type(thefit), intent (out):: out(n+m)
  integer i,j,k
  i=1
  j=1
  do k=1,n+m
    if(i.gt.n)then
      out(k)=d2(j)
      j=j+1
    elseif(j.gt.m)then
      out(k)=d1(i)
      i=i+1
    else
      if(d1(i)%val .gt. d2(j)%val)then
        out(k)=d1(i)
        i=i+1
      else
        out(k)=d2(j)
        j=j+1
      endif
    endif
   enddo
  return
  end subroutine merge2
end module sort_mod


program test
    use galapagos
    use sort_mod
    implicit none
    integer i,j,k,m,di,k1,k2
```

```fortran
    integer OMP_GET_MAX_THREADS
    integer, allocatable :: kstart(:),kend(:)
    type(thefit),allocatable :: data(:),output1(:),output2(:)
    write(*,*)"sort in fortran"
    i=32
    allocate(data(i))

    do j=1,i
        call random_number(data(j)%val)
        data(j)%index=j
        write(*,*)data(j)%index,data(j)%val
    enddo
    write(*,*)
!
    m=4
    di=i/m
    allocate(kstart(m),kend(m))
    kstart(1)=1
    kend(1)=di
    do j=2,m
        kstart(j)=kend(j-1)+1
        kend(j)=kstart(j)+di
    enddo
    kend(m)=i
    write(*,"(8i5)")kstart
    write(*,"(8i5)")kend
!$OMP PARALLEL SECTIONS
!$OMP SECTION
                call sort(data(kstart(1):kend(1)), kend(1)-kstart(1)+1 )
!$OMP SECTION
                call sort(data(kstart(2):kend(2)), kend(2)-kstart(2)+1 )
!$OMP SECTION
                call sort(data(kstart(3):kend(3)), kend(3)-kstart(3)+1 )
!$OMP SECTION
                call sort(data(kstart(4):kend(4)), kend(4)-kstart(4)+1 )
!$OMP END PARALLEL SECTIONS
    do k=1,m
        write(*,*)"start of section ",k
            do j=kstart(k),kend(k)
                write(*,"(i5,1x,f10.8)")data(j)%index,data(j)%val
            enddo
    enddo
!$OMP PARALLEL SECTIONS
!$OMP SECTION
    k1=kend(2)-kstart(1)+1
    allocate(output1(k1))
    call merge2(data(kstart(1):kend(1)),kend(1)-kstart(1)+1, &
                data(kstart(2):kend(2)),kend(2)-kstart(2)+1,output1)
!$OMP SECTION
    k2=kend(4)-kstart(3)+1
    allocate(output2(k2))
    call merge2(data(kstart(3):kend(3)),kend(3)-kstart(3)+1, &
                data(kstart(4):kend(4)),kend(4)-kstart(4)+1,output2)
!$OMP END PARALLEL SECTIONS
    call merge2(output1,k1,output2,k2,data)

    write(*,*)"ending list "
    do j=1,i
        write(*,"(i5,1x,f10.8)")data(j)%index,data(j)%val
    enddo
end program
```

```
module galapagos
    type thefit
      sequence
      real val
      integer index
    end type thefit
end module

module sort_mod

contains

  subroutine Sort(Ain, n)
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer n
    type(thefit), target:: ain(n)
    allocate(work(n))
    nullify(a)
    a=>ain
    call RecMergeSort(1,n)
    deallocate(work)
    return
  end subroutine Sort

  recursive subroutine RecMergeSort(left, right)
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer,intent(in):: left,right
    integer  middle
    if (left < right) then
        middle = (left + right) / 2
        call RecMergeSort(left,middle)
        call RecMergeSort(middle+1,right)
        call Merge(left,middle-left+1,right-middle)
    endif
    return
  end subroutine RecMergeSort

  subroutine Merge(s, n, m)
```

```
    use galapagos
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
    integer s,n,m
    integer i,  j, k, t, u
    k = 1
    t = s + n
    u = t + m
    i = s
    j = t
    if ((i < t) .and. (j < u))then
        do while ((i < t) .and. (j < u))
            if (A(i)%val .ge. A(j)%val)then
                work(k) = A(i)
                i = i + 1
                k = k + 1
            else
                work(k) = A(j)
                j = j + 1
                k = k + 1
            endif
         enddo
    endif
    if(i < t )then
        do while (i < t )
            work(k) = A(i)
            i = i + 1
            k = k + 1
        enddo
    endif
    if(j < u)then
        do while (j < u )
            work(k) = A(j)
            j = j + 1
            k = k + 1
        enddo
    endif
    i = s
! the next line is not in moret & shapiro's book
    k=k-1
    do j = 1 , k
        A(i) = work(j)
        i = i + 1
    enddo
    return
  end subroutine Merge

! this subroutine takes two sorted lists of type(thefit) and merges them
! input d1(n) , d2(m)
! output out(n+m)
subroutine merge2(d1,n,d2,m,out)
  use galapagos
  implicit none
    type(thefit), pointer :: work(:)
    type(thefit), pointer :: a(:)
    common /bonk/ a,work
!$OMP THREADPRIVATE (/bonk/)
  integer n,m
  type(thefit),intent (in):: d1(n),d2(m)
  type(thefit), intent (out):: out(n+m)
  integer i,j,k
  i=1
  j=1
  do k=1,n+m
    if(i.gt.n)then
      out(k)=d2(j)
      j=j+1
    elseif(j.gt.m)then
      out(k)=d1(i)
      i=i+1
    else
      if(d1(i)%val .gt. d2(j)%val)then
        out(k)=d1(i)
        i=i+1
      else
        out(k)=d2(j)
        j=j+1
      endif
    endif
   enddo
  return
  end subroutine merge2
end module sort_mod


program test
    use galapagos
    use sort_mod
    implicit none
    integer i,j,k,m,di,k1,k2
```

