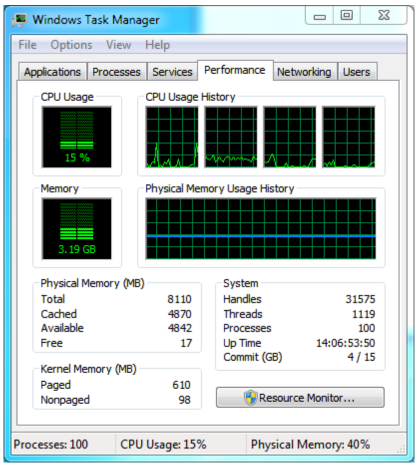
Parallel Computing with OpenMP

Yutaka Masuda

Computing cores

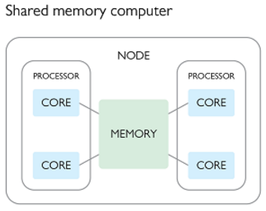


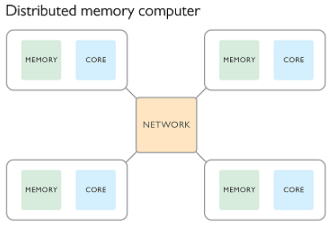
```
Physical Memory (MB)          System
Total          8110           Handles            31575
Cached         4870           Threads             1119
Available      4842           Processes            100
Free             17           Up Time      14:06:53:50
                              Commit (GB)       4 / 15
Kernel Memory (MB)
Paged           610
Nonpaged         98

Processes: 100     CPU Usage: 15%     Physical Memory: 40%
```

- A modern CPU usually has 2 or more *computing cores*.
- A regular program (your Fortran program) uses only 1 core.
- Why don't you use multiple cores for your computations?

Two major approaches

- OpenMP
  - A set of directives
  - Focus on parallelization for loops = limited purpose
  - Automatic management by the program = easier to program
  - Shared memory
- MPI (Message Passing Interface)
  - A collection of subroutines
  - Any kinds of parallel computing = flexible
  - Manual control of data flow & management = complicated
  - Distributed / shared memory

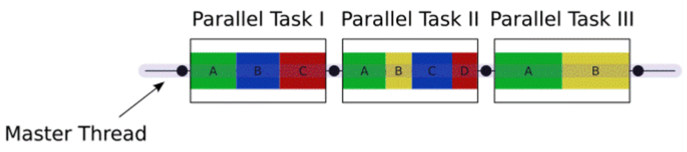




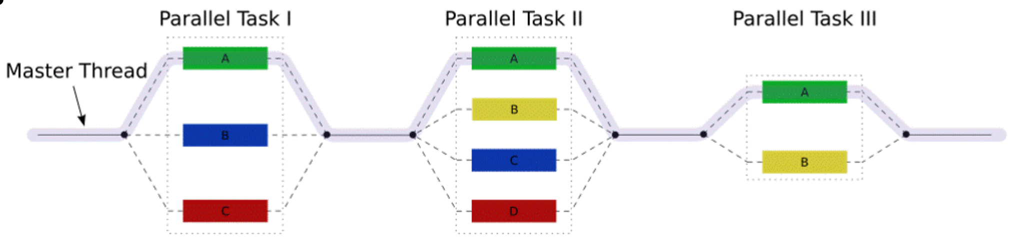
From www.comsol.com

Computing model in OpenMP

Regular (sequential) program:



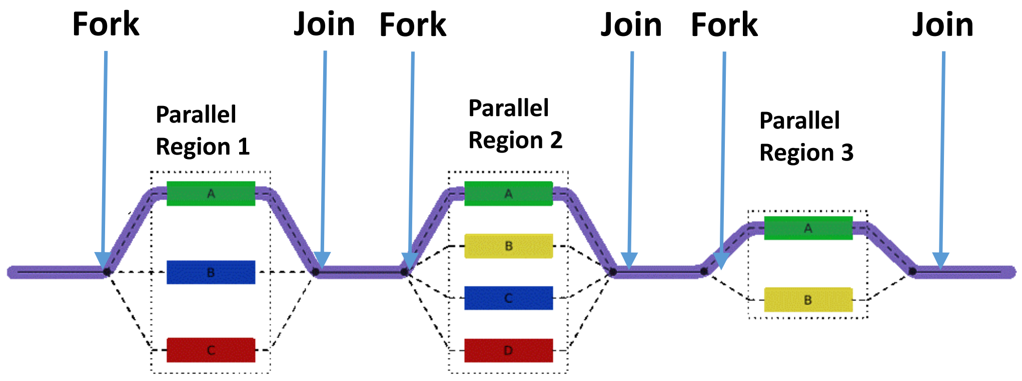
Parallel program:



