Lecture 23 – OpenMP Constructs

- **OpenMP's constructs fall in 5 categories:**
  - Parallel regions
  - Synchronization
  - Worksharing
  - Data environment
  - Runtime functions/environment variables

We discussed these in the last lecture.

Runtime Functions

- **Run time library can be used to control and query the parallel execution environment**
  - OMP_SET_NUM_THREADS()
  - OMP_GET_NUM_THREADS()
  - OMP_GET_THREAD_NUM()
  - OMP_GET_NUM_PROCS()
  - OMP_IN_PARALLEL(): *return true or false*

- **When calling these OMP functions from a C-code, you need to**

  #include <omp.h>

  - On Wopr, omp.h is located at /share/apps/pgi/linux86/7.2-5/include. You can point to this by adding the compiler option

    -I/share/apps/pgi/linux867.2-5/include

Programming Errors

- **Shared memory parallel programming is a mixed bag**
  - It saves programmer from mapping data onto multiple processors
  - However, it may introduce new errors coming from unanticipated shared resource conflicts or contentions

Common SMP Errors

- **Race conditions:**
  - The outcome of the program is dependent on the detailed timing of the threads
- **Deadlock**
  - Threads lock up waiting on a locked resource that will never become available
- **Livelock**
  - Multiple threads working individual tasks which the ensemble can't finish

Race Conditions

- **The result varies unpredictably based on detailed order of execution for each section**
- **Wrong answers produced without warning**

```
!$OMP PARALLEL SECTIONS
      A=B+C
!$OMP SECTION
      B=A+C
!$OMP SECTION
      C=B+A
!$OMP END PARALLEL SECTIONS
```


OpenMP Death-Traps

- **Are you using thread-safe libraries?**
- **I/O inside a parallel region can interleave unpredictably**
- **Private variables can mask global**
- **Understand when shared memory is coherent. When in doubt use FLUSH**
  - FLUSH provides a means for making memory consistent across threads (only for shared variables).
    !$OMP FLUSH (list)
- **NOWAIT removes implied barriers**
  - Threads can proceed to the next statement without waiting for all other threads to complete the “for” (C) or “do” (Fortran) loop execution

Portable Sequential Equivalence

- **What is Portable Sequential Equivalence (PSE)?**
  - A program is sequentially equivalent if its results are the same with one thread or many threads
  - For a program to be portable (runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored
- **Strong SE: bitwise identical results**
- **Weak SE:equivalent mathematically, not bitwise identical**
  - This is the case for MPI codes

Portable Sequential Equivalence

- **Strong SE:**
  - Locate all cases where a shared variable can be written by multiple threads
    - The access to the variable must be protected
    - If multiple threads combine results into a single value, enforce sequential order
- **Weak SE:**
  - Floating point arithmetic is not associative and not commutative
  - In most cases no particular grouping is mathematically preferred so why choose the sequential order?

Example

- **The summation of TMP into RES occurs one thread at a time, but in any order so the result is not bitwise equivalent to the sequential one.**

```
!$OMP PARALLEL PRIVATE(I,TMP)
!$OMP DO
        DO I=1,NDIM
          TMP=FOO(I)
!$OMP CRITICAL
          CALL COMBINE(TMP,RES)
!$OMP END CRITICAL
        END DO
!$OMP END PARALLEL
```

Remember that CRITICAL directive allows only one thread (first to arrive) to execute that section.

To be bitwise equivalent, the same thread always must do the sum in the same order.