```fortran
    integer OMP_GET_MAX_THREADS
    integer, allocatable :: kstart(:),kend(:)
    type(thefit),allocatable :: data(:),output1(:),output2(:)
    write(*,*)"sort in fortran"
    i=32
    allocate(data(i))

    do j=1,i
        call random_number(data(j)%val)
        data(j)%index=j
        write(*,*)data(j)%index,data(j)%val
    enddo
    write(*,*)
!
    m=4
    di=i/m
    allocate(kstart(m),kend(m))
    kstart(1)=1
    kend(1)=di
    do j=2,m
        kstart(j)=kend(j-1)+1
        kend(j)=kstart(j)+di
    enddo
    kend(m)=i
    write(*,"(8i5)")kstart
    write(*,"(8i5)")kend
!$OMP PARALLEL SECTIONS
!$OMP SECTION
            call sort(data(kstart(1):kend(1)), kend(1)-kstart(1)+1 )
!$OMP SECTION
            call sort(data(kstart(2):kend(2)), kend(2)-kstart(2)+1 )
!$OMP SECTION
            call sort(data(kstart(3):kend(3)), kend(3)-kstart(3)+1 )
!$OMP SECTION
            call sort(data(kstart(4):kend(4)), kend(4)-kstart(4)+1 )
!$OMP END PARALLEL SECTIONS
    do k=1,m
            write(*,*)"start of section ",k
        do j=kstart(k),kend(k)
            write(*,"(i5,1x,f10.8)")data(j)%index,data(j)%val
        enddo
    enddo
!$OMP PARALLEL SECTIONS
!$OMP SECTION
    k1=kend(2)-kstart(1)+1
    allocate(output1(k1))
    call merge2(data(kstart(1):kend(1)),kend(1)-kstart(1)+1, &
                data(kstart(2):kend(2)),kend(2)-kstart(2)+1,output1)
!$OMP SECTION
    k2=kend(4)-kstart(3)+1
    allocate(output2(k2))
    call merge2(data(kstart(3):kend(3)),kend(3)-kstart(3)+1, &
                data(kstart(4):kend(4)),kend(4)-kstart(4)+1,output2)
!$OMP END PARALLEL SECTIONS
    call merge2(output1,k1,output2,k2,data)

    write(*,*)"ending list "
    do j=1,i
        write(*,"(i5,1x,f10.8)")data(j)%index,data(j)%val
    enddo
end program
```

Sorting, threadprivate with pointers to derived types

Link to example source.

This example shows how to do a recursive merge sort using OpenMP. Each thread is assigned a section of an array for which to perform a recursive merge sort. The sorted subsections are then merged. It also shows how threadprivate can be used to pass values to subroutines. This can be important for recursive routines.

The array being sorted is actually the derived type:

```
typedef struct {
  float val;
  int index;
} THEFIT;
```

We are sorting an array of type THEFIT using the key "val." Index will contain the original index of "val" in the array that is being sorted.

The program is given here with line numbers for reference.

The routine vector (141-150) allocates an array of type THEFIT and sets the first indices of the array equal to 1. The routine free_vector (151-154) deallocates arrays of type THEFIT.

Our program allocates an array 32 elements long and fills it with random numbers (21-27).

We split the array into sections and use the **#pragma omp sections** directive to sort the sections (29-38). We then merge the sections and print the final merged list (50-53).

The sorting routine is a standard Merge-Sort (56-111) and is described in Moret and Shapiro, *Algorithms from P to NP, Vol 1*. The routine SORT allocates a temporary work array (57) and there is a pointer to the input array (58). These are used by the routines RecMergeSort and Merge. They are accessible to these routines because they are global.

At first glance the sorting routine does not look like it contains any OpenMP. But if we look at the definition of "work" and "a" (8-10) we see that these two pointers to derived types are threadprivate. We have used the threadprivate pointer "a" to point to a subsection of an array. Each thread gets its own pointer "a" so that it can access its own subsection of the derived type input array. Each thread allocates its own copy of the work array for use in the rest of the algorithm.

We have shown that we can use threadprivate with pointers to arrays of derived types. These can be used to pass values between subroutines, with each thread having its own copy.

The final output of this program is:

```
15  0.9834372
 5  0.9476279
31  0.9315229
26  0.9172045
32  0.8699211
21  0.8229514
30  0.7854022
20  0.7802369
19  0.7671388
17  0.7656819
 7  0.7022312
18  0.6464732
23  0.6254767
29  0.6067584
16  0.5353979
 4  0.5345339
27  0.5197600
 1  0.5138701
 9  0.4947734
```

```
28  0.4011542
12  0.3896471
14  0.3680707
25  0.3469011
24  0.3146848
 3  0.3086515
13  0.2772258
 8  0.2264307
 2  0.1757413
 6  0.1717363
22  0.1519323
10  0.1247203
11  0.0838988
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
typedef struct {
  float val;
  int index;
} THEFIT;

    THEFIT *work;
    THEFIT *a;
#pragma omp threadprivate (work,a)


void RecMergeSort(int left, int right);
void Sort(THEFIT *Ain, int n);
void Merge(int s, int n, int m);
void merge2(THEFIT *d1,int n,THEFIT *d2,int m,THEFIT *out);