From Wikipedia

Fork-Join model

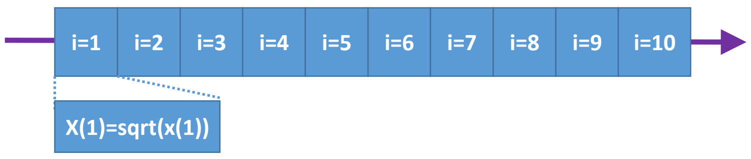
Fork: creation & initialization of threads
Join: synchronization of the threads



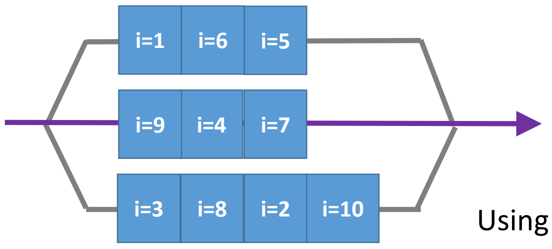
From Wikipedia

Program structure with OpenMP

```
do i=1,10
  x(i)=sqrt(x(i))
end do
```



```
!$omp parallel
!$omp do
do i=1,10
  x(i)=sqrt(x(i))
end do
!$omp end do
!$omp end parallel
```



Using 3 threads.

OpenMP directives

```
!$omp parallel
!$omp do
do i=1,10
  x(i)=sqrt(x(i))
end do
!$omp end do
!$omp end parallel
```

- The directive must begin with a keyword !$omp.
  - The directives will be effective obly if you put a compiler option.
  - Otherwise, the directives will be ignored (because it looks like a comment).
- An OpenMP region must be encircled with !$omp *directive* and !$omp end *directive*.

OpenMP directives (cont'd)

```
!$omp parallel private(i) shared(x)
!$omp do
do i=1,10
  x(i)=sqrt(x(i))
end do
!$omp end do
!$omp end parallel

!$ print *,'OpenMP is active!'
```

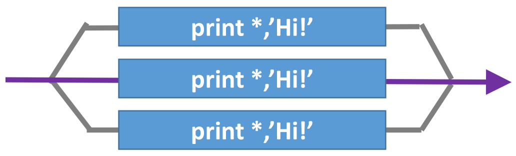
- Each directive can have an optional clause.
  - Variable type, number of threads, conditional execution etc.
- A statement starts with !$ will be complied only when the OpenMP is effective (conditional compilation).
  - Put a space between !$ and the statement.

Compiler options

- Depends on compilers
  - Intel Fortran Compiler (ifort): `-openmp` or `-qopenmp` (v16 or later)
  - Gfortran: `-fopenmp`
  - Absoft: `-openmp`
  - NAG: `-openmp`
  - PGI: `-mp`
- Examples
  - `ifort -openmp prog.f90`
  - `gfortran -fopenmp prog.f90`

Directive: **parallel**

```
!$omp parallel
print *,'Hi!'
!$omp end parallel
```

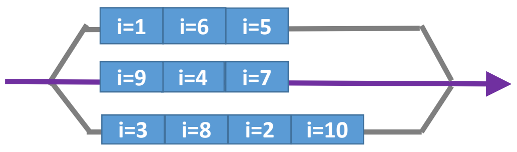


```
(Output)
 Hi!
 Hi!
 Hi!
```

- Defines a parallel region and assigns the task to each thread.
  - The region will be executed by multiple threads.
  - The number of threads can be controlled by the an optional clause, supplemental functions or an environmental variable.