OpenMP Scalability

- **Memory is often the limit to achievable performance of a shared memory program**
- **On scalable architectures, the latency and bandwidth of memory accesses depend on the locality of those accesses**
- **In achieving good speedup of a shared memory program, data locality is an *essential element***

Data Distribution on DSM Computers

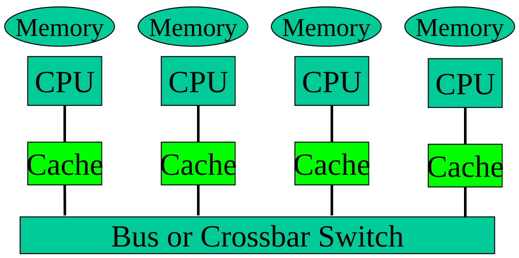
- **Initial data distribution determines on which node of a Distributed Shared Memory machine the memory is placed. This is dependent on**
  - "First touch" or "round-robin" system policies
  - Data distribution directives
  - Explicit page placement
- **Work sharing, e.g., loop scheduling, determines which thread accesses which data**
- **Cache friendliness determines how often main memory is accessed**

False Sharing

- ***Contention* is an issue specific to parallel loops, e.g., *false sharing of cache lines***
- ***Cache Friendliness* = high locality of references**
  +
  low contention

SMP- NUMA

- **Memory hierarchies exist in single-CPU computers and Symmetric Multiprocessors (SMPs)**
- **Distributed shared memory (DSM) machines based on Non-Uniform Memory Architecture (NUMA) (like the older SGI Origin) add levels to the hierarchy:**
  - *Local memory* has low latency
  - *Remote memory* has high latency



SGI Origin 2000 Memory

Level	Latency (cycles)
register	0
primary cache	2...3
secondary cache	8...10
local main memory & TLB hit	75
remote main memory & TLB hit	250
main memory & TLB miss	2000
page fault	10^6

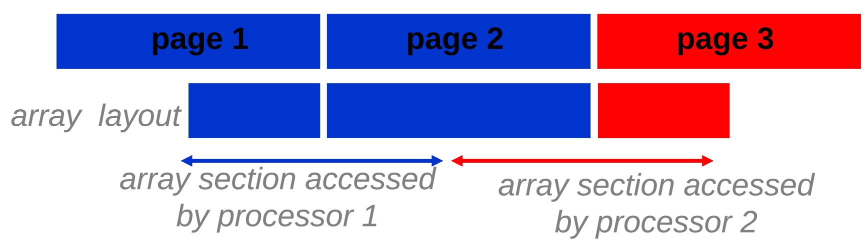
TLB: Translation Lookaside Buffer

Page Locality

- **An ideal application has full *page locality*:**
  - Pages accessed by a processor are on the same node as the processor, and
  - No page is accessed by more than one processor (no page sharing)
- **Twofold benefit:**
  - Low memory latency
  - Scalability of memory bandwidth

Page Locality Example

- **Consider an array whose size is twice the size of a page, and which is distributed between two nodes**
- **Page 1 and page 2 are located on node 1, page 3 is on node 2**



- **Page 2 is shared by the two processors, due to the array not starting on a page boundary**

Page Locality

- **The benefits of page locality are more important for programs that are *not cache friendly***
- **Several data placement strategies for improving page locality include:**
  - System based placement (such as on IRIX)
    - *first-touch*: the process which first references a virtual address causes that address to be mapped to a page on the node where the process runs
    - *round-robin:* pages allocated to a job are selected from nodes traversed in round-robin order
  - Data initialization and directives (system specific, not in the OpenMP standard)

Using an SPMD Approach Instead of Loop Breakup

- **An SPMD approach can be used in both Distributed- and Shared-Memory Systems**
- **Data is distributed explicitly among processes**
- **With message passing, e.g., MPI, where no data is shared, data is explicitly communicated**
  - Synchronization is explicit or embedded in communication
- **With parallel regions in OpenMP, both SPMD and data sharing are supported**

Using an SPMD Approach Instead of Loop Breakup

- **One can achieve a potentially higher parallel fraction with an SPMD approach rather than with loop parallelism in Shared-Memory Systems**
  - The fewer parallel regions, the less overhead
  - However, more explicit synchronization needed than for loop parallelization
    - Essentially using OpenMP to mimic the same type of parallelization strategy that would be used with MPI
  - Does not promote incremental parallelization and requires manually assigning data subsets to threads

