Parallelism

- Primitives for matrix manipulation hide all details of BLAS and LAPACK for linear algebra
- Primitives.

Sequential code designed with occasional calls to parallel

concert.

There are several approaches which can be used individually and in codes on the SGIs fairly typical.

The current environment for implementing shared memory parallel

Loop-based Parallel Programming on the SGI Origin 200
- Assumes the use of Fortran90
- No automatic dependence analysis assumed.
- Directive form used
- Such as OpenMP

(semi)Explicit parallel programming via emerging standards

Commercial/academic codes
- Parse 'KAP', VAST and several other
- APo on SGI/Cray machines
- Power Fortran and Power C on SGI's (now outdated)

Directives plus an automatic parallelizing compiler •
Used by 190, 77, cc, C89 compilers.

form, lock, wait, stratege, etc.

They include processor, chip, cache, miss penalties, prefetch

Apollo and Orion have the manuals of interest for OpenMP

machines describe these options (loop nest optimizing group)

man into and the initialize online documentation on the SCI

drive the code generation process

aspects of the architecture that the user wants used to

The SCI also allows several environment options to specify
The mp directive is required to generate parallel code, i.e., performance evaluation and controlling certain side effects.

O3 or Ofast typically used. The others can be used for compiler.

They are specified as a command line parameter to the -O0, -O1, -O2, -O3, -Ofast.

Several levels of code generation optimization are also possible.
Chapter 5 is also important especially for Origin 2000 users.

such as TNO.

Chapter 3 gives a complete discussion of the various directives through 4.7, and 4.11 and 4.12.

Read all of the sections with particular attention to 4.3 directives.

Chapter 4 contains a good introduction to the OpenMP Fortran 90 Commands and Directives manual.

OpenMP and Fortran 90 are taken from the Mipspro 7

There are several manuals of interest but the descriptions of
We must consider:

- Synchronization - When do we do it?
- Scheduling - Where do we do it?
- Control - What do we do?
- Memory Space - On what data do we operate?
is required.
careful coordination of parallel reads/writes of the shared data

Since every processor can, potentially, read every variable

- are referring to the same location in memory.

... processors access the same variable name, say, A, then they

- Unless otherwise specified (language dependent), if two

issue addresses

Each processor can access variables defined in the program

- Multiple processors cooperate to execute a program

Shared Memory
Interactions are mapped to particular processors.

• Scheduling policies are needed for loops to decide how

We will concentrate on parallel loops for control

• OPENMP functions like an explicit parallel language (almost)

OPENMP adds directives to indicate the user’s intention.

• OPENMP web page for a document that presents the C constructs

We assume Fortran 90 as our loop-based language. (See the

Control and Scheduling
A simple parallel do loop is specified as follows:

```sql
i$omp parallel do

    i

    enddo

enddo

q^2 = 3*a

do i=1,n

i$omp parallel do

    a
```
continuing sequentially.

A join is assumed after the loop with the master thread.

• Setting of other environment or directive variables.

The scheduling mechanism has not been specified above.

• Interactions.

The work in the parallel do is split across all of the threads.

The iteration variable is implicitly assumed to be private to

the threads.
i$OMP END PARALLEL DO

enddo

q(1) = 3*4(1)
do 1=n, n

i$OMP PARALLEL DO IF(n.e.1000), PRIVATE(1)

Several options can be specified explicitly.
must have a private copy for each thread.

• PRIVATE can be used to declare any number of variables that
  • are private.
  • The PRIVATE clause explicitly declares the iteration variable.
  • Parallelism is invoked only if the condition is true.
  • The IF clause gives a runtime condition that is evaluated.
examine the current environment.
- recall SETH and PRINTENV UNIX commands to set and
- operate system over the life of a code.
- overridden within a code or changed dynamically by the
  number of threads to be used by a code. This can be
  sets the environment variable OMP_NUM_THREADS
  variables controlling Fortran, C, and C++.
- man omp-threads gives details of routines and environment
  statically.
- The number of threads can be specified dynamically or
is used as a maximum value.
number of threads as it sees fit and any user-specified number
default. When the execution environment may also set the
• In general, it is recommended that this be set to false as a
dynamically.
true if the code is allowed to change the number of threads
The environment variable OMPI_DYNAMIC must be set to
•
CALL OMP-SET-NUM-THREADS() overrides the environment variable OMP_DYNAMIC.

- CALL OMP-SET-DYNAMIC(TRUE, or FALSE) overrides the same as the number of threads.

- CALL OMP-GET-NUM-PROCS() returns the number of physical processors currently available to the code. This is not returned when operating sequentially.

- CALL OMP-GET-NUM-THREADS() returns an integer master thread that spawned the parallelism and is also between 0 and OMP-GET-NUM-PROCS(). 0 is the master threads currently used in the team executing the parallel code.

- CALL OMP-GET-NUM-THREADS() returns the number of threads.

There are several Fortran routines that are relevant to the
environment variable

OMP_NUM_THREADS. It is

interpreted differently depending on the value of the
OMP_DYNAMIC.
Scheduling of a loop can be specified at compile-time or run-time.

- **SCHEDULE (DYNAMIC, chunk)** - chunk is an integer that specifies how many contiguous iterations are grouped together into the basic piece of work that is scheduled. The thread is used with one contiguous group of iterations sent to each thread.
- **SCHEDULE (STATIC, chunk)** - chunk is an integer that specifies how many contiguous iterations are grouped together into the basic piece of work that is scheduled.

In the PARALLEL loop directive the schedule can be characterized.

man.pe.environ discusses the environment variables associated with scheduling and threads.
is effectively a *docross* type scheduling of iterations.

Dynamically determines the next piece it will execute. This

After each thread finishes the last piece it was assigned it

together into the basic piece of work that is scheduled.
the end.
and then halt in load-balancing problems with smaller tasks at
granularity and reduce synchronization points at the beginning.
It uses a hybrid of static and dynamic ideas to increase
chunk size is reached.
for a piece is serviced. This continues until the minimum
(like static scheduling) then it is halved as each new request
starts at the number of iterations over the number of threads
execute. However, the size of the piece is not constant. It
assigned it dynamically determines the next piece it will
scheduled. After each thread mines the last piece it was
are grouped together into the basic piece of work that is
specified the minimum number of contiguous iterations that
SCHEDULED(GUIDED,chunk) – chunk is an integer that
A third mechanism is also allowed.
use of the environment variables for scheduling.

The clause `SCHEDULE(RUNTIME)` is used to indicate the

This is done via the environment variables.

This also possible to compile the code once and determine the

schedule at runtime.

The previous examples are all scheduled at compile time.
- setenv OMP_SCHEDULE "GUIDED,4"
- setenv OMP_SCHEDULE "DYNAMIC"
- setenv OMP_SCHEDULE "STATIC,2"
- setenv OMP_SCHEDULE "STATIC"

It is set with a type and chunk.

The run-time scheduling decision.

The environment variable OMP_SCHEDULE determines •
i$omp end parallel do
  enddo
  
  (p(t) = 3*t+t)
  do t=1,n  
    run-time:
    i$omp parallel do if(n.ge.1000), schedule(dynamic)
  enddo
  
  (p(t) = 3*t+t)
  do t=1,n  
    compile-time:
    i$omp parallel do if(n.ge.1000), schedule(dynamic)