Directive **do**

```
!$omp parallel
!$omp do
do i=1,10
  x(i)=sqrt(x(i))
end do
!$omp end do
!$omp end parallel
```

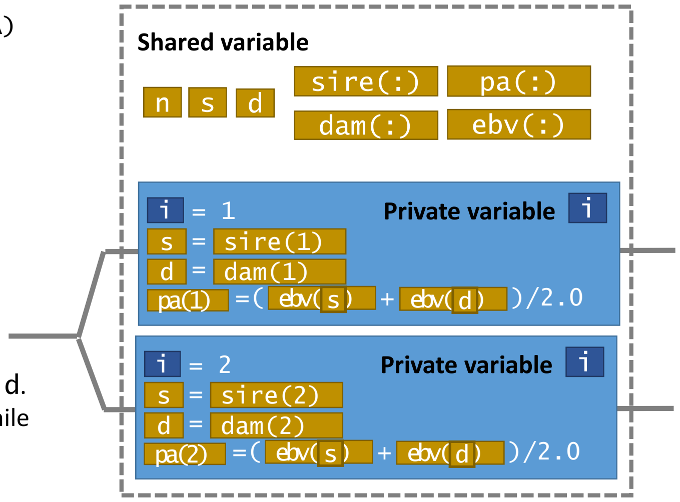


- Perform the do-loop with multiple threads.
  - The !$omp do directive must be placed just before a do-loop.
  - The directive must be surrounded by parallel.
  - The counter is not necessarily incremented in order.
  - The counter *i* is treated as a separate variable for each thread (private variable).

Shared variable by default

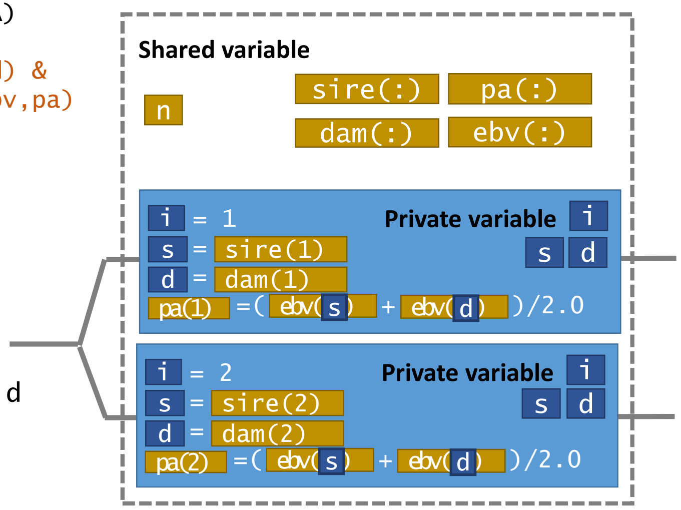
```
! compute parent average (PA)

!$omp parallel
!$omp do
do i=1,n
  s=sire(i)
  d=dam(i)
  pa(i)=(ebv(s)+ebv(d))/2.0
end do
!$omp end do
!$omp end parallel
```

- All threads share the variables s and d.
- One thread rewrites the variables while another thread cites the variable!



Private and shared variable

```
! compute parent average (PA)

!$omp parallel private(i,s,d) &
!$omp   shared(n,sire,dam,ebv,pa)
!$omp do
do i=1,n
  s=sire(i)
  d=dam(i)
  pa(i)=(ebv(s)+ebv(d))/2.0
end do
!$omp end do
!$omp end parallel
```



- Each thread has own variables s and d so there is no competition any more.

Clause: `shared` and `private`

```
! compute parent average (PA)

!$omp parallel private(i,s,d) &
!$omp   shared(n,sire,dam,ebv,pa)
!$omp do
do i=1,n
  s=sire(i)
  d=dam(i)
  pa(i)=(ebv(s)+ebv(d))/2.0
end do
!$omp end do
!$omp end parallel
```

- Define variable types.
  - Use `private()` and `shared()` clauses in the parallel directive.
  - *Private variables* will be created for each thread.
  - *Shared variables* will be shared (rewritten) by all threads.
  - Variables will be shared by default except loop counters.
  - Always declare the variable type to avoid bugs.

Clause: `reduction`

```
known=0

!$omp parallel private(i,s,d) &
!$omp   shared(n,sire,dam,ebv,pa) &
!$omp   reduction(+:known)
!$omp do
do i=1,n
  s=sire(i)
  d=dam(i)
  pa(i)=(ebv(s)+ebv(d))/2.0
  if(s/=0.and.d/=0) known=known+1
end do
!$omp end do
!$omp end parallel
```

- Specify variable for “reduction” operations.
  - A variable *known* is treated as private for each thread.
  - In the end of the loop, all threads will add their private *known* to the global *known*.
  - Other operations (instead of +) are available:
    - `+`,`*`,`max`,`min` etc.