THEFIT *vector(int nl, int nh);
void free_vector(THEFIT *v, int nl);

int main() {
    THEFIT *data,*output;
    int i,j,k,k1,k2,k3,k4;
    printf("sort in c\n");
    i=32;
    data=vector(1,i);
    for(j=1;j<=i;j++) {
        data[j].index=j;
        data[j].val=(float)rand()/(float)RAND_MAX;
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    k=i/2;
    k1=k+1;
    k2=(i-k1)+1;
#pragma omp sections
    {
#pragma omp section
    Sort(&data[1],k);
#pragma omp section
    Sort(&data[k1],k2);
    }
    for(j=1;j<=k;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    for(j=k1;j<=i;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    output=vector(1,i);
    merge2(&data[1],k,&data[k1],k2,&output[1]);
    for(j=1;j<=i;j++) {
        printf("%2d %10.7f\n",output[j].index,output[j].val);
    }
    return 0;
}

 void Sort(THEFIT *Ain, int n){
    work=vector(1,n);
    a=Ain-1;
    RecMergeSort(1,n);
    free_vector(work,1);
 }

  void RecMergeSort(int left, int right) {
    int  middle;
    if (left < right) {
        middle = (left + right) / 2;
        RecMergeSort(left,middle);
        RecMergeSort(middle+1,right);
        Merge(left,middle-left+1,right-middle);
    }
  }

  void Merge(int s, int n, int m) {
    int i,  j, k, t, u;
    k = 1;
    t = s + n;
    u = t + m;
    i = s;
    j = t;
    if ((i < t) && (j < u)){
        while ((i < t) && (j < u)){
            if (a[i].val >= a[j].val){
                work[k] = a[i];
                i = i + 1;
                k = k + 1;
            } else {
                work[k] = a[j];
                j = j + 1;
                k = k + 1;
            }
         }
     }
     if(i < t ){
         while (i < t ) {
             work[k] = a[i];
             i = i + 1;
             k = k + 1;
         }
     }
     if(j < u){
         while (j < u ) {
             work[k] = a[j];
             j = j + 1;
             k = k + 1;
         }
     }
     i = s;
     k=k-1;
     for( j = 1; j<= k; j++) {
         a[i] = work[j];
         i = i + 1;
     }
   }

/*
! this subroutine takes two sorted lists of type(THEFIT) and merges them

! input d1(1:n) , d2(1:m)
! output out(1:n+m)
*/
void merge2(THEFIT *d1,int n,THEFIT *d2,int m,THEFIT *out) {
  int i,j,k;
  i=1;
  j=1;
  d1--; d2--; out--;
  for( k=1; k<=n+m;k++) {
    if(i > n){
      out[k]=d2[j];
      j=j+1;
    }
    else if(j > m){
      out[k]=d1[i];
      i=i+1;
    } else {
      if(d1[i].val > d2[j].val){
        out[k]=d1[i];
        i=i+1;
      } else {
        out[k]=d2[j];
        j=j+1;
      }
    }
   }
  }


THEFIT *vector(int nl, int nh)
{
        THEFIT *v;

        v=(THEFIT *)malloc((unsigned) (nh-nl+1)*sizeof(THEFIT));
        if (!v) {
            printf("allocation failure in ivector()\n");
                exit(1);
    }
        return v-nl;
}


void free_vector(THEFIT *v, int nl)

{

 free((char*) (v+nl));

}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
typedef struct {
  float val;
  int index;
} THEFIT;

    THEFIT *work;
    THEFIT *a;
#pragma omp threadprivate (work,a)


void RecMergeSort(int left, int right);
void Sort(THEFIT *Ain, int n);
void Merge(int s, int n, int m);
void merge2(THEFIT *d1,int n,THEFIT *d2,int m,THEFIT *out);

THEFIT *vector(int nl, int nh);
void free_vector(THEFIT *v, int nl);

int main() {
    THEFIT *data,*output;
    int i,j,k,k1,k2,k3,k4;
    printf("sort in c\n");
    i=32;
    data=vector(1,i);
    for(j=1;j<=i;j++) {
        data[j].index=j;
        data[j].val=(float)rand()/(float)RAND_MAX;
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    k=i/2;
    k1=k+1;
    k2=(i-k1)+1;
#pragma omp sections
    {
#pragma omp section
    Sort(&data[1],k);
#pragma omp section
    Sort(&data[k1],k2);
    }
    for(j=1;j<=k;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    for(j=k1;j<=i;j++) {
        printf("%d %g\n",data[j].index,data[j].val);
    }
    printf("\n\n");
    printf("\n\n");
    output=vector(1,i);
    merge2(&data[1],k,&data[k1],k2,&output[1]);
    for(j=1;j<=i;j++) {
        printf("%2d %10.7f\n",output[j].index,output[j].val);
    }
    return 0;
}

 void Sort(THEFIT *Ain, int n){
    work=vector(1,n);
    a=Ain-1;
    RecMergeSort(1,n);
    free_vector(work,1);
 }

  void RecMergeSort(int left, int right) {
    int  middle;
    if (left < right) {
        middle = (left + right) / 2;
        RecMergeSort(left,middle);
        RecMergeSort(middle+1,right);
        Merge(left,middle-left+1,right-middle);
    }
  }

  void Merge(int s, int n, int m) {
    int i,  j, k, t, u;
    k = 1;
    t = s + n;
    u = t + m;
    i = s;
    j = t;
    if ((i < t) && (j < u)){
        while ((i < t) && (j < u)){
            if (a[i].val >= a[j].val){
                work[k] = a[i];
                i = i + 1;
                k = k + 1;
            } else {
                work[k] = a[j];
                j = j + 1;
                k = k + 1;
            }
         }
    }
    if(i < t ){
        while (i < t ) {
            work[k] = a[i];
            i = i + 1;
            k = k + 1;
        }
    }
    if(j < u){
        while (j < u ) {
             work[k] = a[j];
             j = j + 1;
             k = k + 1;
         }
    }
    i = s;
    k=k-1;
    for( j = 1; j<= k; j++) {
        a[i] = work[j];
        i = i + 1;
    }
  }
/*
! this subroutine takes two sorted lists of type(THEFIT) and merges them
! input d1(1:n) , d2(1:m)
! output out(1:n+m)
*/
void merge2(THEFIT *d1,int n,THEFIT *d2,int m,THEFIT *out) {
  int i,j,k;
  i=1;
  j=1;
  d1--; d2--; out--;
  for( k=1; k<=n+m;k++) {
    if(i > n){
      out[k]=d2[j];
      j=j+1;
    }
    else if(j > m){
      out[k]=d1[i];
```

```
          i=i+1;
        } else {
          if(d1[i].val > d2[j].val){
            out[k]=d1[i];
            i=i+1;
          } else {
            out[k]=d2[j];
            j=j+1;
          }
        }
      }
    }