SPMD Example: Matrix Initialization

- **A single parallel region, no scheduling needed, each thread explicitly determines its work**
  - **threads are independent**

```
  program mat_init
  implicit none
  integer, parameter::N=1024
  real A(N,N)!A and N are shared
  integer :: iam, np
  iam = 0
  np = 1
!$omp parallel private(iam,np)
  np = omp_get_num_threads()
  iam = omp_get_thread_num()
! Each thread calls work
  call work(N, A, iam, np)
!$omp end parallel
  end
```

```
 subroutine work(n, A, iam, np)
 integer n, iam, n
 real A(n,n)
 integer :: chunk,low,high,i,j
 chunk = (n + np - 1)/np
 low = 1 + iam*chunk
 high=min(n,(iam+1)*chunk)
! Note that both loops are
!   parallelized with SPMD approach
 do j = low, high
   do I=1,n
    A(I,j)=sqrt(real(i*i+j*j))
   enddo
 enddo
 return
```

Summary of OpenMP Constructs

- **Parallel region**

```
!$omp parallel        #pragma omp parallel
```

- **Worksharing**

```
!$omp do              #pragma omp for
!$omp sections        #pragma omp sections
!$single              #pragma omp single
!$workshare           #pragma workshare
```

- **Data environment**
  – directive: threadprivate
  – clauses: shared, private, lastprivate, reduction, copyin, copyprivate
- **Synchronization**
  – directives: critical, barrier, atomic, flush, ordered, master
- **Runtime functions/environment variables**

Achieving Good Parallel Performance

- **Optimize single-CPU performance**
  – Maximize cache reuse
  – Eliminate cache misses
  – Use compiler flags to optimize when possible
- **Parallelize as high a fraction of the work as possible with OpenMP**
  – Preserve cache friendliness
  – Avoid synchronization and scheduling overhead:
    - partition in few parallel regions,
    - avoid reduction, single and critical sections,
    - use static scheduling
    - partition work to achieve load balancing
- **Check correctness of parallel code**
  – Run OpenMP compiled code first on one thread, then on several threads

OpenMP or MPI?

- **Do you need total portability?**
  MPI
- **Do you need to use hundreds of processors?**
  MPI
- **Do you have access to DSM memory machines?**
  OpenMP
- **Do you want to be ready for the next generation of computers?**
  MPI and OpenMP or
  MPI and OpenACC (for GPUs)

OpenMP on Wopr/Vortex

- **There are a couple of ways to perform shared-memory parallel computing of loops on Wopr and Vortex using the pgf90 compiler**
  – You can read about the various options by reading the manpages of pgf90:
    man pgf90 (or by reading the users' manual)
  – Compiler options include:
    - The –concur=option[altcode, noaltcode, dist:block, dist:cyclic, cncall, noassoc] (See the manpage for description of options)
    - The –Mconcur option must be used in linking step.
  – OpenMP can be invoked:
    - Add the –mp option during compilation and linking (see chapter 10 of PGI users' manual)
    - The –noopenmp option will turn off any OpenMP directives so that you can run serial loops
  – You can try these out on your single-block simulation code (project-1)

Manuals and Other Information

- **I have put the following manuals out on smartsite under “Additional Material/OpenMP”:**
  - pgf90 manuals
  - OpenMP manuals
  - An example program, qsub script, and qsub command “runme” for Wopr or Vortex

OpenMP on Wopr

- **Compiling:**

  /share/apps/pgi/linux86/7.2-5/bin/pgf90 –c –mp file.f

- **Linking:**

  /share/apps/pgi/linux86/7.2-5/bin/pgf90 –o exec -mp file.o –lpthread

Homework 8

- **Read Chaps 9 and 11 in <u>Introduction to Parallel Computing</u> by Grama et al.**