Introduction to parallel computing with OpenMP

Sergiy Bubin

Department of Physics
Nazarbayev University

Why parallel computing?

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously

- High-performance computing requires huge number-crunching capabilities
- Physical limitations on frequency scaling and the associated heat removal from CPU

$$P = C \times V^2 \times F$$

where P is the power consumption, C is the capacitance being switched per clock cycle, V is voltage, and F is the processor frequency (cycles per second). Normally, an increase in voltage is also necessery to get to a higher processor frequency. So the actual power goes more like $P \propto F^2$

Serial computing

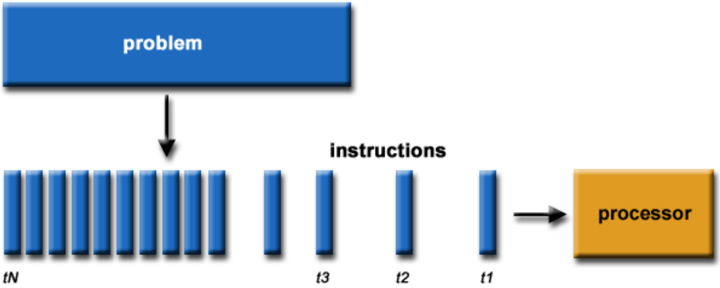


image credit: LLNL

Parallel computing

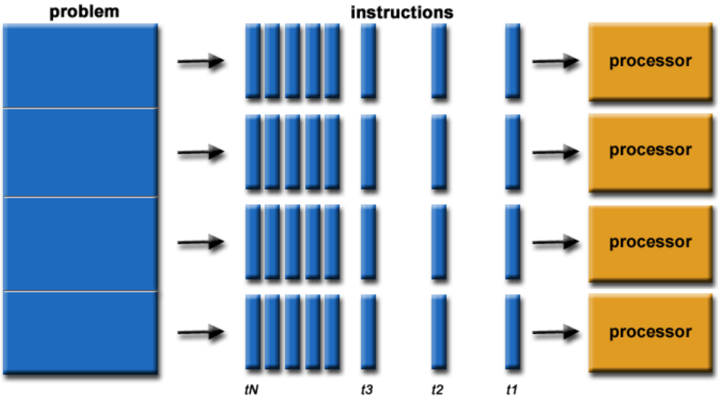
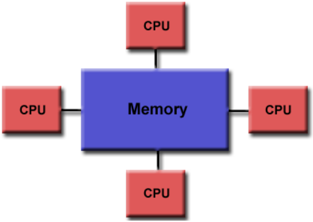
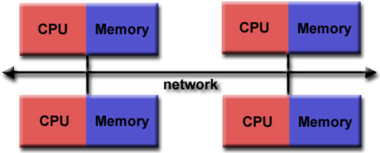


image credit: LLNL

Parallel Computer Memory Architectures

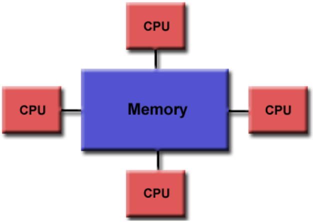


Shared Memory



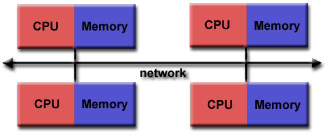
Distributed Memory

Shared Memory Parallel Computers



- All processors have the ability to access all memory as global address space
- Multiple processors can operate independently but share the same memory resources
- Changes in a memory location made by one processor are visible to all others
- Global address space provides user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Distributed Memory Parallel Computers



- Distributed memory systems require a communication network to connect inter-processor memory
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated
- Memory is scalable with the number of processors. However, the communication speed between processors (usually via network) is slow.

What is OpenMP?

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports **shared memory** multiprocessing programming

- Available on most platform and operating systems (Linux, OS X, Windows, Solaris, AIX, HP-UX, etc)
- API components: Compiler Directives, Runtime Library Routines, Environment Variables

Basic Features

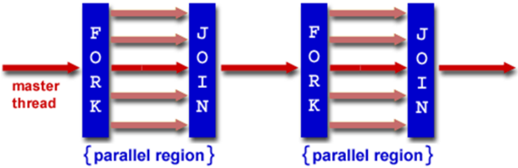
- OpenMP API is specified for C/C++ and Fortran
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++
- A programmer can parallelize his/her code incrementally, one function or even a loop at a time. This is convenient, but fundamentally it could be a good thing or a bad thing.
- OpenMP is the de facto standard for writing shared memory programs

Compilation and running programs that use OpenMP

Provided that a program is properly written (more on that later) these are the steps

- To compile:
  ```
  gfortran -fopenmp myprog.f90 -o myprog
  ```
  [Fortran]
  ```
  gcc -fopenmp myprog.c -o myprog
  ```
  [C]
- To run:
  First set the value of the environmental variable OMP_NUM_THREADS, which defines the number of threads (otherwise a default number will be used, which is usually equal to the number of logical cores in the computer)
  ```
  export OMP_NUM_THREADS=4
  ```
  [bash]
  Then run the program as usual
  ```
  ./myprog
  ```

OpenMP Execution Model



- An OpenMP program begins with a single master thread
- The master thread executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK).
- When the team threads complete the parallel region, they synchronize and terminate (JOIN). After that only a single master thread continues.
- There i no limit on how many times a program can FORK and JOIN

OpenMP General Code Structure

```
program main
use omp_lib
   [... declarations ...]
   [... do some serial tasks]
   !Beginning of the parallel section of the code
   !$omp parallel shared (x,y) private (i) ←—OpenMP clauses
   !$omp do ←— OpenMP clauses
   do i = 1,1000
      x(i)=x(i)*2
      y(i)=sin(3.0d0*i)
   end do
   !$omp end do ←— OpenMP clauses
   !End of the parallel section of the code
   [... do some other serial tasks]
end program main
```

Private and Shared data

- Variables in the global data space are accessed by all parallel threads (shared variables)
- Variables in a threads private space can only be accessed by the thread (private variables)
- One could a clause such as !$omp parallel default(shared) private(i)

Reductions

Example of computing a sum:

```
!$omp parallel do reduction (+:s)
do i = 1,n
   s = s + a(i)*b(i)
enddo
write(*,*) 's=',s
```

Calling functions and subroutines

If you need to call functions (or subroutines) in parallel parts of your code you must ensure those are thread safe.
One simple way of ensuring that is to declare all necessary functions with a prefix `recursive`, e.g.

```
recursive integer function myfunc(...
recursive subroutine mysub(...
```

This way all function/subroutine local variables declared within the function/subroutine will be on stack (thread-local) and consistent with threading.

References

A huge number of tutorials and reference books exist on the internet. Feel free to google. Check out these, for example:

- https://computing.llnl.gov/tutorials/openMP/
- http://www.openmp.org/resources/openmp-books/
- http://sc.tamu.edu/shortcourses/SC-openmp/OpenMPSlides_tamu_sc.pdf