THEFIT *vector(int nl, int nh)
{
        THEFIT *v;

        v=(THEFIT *)malloc((unsigned) (nh-nl+1)*sizeof(THEFIT));
        if (!v) {
            printf("allocation failure in ivector()\n");
                exit(1);
    }
        return v-nl;
}

void free_vector(THEFIT *v, int nl)

{

 free((char*) (v+nl));

}
```

Atomic operation to update an array index

The following example is an expansion on example A.12 from the standards document, *Using the atomic Directive*. It is included here because in some early implementations of OpenMP atomic did not work correctly on this example. The output from this program is:

```
0 9 0
1 8 1
2 7 4
3 6 9
4 5 16
5 4 25
6 3 36
7 2 49
8 1 64
9 0 81
```

The example avoids race conditions (simultaneous updates of an element of x) by multiple threads by using the atomic directive.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
main() {
    float *x,*y,*work1,*work2;
    int *index;
    int n,i;
    n=10;
    x=(float*)malloc(n*sizeof(float));
    y=(float*)malloc(n*sizeof(float));
    work1=(float*)malloc(n*sizeof(float));
    work2=(float*)malloc(n*sizeof(float));
    index=(int*)malloc(10*sizeof(float));
    for( i=0;i < n;i++) {
        index[i]=(n-i)-1;
        x[i]=0.0;
        y[i]=0.0;
        work1[i]=i;
        work2[i]=i*i;
    }
#pragma omp parallel for  shared(x,y,index,n)
    for( i=0;i< n;i++) {
#pragma omp atomic
        x[index[i]] += work1[i];
        y[i] += work2[i];
    }
        for( i=0;i < n;i++)
                printf("%d %g %g\n",i,x[i],y[i]);
}
```

The advantage of using the atomic directive in this example is that it allows updates of two different elements of x to occur in parallel. If a critical directive were used instead, that all updates to elements of x would be executed serially (though not in any guaranteed order).

RUNTIME scheduling, FFTs and performance issues

Link to example source.

This example illustrates the usage of RUNTIME scheduling. It also shows the effects of different chunk sizes on static scheduling. It also discusses some issues associated with OpenMP performance, that is, they way a program is compiled can effect run times. In particular, it shows that compiling with the OpenMP compiler (when not absolutely required) can hurt performance. These illustrations are done in the context of a kernel style program. That is, the example program performs the type of calculations that are done in a "real" program but it does not contain all of the details of the original program.

This example is the kernel of an optical propagation program. It does a series of 2d Fourier transforms, ffts, followed by a multiplication. It is modeled after the AFWL program HELP or High Energy Laser Propagation. It does the 2d fft by first doing a collection of 1d ffts, one for each column of data, then a transpose, followed by a second collection of 1d ffts. After the 2d fft is performed the resulting array is multiplied by a scaling factor.

The outline of our program is

For a matrix "a" size of "size"

```
do n=1,20
        do a collection of "size" 1d ffts on a column of "a"
        do a matrix transpose
        do a collection of "size" 1d ffts on a column of "a"
        multiply "a" by some factor
enddo
```

Each of the operations shown above is timed. We sum and report the times.

The timing routine used in this program, ccm_time, returns values with a precision of milliseconds. It returns time is seconds from the beginning of the year in which the program was started. The routine is portable across Fortran 90. It was written as part of the software delivered under *High Performance Computing Modernization Program, Task Number: CE 019, Title: SPMD Collective Communication*

Module.

Parallelism is obtained by applying OpenMP directives to the outer do loops.

For the FFTs we use the routine four1. Four1 does a fft on a column of data. We call four1 in a do loop, with each iteration calling four1 on a different column of data. The ffts for each column are independent so we can do them in parallel. Thus we use the, PARALLEL DO, directive with, SCHEDULE (RUNTIME) clause.

```
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do i=1,size
                 call four1(a(:,i),size,isign)
            enddo
```

When the SCHEDULE (RUNTIME) clause is used the actual scheduling for the loop is determined by the setting of the environment variable OMP_SCHEDULE. For example, run the loop using STATIC,64 scheduling we would do a:

```
setenv OMP_SCHEDULE "STATIC,64"
```

before the program is run.

We have a similar OpenMP directive for the transpose operation.

```
!$OMP PARALLEL DO SCHEDULE (RUNTIME) PRIVATE(i,j,k,tmp)
            do k=1,size
                i=k
                do j=i,size
                    tmp=a(i,j)
                    a(i,j)=a(j,i)
                    a(j,i)=tmp
                enddo
             enddo
```

In the code, we next have a second do loop in which we call four1.

Finally, we have a OpenMP directive for the multiplication operation.

```
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do j=1,size
               do i=1,size
                  a(i,j)=factor*a(i,j)
               enddo
            enddo
```

The routines four1 was taken from the book "*Numerical Recipes in Fortran, 1st addition*." However, the authors of that book derived their routine from the routine, fourn, that was in the AFWL program HELP. The original routine, fourn, contained many additional options. The algorithm in four1 is a subset of the algorithm in fourn, down to the variable names.

Disclaimer:
For a production code you would most likely use a vendor supplied library to do the fft instead of a hand written one.

The program was run on three different machines, one "old" machine and two preproduction machines. One of the new machines had only 2 shared memory processors so it was run using 2 threads. The other machines were run using up to 8 threads.

The program was compiled as a serial application called "one" and as a OpenMP application called "two". It was then run using a script similar to:

```
#!/bin/csh
./one
setenv OMP_NUM_THREADS 2
setenv OMP_SCHEDULE "STATIC,2"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,4"
```

```
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,8"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,16"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,32"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,63"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
setenv OMP_SCHEDULE "STATIC,64"
echo $OMP_NUM_THREADS" "$OMP_SCHEDULE
./two
```

The output from a single run was similar to the following:

```
4 STATIC,32
(    0.9882567199    ,     0.000000000    )
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
(    0.9882567199    ,    0.2382016198E-14)
 number of  transforms 20
      fft1 time=    0.7480
 transpose time=    0.5350
      fft2 time=    0.7450
   scaling time=    0.0930
    total time =    2.1210 for matrix of size  1024
 THREADS   =  4
```

The numbers in parentheses

```
(    0.9882567199    ,     0.000000000    )
```

and

```
(    0.9882567199    ,    0.2382016198E-14)
```

are the first element, a complex number, of the input array and the same element after the calculation is completed. The scaling parameter is set so that these two numbers should be the same.