Clause: `if`

```
known=0

!$omp parallel private(i,s,d) &
!$omp   shared(n,sire,dam,ebv,pa) &
!$omp   reduction(+:known) &
!$omp   if(n>100000)
!$omp do
do i=1,n
  s=sire(i)
  d=dam(i)
  pa(i)=(ebv(s)+ebv(d))/2.0
  if(s/=0.and.d/=0) known=known+1
end do
!$omp end do
!$omp end parallel
```

- Conditional use of OpenMP
  - If the condition is true, OpenMP will be invoked in the parallel region.
  - If not, the OpenMP directives in this region will be ignored (i.e. single-thread execution).

Built-in functions/subroutines

```
use omp_lib
```

or

```
!$ use omp_lib
```

- Built-in functions/subroutines for OpenMP are defined in the module omp_lib.
  - Recommendation: always cite this module as !$ use omp_lib because the module is usable only when you put a compiler option.
- See the textbook or openmp.org for details.

Built-in function: `omp_get_wtime`

- OpenMP function omp_get_wtime() returns wall-clock time.

```
!$ use omp_lib
integer,parameter :: r8=selected_real_kind(15,300)
real(r8) :: tic,toc
...
!$ tic=omp_get_wtime()
!$omp parallel
!$omp do
do
...
end do
!$omp end do
!$omp end parallel
!$ toc=omp_get_wtime()
!$ print *,'running time=',toc-tic
```

Number of threads

- The default number of threads is the maximum number on your system.
- A parallel program will be slow if …
  - You separately run another parallel program and each program tries to use the maximum number of threads.
- Three different ways to change the number of threads.
  1. Region-specific configuration (use of a clause in the parallel directive)
  2. Program-specific configuration (use of a built-in subroutine)
  3. Run-time configuration (use of an environmental variable)

Approach 1

```
integer :: n
n = 2
!$omp parallel num_threads(n)
!$omp do
do
...
end do
!$omp end do
!$omp end parallel
```

- Use of num_threads clause.
  - This is a region-specific configuration.

Approach 2

```
!$ use omp_lib
integer :: n
N=2

!$call omp_set_num_threads(n)

!$omp parallel
!$omp do
do
...
end do
!$omp end do
!$omp end parallel
```

- Use of a built-in function omp_set_num_threads.
  - It changes the default number of threads in the program.
  - It affects all the subsequent parallel regions without the num_threads clause.

Approach 3

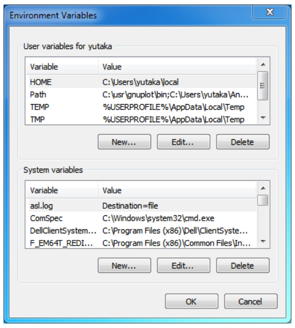
Linux and Mac OS X:

```
$ export OMP_NUM_THREADS=5
```

or

```
$ OMP_NUM_THREADS=5 ./a.out
```

Windows:



User variables for yutaka

Variable	Value
HOME	C:\Users\yutaka\local
Path	C:\usr\gnuplot\bin;C:\Users\yutaka\An...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
asl.log	Destination=file
ComSpec	C:\Windows\system32\cmd.exe
DellClientSystem...	C:\Program Files (x86)\Dell\ClientSyste...
F_EM64T_REDI...	C:\Program Files (x86)\Common Files\In...

- Use of an environmental variable **OMP_NUM_THREADS**.
  - It means you don't have to change the program. You can just change the system variable.
  - In Linux and Mac OS X, this variable is effective only in the session. Write the variable in your Bash-profile.
  - In Windows, open the computer's property to set the variable.

OpenMP is not perfect.

- Suitable: A task can be split into several *independent* computations.
  - Not directly applicable if there are data-dependencies.

```
do i=3,n
  x(i)=x(i-1)+x(i-2)
end do
```

- Even if OpenMP is applicable, it is not always working well.
  - There is always overhead to control/synchronize the threads.
- OpenMP is useful only if the overhead can be ignored e.g. heavy computations repeated many times.

BLUPF90 programs and parallelization

- BLUPF90 programs depends on parallel libraries and modules.
  - A genomic module depends on Intel MKL i.e. optimized BLAS & LAPACK subroutines. MKL is parallelized by OpenMP.
  - The module also uses OpenMP directives.
  - YAMS (a sparse matrix library) calls MKL as well.
  - BLUPF90IOD2 (a commercial product) supports parallel computing with OpenMP.
- Please make sure how many threads you will be actually using before running the parallel programs.