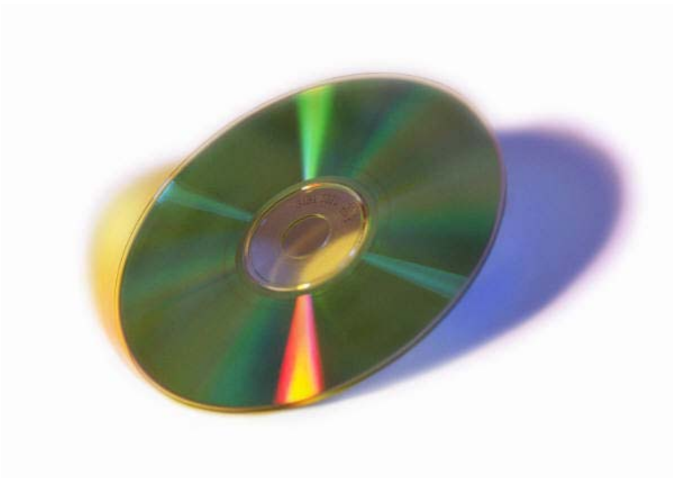

Parallel software

Parallel software - MIMD

- Hardware and compilers can keep up the pace needed.
- From now on…
  - In shared memory programs:
    - Start a single process and fork threads.
    - Threads carry out tasks.
  - In distributed memory programs:
    - Start multiple processes.
    - Processes carry out tasks.

SPMD – single program multiple data

- Instead of running a different program on each core
- A SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

SPMD

- SPMD programs can readily implement data-parallelism.
- For example:

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.
2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

```
double x[n], y[n];
. . .
for (int i = 0; i < n; i++)
    x[i] += y[i];
```

Shared Memory

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.
- Static threads
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.

Nondeterminism

```
. . .
printf ( "Thread %d > my_val = %d\n" ,
           my_rank , my_x ) ;
. . .
```

Thread 1 > my_val = 19
Thread 0 > my_val = 7

Thread 0 > my_val = 7
Thread 1 > my_val = 19

Nondeterminism

- In many cases nondeterminism isn't a problem
- However, there are also many cases in which nondeterminism can be disastrous → program errors
- E.g. Two threads want to execute the code (x is shared)

```
my_val = Compute_val(my_rank);
x += my_val;
```

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

Nondeterminism

- Note that the code does ***not*** impose any predetermined order on the threads
- Either thread 0 or thread 1 can execute x += my_val first
- The use of a mutex enforces **serialization** of the critical section → minimize critical sections

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

Busy-waiting

- There are alternatives to mutexes
- In **busy-waiting**, a thread enters a loop whose purpose is to test a condition
- Easy to understand and implement **BUT** very wasteful of system resources
- E.g. suppose there is a shared variable *ok_for_1* that has been initialized to false

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);   /* Busy-wait loop */
x += my_val;            /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;     /* Let thread 1 update x */
```

More alternatives…

- Semaphores
- Monitors
- Transactional memory

Input and Output rules/assumptions

- In distributed memory programs, only process 0 will access *stdin*.
- In shared memory programs, only the master thread or thread 0 will access *stdin*.

- In both distributed memory and shared memory programs all the processes/threads can access *stdout* and *stderr*.

Input and Output rules/assumptions

- However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

- Debug output should always include the rank or id of the process/thread that's generating the output.