We report the total number of 2d transforms, the total time spent in the first loop calling four1, the time spent in the loop performing the transpose, the second fft time, and the time spent in the scaling loop.

The timings for the three machines are reported below with comments following.

Timings for the optics kernel
Machine A

OpenMP parameters		**TIMES**				
Threads	**Scheduling**	**fft 1**	**transpose**	**fft 2**	**scaling**	**total**
1	SERIAL	8.291	1.898	8.294	0.160	18.643
2	STATIC,2	4.463	2.319	4.495	0.182	11.459
2	STATIC,4	4.453	1.659	4.452	0.161	10.725

2	STATIC,8	4.451	1.599	4.449	0.154	10.653
2	STATIC,16	4.451	1.193	4.446	0.145	10.235
2	STATIC,32	4.455	1.123	4.443	0.143	10.164
2	STATIC,63	4.517	1.339	4.512	0.144	10.512
2	STATIC,64	4.452	1.294	4.454	0.141	10.341
2	STATIC,1024	8.870	1.902	8.863	0.160	19.795

Timings for the optics kernel Machine B

OpenMP parameters		**TIMES**				
Threads	**Scheduling**	**fft 1**	**transpose**	**fft 2**	**scaling**	**total**
1	SERIAL	6.706	3.141	6.690	0.595	17.132
2	STATIC,2	4.177	5.270	4.219	0.234	13.900
2	STATIC,4	4.068	2.564	4.252	0.226	11.110
2	STATIC,8	4.076	2.551	4.203	0.230	11.060
2	STATIC,16	4.065	2.150	4.159	0.221	10.595
2	STATIC,32	4.062	2.016	4.163	0.220	10.461

2	STATIC,63	4.152	1.560	4.191	0.222	10.125
2	STATIC,64	4.067	1.998	4.145	0.228	10.438
2	STATIC,1024	8.432	3.266	8.412	0.591	20.701
4	STATIC,2	2.037	2.581	2.183	0.099	6.900
4	STATIC,4	1.974	2.452	2.158	0.080	6.664
4	STATIC,8	1.944	1.327	2.120	0.078	5.469
4	STATIC,16	1.974	1.175	2.108	0.076	5.333
4	STATIC,32	1.969	1.158	2.108	0.076	5.311
4	STATIC,63	2.049	1.215	2.170	0.081	5.515
4	STATIC,64	1.952	1.254	2.122	0.082	5.410
4	STATIC,1024	8.427	3.302	8.393	0.573	20.695
6	STATIC,2	1.391	1.791	1.463	0.084	4.729
6	STATIC,4	1.330	1.642	1.467	0.067	4.506
6	STATIC,8	1.351	0.969	1.508	0.079	3.907
6	STATIC,16	1.343	0.819	1.468	0.060	3.690
6	STATIC,32	1.479	0.821	1.608	0.069	3.977
6	STATIC,63	1.502	0.977	1.613	0.065	4.157
6	STATIC,64	1.447	0.960	1.612	0.054	4.073
6	STATIC,1024	8.433	3.236	8.409	0.577	20.655

8	STATIC,2	1.042	1.439	1.184	0.056	3.721
8	STATIC,4	0.996	1.251	1.124	0.050	3.421
8	STATIC,8	1.010	0.865	1.128	0.055	3.058
8	STATIC,16	0.971	0.669	1.078	0.031	2.749
8	STATIC,32	0.973	0.717	1.108	0.042	2.840
8	STATIC,63	1.065	0.858	1.177	0.057	3.157
8	STATIC,64	0.985	0.832	1.133	0.057	3.007
8	STATIC,1024	8.440	3.220	8.403	0.579	20.642

Timings for the optics kernel
Machine C

OpenMP parameters		**TIMES**				
Threads	**Scheduling**	**fft 1**	**transpose**	**fft 2**	**scaling**	**total**
1	SERIAL	2.576	1.817	2.585	0.334	7.312
2	STATIC,2	1.536	1.087	1.532	0.169	4.324
2	STATIC,4	1.508	1.290	1.518	0.160	4.476
2	STATIC,8	1.498	1.015	1.504	0.157	4.174

2	STATIC,16	1.504	0.962	1.509	0.152	4.127
2	STATIC,32	1.504	0.999	1.503	0.159	4.165
2	STATIC,63	1.531	1.009	1.529	0.167	4.236
2	STATIC,64	1.507	1.042	1.504	0.156	4.209
2	STATIC,1024	2.995	1.826	2.991	0.292	8.104
4	STATIC,2	0.769	0.810	0.779	0.096	2.454
4	STATIC,4	0.760	0.650	0.760	0.090	2.260
4	STATIC,8	0.793	0.621	0.751	0.089	2.254
4	STATIC,16	0.784	0.587	0.759	0.103	2.233
4	STATIC,32	0.758	0.532	0.789	0.090	2.169
4	STATIC,63	0.784	0.633	0.794	0.082	2.293
4	STATIC,64	0.751	0.602	0.743	0.085	2.181
4	STATIC,1024	3.048	1.936	3.031	0.353	8.368
6	STATIC,2	0.640	0.518	0.542	0.156	1.856
6	STATIC,4	0.607	0.481	0.505	0.114	1.707
6	STATIC,8	0.552	0.532	0.573	0.159	1.816
6	STATIC,16	0.619	0.424	0.631	0.104	1.778
6	STATIC,32	0.612	0.428	0.678	0.148	1.866
6	STATIC,63	0.641	0.549	0.623	0.100	1.913
6	STATIC,64	0.634	0.541	0.652	0.095	1.922

