Loop-based Parallel Programming on the SGI Origin 200

The current environment for implementing shared memory parallel codes on the SGI is fairly typical.

There are several approaches which can be used individually and in concert.

• Sequential code design with occasional calls to parallel primitives.
  – BLAS and LAPACK for linear algebra
  – Primitives for matrix manipulation hide all details of parallelism
• Directives plus an automatic parallelizing compiler
  – Power Fortran and Power C on SGI’s (now outmoded)
  – APO on SGI/Cray machines
  – Parafrase, KAP, VAST and several other commercial/academic codes

• (semi)Explicit parallel programming via emerging standards such as OpenMP
  – Directive form used
  – No automatic dependence analysis assumed.
  – Assumes the use of Fortran90
• The SGI also allows several environment option-
s to specify aspects of the the architecture that
the user wants used to drive the code generation
process

• **man Ino** and the **insight** online documentation
on the SGI machines describes these options (loop
nest optimizing group)

• They include processor chip, cache miss penalties,
prefetch form, lock wait strategies, etc.

• Used by f90, f77, cc, c89 compilers.

• Several levels of code generation optimization are
also possible: -O0, -O1, -O2, -O3, -Ofast.

• They are specified as a command line parameter
to the compiler.

• O3 or Ofast typically used. The others can be
useful for performance evaluation and controlling
certain side effects.

• The -mp directive is required to generate paral-
lel code, i.e., obey the directives and use parallel
libraries.
• There are several manuals of interest but the descriptions of OpenMP and Fortran 90 are taken from the *MIPSpro 7 Fortran 90 Commands and Directives* manual.

• Chapter 4 contains a good introduction to the OpenMP directives.

• Read all of the sections with particular attention to 4.3 through 4.7, and 4.11 and 4.12.

• Chapter 3 gives a complete discussion of the various directives such as LNO.

• Chapter 5 is also important especially for Origin 2000 users.
We must consider:

- Memory space – On what data do we operate?
- Control – What do we do?
- Scheduling – Where do we do it?
- Synchronization – When do we do it?
Shared Memory

- Multiple processors cooperate to execute a program

- Each processor can access variables defined in the program (issue addresses)

- Unless otherwise specified (language dependent), if two processors access the same variable name, say, $A$, then they are referring to the same location in memory.

- Since every processor can, potentially, read every variable careful coordination of parallel reads/writes of the shared data is required.
Control and Scheduling

- We assume Fortran 90 as our loop-based language. (See the web page for a document that presents the C constructs)

- OPENMP adds directives to indicate the user's intention.

- OPENMP functions like an explicit parallel language (almost)

- We will concentrate on parallel loops for control

- scheduling policies are needed for loops to decide how iterations are mapped to particular processors.
A simple parallel do loop is specified as follows

```!
$OMP PARALLEL DO

    do i=1,n
        b(i) = 3*a(i)
    enddo

$OMP END PARALLEL DO!
```
- The iteration variable is implicitly assumed to be private to the threads.

- The work in the parallel do is split across all of the worker threads (think of them as processors) that consume the iterations.

- The scheduling mechanism has not been specified above explicitly and therefore would have to be deduced from the setting of other environment or directive variables.

- A join is assumed after the loop with the master thread continuing sequentially.
Several options can be specified explicitly.

!$OMP PARALLEL DO IF(n.ge.1000), PRIVATE(i)

    do i=1,n
        b(i) = 3*a(i)
    enddo

!$OMP END PARALLEL DO
• The IF clause gives a runtime condition that is evaluated. Parallelism is invoked only if the condition is true.

• The PRIVATE clause explicitly declares the iteration variable private.

• PRIVATE can be used to declare any number of variables that must have a private copy for each thread.
• The number of threads can be specified dynamically or statically.

• **man omp_threads** gives details of routines and environment variables controlling Fortran, C, and C++.

• The environment variable **OMP_NUM_THREADS** sets the number of threads to be used by a code. This can be superseded within a code or changed dynamically by the operating system over the life of a code.

• recall setenv and printenv UNIX commands to set and examine the current environment.
• The environment variable OMP_DYNAMIC must be set to true if the code is allowed to change the number of threads dynamically.

• In general it is recommended that this be set to false as a default. When true the runtime environment may also set the number of threads as it sees fit and any user specified number is used as a maximum value.
• There are several Fortran routines that are relevant to the number of threads.

• CALL OMP_GET_NUM_THREADS() returns the number of threads currently used in the team executing the parallel code. It returns 1 if called during sequential execution.

• CALL OMP_GET_THREAD_NUM() returns an integer between 0 and OMP_GET_NUM_THREADS()-1. 0 is the master thread that spawned the parallelism and is also returned when operating sequentially.

• CALL OMP_GET_NUM_PROCS() returns the number of physical processors currently available to the code. This is not the same as the number of threads.

• CALL OMP_SET_DYNAMIC(.TRUE. or .FALSE.) overrides the environment variable OMP_DYNAMIC.

• CALL OMP_SET_NUM_THREADS() overrides the environment variable OMP_NUM_THREADS. It is interpreted differently depending on the value of the environment variable OMP_DYNAMIC.
• Scheduling of a loop can be specified at compile-time or run-time.

• **man pe_environ** discusses the environment variables associated with scheduling and threads.

• In the PARALLEL loop directive the schedule can be characterized.
  
  – **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. The pieces are assigned to the threads in an interleaved fashion. If no chunk is specified then standard STATIC scheduling is used with one contiguous group of iterations sent to each thread.

  – **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. After each thread finishes the last piece it was assigned it dynamically determines the next piece it will execute. This is effectively a **doacross** type scheduling of iterations.
• A third mechanism is also allowed.

• \texttt{SCHEDULE(GUIDED,chunk)} – chunk is an integer that specifies the minimum number of contiguous iterations that are grouped together into the basic piece of work that is scheduled. After each thread finishes the last piece it was assigned it dynamically determines the next piece it will execute. However the size of the piece is not constant. It starts at the number of iterations over the number of threads (like static scheduling) then it is halved as each new request for a piece is serviced. This continues until the minimum chunk size is reached.

• It uses a hybrid of static and dynamic ideas to increase granularity and reduce synchronization points at the beginning and then fill in load balancing problems with smaller tasks at the end.
• The previous examples are all scheduled at compile time.

• It is also possible to compile the code once and determine the schedule at run-time.

• This is done via environment variables.

• The clause SCHEDULE(RUNTIME) is used to indicate the use of the environment variables for scheduling.
• The environment variable **OMP_SCHEDULE** determines the run-time scheduling decision.

• It is set with a type and chunk.
  – `setenv OMP_SCHEDULE "STATIC"`
  – `setenv OMP_SCHEDULE "STATIC,2"`
  – `setenv OMP_SCHEDULE "DYNAMIC"`
  – `setenv OMP_SCHEDULE "GUIDED,4"`
Compile-time:

!$OMP PARALLEL DO IF(n.ge.1000), SCHEDULE(DYNAMIC,4)
   do i=1,n
      b(i) = 3*a(i)
   enddo
!$OMP END PARALLEL DO

Run-time:

!$OMP PARALLEL DO IF(n.ge.1000), SCHEDULE(RUNTIME)
   do i=1,n
      b(i) = 3*a(i)
   enddo
!$OMP END PARALLEL DO