6	STATIC,1024	3.066	1.844	3.036	0.353	8.299
8	STATIC,2	0.528	0.506	0.409	0.181	1.624
8	STATIC,4	0.524	0.414	0.491	0.142	1.571
8	STATIC,8	0.539	0.380	0.465	0.114	1.498
8	STATIC,16	0.450	0.335	0.457	0.090	1.332
8	STATIC,32	0.460	0.350	0.422	0.082	1.314
8	STATIC,63	0.536	0.519	0.483	0.156	1.694
8	STATIC,64	0.547	0.478	0.525	0.118	1.668
8	STATIC,1024	3.069	1.936	3.080	0.375	8.460

OMP_SCHEDULE does effect timing

We first note that setting the environment variable OMP_SCHEDULE does effect timing for this program. That is, RUNTIME scheduling does work.

Changing scheduling has the biggest effect on the transpose, particularly for machines A and B. For these, the biggest change in the runtime is from the time spent in the transpose.

Not all changes in timing are from OpenMP overhead

Consider machine, A, where the time for the transpose with STATIC,2 scheduling is 2.319 seconds and 1.659 seconds for STATIC,4. The OpenMP microkernel benchmark was run on machine A. We find from that benchmark that the overhead associated with using STATIC,2 scheduling is 2.05 microseconds and STATIC,4 is 1.67 microseconds. The difference in overhead is insufficient to explain the differences in this program. It is possible that the poorer performance for the STATIC,2 case is from cache conflicts . The data being accessed by two threads might be on the same cache line. For larger chunk sizes we might be seeing better performance because the two processors are able to access the data on different cache lines.

OpenMP can decrease performance

For machines A and B the runtime of the transpose operation is actually longer using STATIC,2 scheduling than when the loop is run using a single thread. There are some loops, on some machines, with some scheduling algorithms that will run slower. Sometimes this is from cache conflicts as discussed above. For some of the first OpenMP compilers the slow down on some loops was dramatic. The slow down exceeded what was cause by cache conflict. It was from the compiler just generating bad code. Fortunately the instances of compliers generating bad code has decrease with newer compilers but it still does happen.

Running with a chunk size of 1024 exposes a compiler problem

The outer loops in this program have counts of 1024. So why run with a chunk size of 1024 when this forces all of the computation to run using a single thread?

This exposes additional overhead that is introduced by the OpenMP compiler. For machine B we have a serial runtime of 17.132 seconds. With a chunk size of 1024 the runtime is 20.701 seconds or about 3.5 seconds more. Most of the extra time is in the doing the loops that contains the routine four1. There should not be any significant synchronization associated with this loop. The other possibility is that turning on OpenMP effects the efficiency of the code generated for the routine four1. To test this, the program was recompiled and rerun. The routine four1 was split out into its own file and compiled with OpenMP turned off. The rest of the program was compiled with OpenMP turned on and then linked with four1. The difference in results is dramatic. The runtime for the case where chunk size was 1024 dropped from 20.701 seconds to 17.623 seconds, over 3 seconds. All of the reduction was seen in the loop that performed the fft. All of the cases showed speed up in four1 when the routine was compiled with OpenMP turned off. For one instance, 2 threads; STATIC,4, the transpose took longer. See the new results for machines B and C below.

Apparently, some optimizations for the routine four1 were not performed when OpenMP was turned on. There is no reason, from the standpoint of the language definition, that this should occur. As compilers mature this type of anomalous behavior should diminish.

Why 63?

One of the first machines that this program was run on was a Cray T90. The Cray T90 memory system did not do as well when multiple processors tried to access data that was offset in memory by certain strides. This was well documented and it was recommended that program writers avoid such simultaneous memory access. When this program was run with a chunk size of 64 the memory access recommendation was violated. When run with a chunk size of 63 the performance was much better. The Cray T90 is not in wide use today. However this example is included to point out again that OpenMP performance can be effected by the memory subsystem of a machine.

Advice

Check the runtime of your loops. Check that your loops actually show speed up. If they don't show speed up it could be because of memory conflicts. It pays to know the cache and memory subsystem of your machine. You may want to use RUNTIME scheduling. You can then try different scheduling algorithms without recompiling your program. For some machines you may even be able to change the scheduling on the fly by changing the environment variable while the program is running. (Some machines have library calls that allow system calls to do such things.) For important subroutines, try compiling them separately with OpenMP turned off. This may lead to additional optimizations being performed.

Timings for the optics kernel Machine B Routine four1 compiled separately.

OpenMP parameters		**TIMES**				
Threads	**Scheduling**	**fft 1**	**transpose**	**fft 2**	**scaling**	**total**
1	SERIAL	6.705	3.107	6.694	0.589	17.095
2	STATIC,2	3.378	5.341	3.529	0.251	12.499
2	STATIC,4	3.355	4.353	3.478	0.242	11.428

2	STATIC,8	3.365	2.513	3.477	0.245	9.600
2	STATIC,16	3.377	2.160	3.483	0.248	9.268
2	STATIC,32	3.330	1.999	3.417	0.221	8.967
2	STATIC,63	3.378	1.939	3.450	0.230	8.997
2	STATIC,64	3.332	1.936	3.409	0.220	8.897
2	STATIC,1024	6.986	3.113	6.938	0.586	17.623
4	STATIC,2	1.623	2.514	1.781	0.098	6.016
4	STATIC,4	1.597	2.454	1.783	0.103	5.937
4	STATIC,8	1.585	1.310	1.754	0.074	4.723
4	STATIC,16	1.582	1.155	1.745	0.089	4.571
4	STATIC,32	1.576	1.160	1.738	0.074	4.548
4	STATIC,63	1.672	1.192	1.791	0.093	4.748
4	STATIC,64	1.614	1.226	1.775	0.090	4.705
4	STATIC,1024	6.972	3.178	6.938	0.585	17.673
6	STATIC,2	1.143	1.982	1.235	0.079	4.439
6	STATIC,4	1.085	1.881	1.228	0.065	4.259
6	STATIC,8	1.125	0.835	1.254	0.047	3.261
6	STATIC,16	1.106	0.827	1.210	0.057	3.200
6	STATIC,32	1.178	0.852	1.316	0.063	3.409
6	STATIC,63	1.207	0.971	1.333	0.080	3.591

6	STATIC,64	1.197	0.947	1.341	0.069	3.554
6	STATIC,1024	6.960	3.272	6.925	0.579	17.736
8	STATIC,2	0.803	1.552	0.907	0.045	3.307
8	STATIC,4	0.801	1.389	0.908	0.046	3.144
8	STATIC,8	0.803	0.634	0.913	0.054	2.404
8	STATIC,16	0.786	0.653	0.912	0.041	2.392
8	STATIC,32	0.890	0.741	0.970	0.058	2.659
8	STATIC,63	0.876	0.825	0.937	0.043	2.681
8	STATIC,64	0.804	0.848	0.967	0.049	2.668
8	STATIC,1024	6.975	3.207	6.946	0.590	17.718

Timings for the optics kernel
Machine C
Routine four1 compiled separately.

OpenMP parameters		**TIMES**				
Threads	**Scheduling**	**fft 1**	**transpose**	**fft 2**	**scaling**	**total**
1	SERIAL	2.577	1.900	2.581	0.337	7.395

2	STATIC,2	1.329	1.372	1.338	0.162	4.201
2	STATIC,4	1.318	1.287	1.327	0.157	4.089
2	STATIC,8	1.298	1.098	1.308	0.156	3.860
2	STATIC,16	1.293	1.144	1.298	0.157	3.892
2	STATIC,32	1.306	0.973	1.307	0.159	3.745
2	STATIC,63	1.317	1.033	1.317	0.156	3.823
2	STATIC,64	1.309	1.052	1.306	0.160	3.827
2	STATIC,1024	2.602	1.856	2.606	0.297	7.361
4	STATIC,2	0.662	0.738	0.673	0.092	2.165
4	STATIC,4	0.659	0.714	0.680	0.095	2.148
4	STATIC,8	0.674	0.633	0.657	0.093	2.057
4	STATIC,16	0.658	0.607	0.653	0.090	2.008
4	STATIC,32	0.672	0.523	0.663	0.095	1.953
4	STATIC,63	0.740	0.621	0.673	0.094	2.128
4	STATIC,64	0.668	0.625	0.665	0.092	2.050
4	STATIC,1024	2.611	1.877	2.576	0.314	7.378
6	STATIC,2	0.516	0.634	0.479	0.162	1.791
6	STATIC,4	0.506	0.541	0.521	0.129	1.697
6	STATIC,8	0.552	0.461	0.486	0.147	1.646
6	STATIC,16	0.470	0.441	0.561	0.101	1.573

6	STATIC,32	0.571	0.491	0.541	0.178	1.781
6	STATIC,63	0.504	0.523	0.520	0.097	1.644
6	STATIC,64	0.505	0.562	0.546	0.070	1.683
6	STATIC,1024	2.598	1.825	2.603	0.381	7.407
8	STATIC,2	0.403	0.496	0.424	0.188	1.511
8	STATIC,4	0.425	0.454	0.450	0.230	1.559
8	STATIC,8	0.404	0.392	0.470	0.066	1.332
8	STATIC,16	0.505	0.363	0.427	0.083	1.378
8	STATIC,32	0.409	0.360	0.382	0.066	1.217
8	STATIC,63	0.392	0.488	0.394	0.134	1.408
8	STATIC,64	0.394	0.468	0.414	0.051	1.327
8	STATIC,1024	2.642	1.894	2.643	0.303	7.482

```
!this program is the kernel for an optical propagation program.
!it does a series of 2d ffts followed by a multiplication.
!it is modeled after the AFWL program HELP or High Energy
!Laser Propagation.
!
!it does the 2d fft by first doing a collection of 1d ffts
!then a transpose followed by a second collection of 1d ffts.
!
!the sections of the program that are commented out represent
!different ways of doing the operations.  the most interesting
!addition is using the using the subroutine shuff to generate
!a nonuniform ordering for accessing the array.

!the routines four1 was taken from the book
!"numerical recipes in fortran, 1st addition."
!however, the authors of that book derived their routine
!from the routine fourn that was in the AFWL program HELP.
!the original routine, fourn, contained many additional
!options


!disclaimer:  for a production code you would most likely
!use a vendor supplied library to do the fft instead of a
!hand written one.


module ccm_numz
! basic real types
    integer, parameter:: b8 = selected_real_kind(10)
contains
     function ccm_time()
        implicit none
        integer i
        integer :: ccm_start_time(8) = (/(-100,i=1,8)/)
        real(b8) :: ccm_time,tmp
        integer,parameter :: norm(13)=(/  &
               0, 2678400, 5097600, 7776000,10368000,13046400,&
        15638400,18316800,20995200,23587200,26265600,28857600,31536000/)
        integer,parameter :: leap(13)=(/  &
               0, 2678400, 5184000, 7862400,10454400,13132800,&
        15724800,18403200,21081600,23673600,26352000,28944000,31622400/)
        integer :: values(8),m,sec
        save
        call date_and_time(values=values)
        if(mod(values(1),4) .eq. 0)then
           m=leap(values(2))
        else
           m=norm(values(2))
        endif
        sec=((values(3)*24+values(5))*60+values(6))*60+values(7)
        tmp=real(m,b8)+real(sec,b8)+real(values(8),b8)/1000.0_b8
        !write(*,*)"vals ",values
        if(values(1) .ne. ccm_start_time(1))then
            if(mod(ccm_start_time(1),4) .eq. 0)then
                tmp=tmp+real(leap(13),b8)
            else
                tmp=tmp+real(norm(13),b8)
            endif
        endif
        ccm_time=tmp
    end function
end module ccm_numz

program two_d_fft
        use ccm_numz
        implicit none
!       integer size
        integer,parameter:: size=1024
!        integer omp_get_max_threads
        integer i,j,k,ijk,isign,iseed
        real(b8),allocatable:: x(:)
        integer, allocatable:: index(:)
        complex(b8), allocatable:: a(:,:)
!        complex(b8) :: a(size,size)
!        complex(b8), allocatable:: temp(:)
        complex(b8) tmp
        complex(b8) factor
        real(b8) gen,fft1,fft2,trans,totf,fact
        real(b8) t0,t1,t2,t3,t4,t5
        integer OMP_GET_MAX_THREADS
        interface
            subroutine shuff(index,m,n)
              dimension index(:)
              integer m,n
            end subroutine
        end interface
        factor=size
        factor=1.0_b8/(factor)
        iseed=-12345
        isign=1
        gen=0
        fft1=0
        fft2=0
        trans=0
        totf=0
        fact=0
!       read(12,*)size
        allocate(a(size,size))
        allocate(x(size),index(size))
        call shuff(index,size,8)
!       dummy=ran1(iseed)
!       write(*,*)"dummy=",dummy
        t0=ccm_time ()
        do j=1,size
           call random_number(x)
           do i=1,size
                a(i,j)=cmplx(x(i),0.0_b8)
           enddo
        enddo
        write(*,'(("(",g20.10,",",g20.10,")"))')a(size/2+1,size-2)

!        do 10 ijk=1,20  ! change to 4 to run faster
        do 10 ijk=1,20
            t1=ccm_time ()
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do i=1,size
                 call four1(a(:,i),size,isign)
                !call four1(a(i,:),size,isign)
            enddo
!$OMP END PARALLEL DO
            t2=ccm_time ()
!$OMP PARALLEL DO SCHEDULE (RUNTIME) PRIVATE(i,j,k,tmp)
            do k=1,size
                i=k
! i=index(k)
                do j=i,size
! tmp=a(j,i)
! a(j,i)=a(i,j)
! a(i,j)=tmp
                    tmp=a(i,j)
                    a(i,j)=a(j,i)
                    a(j,i)=tmp
                enddo
! j=i
! temp(j:size)=a(i,j:size)
! a(i,j:size)=a(j:size,i)
! a(j:size,i)=temp(j:size)
            enddo
```

```
!$OMP END PARALLEL DO
            t3=ccm_time ()
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do i=1,size
                 call four1(a(:,i),size,isign)
                !call four1(a(i,:),size,isign)
            enddo
!$OMP END PARALLEL DO
            t4=ccm_time ()
!$OMP PARALLEL DO SCHEDULE (RUNTIME)
            do j=1,size
            do i=1,size
                  a(i,j)=factor*a(i,j)
               enddo
            enddo
!$OMP END PARALLEL DO
            t5=ccm_time ()
            gen=gen+t1-t0
            fft1=fft1+t2-t1
            fft2=fft2+t4-t3
            trans=trans+t3-t2
            totf=totf+t5-t1
            fact=fact+t5-t4
            isign=isign*(-1)
            write(*,'(i3)',advance="no")ijk
    10  continue
        write(*,*)
        write(*,'(("(",g20.10,",",g20.10,")"))')a(size/2+1,size-2)
        write(*,*)"number of  transforms",ijk-1
        !write(*,'("generation time= ",f7.1)')gen
        write(*,'("      fft1 time= ",f9.4)')fft1
        write(*,'(" transpose time= ",f9.4)')trans
        write(*,'("      fft2 time= ",f9.4)')fft2
        write(*,'("   scaling time= ",f9.4)')fact
        write(*,'("    total time = ",f9.4)',advance="no")totf
        write(*,'(" for matrix of size",i6)')size
        write(*,*)"THREADS   = ",OMP_GET_MAX_THREADS()
        stop
end program two_d_fft

      subroutine four1(data,nn,isign)
      use ccm_numz
      implicit none
      integer i,j,isign,nn,n,m,mmax,istep
      real(b8), parameter :: two_pi = 6.283185307179586477_b8
      real(b8) wr,wi,wpr,wpi,wtemp,theta,tempr,tempi
      real(b8) data(16384)
      n=2*nn
      j=1
      do 11 i=1,n,2
        if(j.gt.i)then
          tempr=data(j)
          tempi=data(j+1)
          data(j)=data(i)
          data(j+1)=data(i+1)
          data(i)=tempr
          data(i+1)=tempi
        endif
        m=n/2
1       if ((m.ge.2).and.(j.gt.m)) then
          j=j-m
          m=m/2
        go to 1
        endif
        j=j+m
11    continue
      mmax=2
2     if (n.gt.mmax) then
        istep=2*mmax
        theta=two_pi/(isign*mmax)
        wpr=-2.0_b8*sin(0.5_b8*theta)**2
        wpi=sin(theta)
        wr=1.0_b8
        wi=0.0_b8
        do 13 m=1,mmax,2
          do 12 i=m,n,istep
            j=i+mmax
            tempr=(wr)*data(j)-(wi)*data(j+1)
            tempi=(wr)*data(j+1)+(wi)*data(j)
            data(j)=data(i)-tempr
            data(j+1)=data(i+1)-tempi
            data(i)=data(i)+tempr
            data(i+1)=data(i+1)+tempi
12        continue
          wtemp=wr
          wr=wr*wpr-wi*wpi+wr
          wi=wi*wpr+wtemp*wpi+wi
13      continue
        mmax=istep
      go to 2
      endif
      return
      end

subroutine shuff(index,m,n)
    integer tens,ones
    dimension index(:)
    tens=n
    ones=1
    j=0
    k=1
    do while (j < m)
       j=j+1
       if(k.gt. m)then
         write(*,*)k,m
         stop
       endif
       index(k)=j
       k=k+tens
       if(k .gt. m)then
         ones=ones+1
         k=ones
       endif
    enddo
end subroutine shuff
```