Introduction to PARALLEL COMPUTING with OpenMP and MPI

2012 May 24
The speaker

Davide Del Vento, PhD in Physics
Consulting Services, Software Engineer
NCAR - CISL
http://www2.cisl.ucar.edu/uss/csg
office: Mesa Lab, Room 55G
phone: (303) 497-1233
email: ddvento@ucar.edu

ExtraView Tickets: cislhelp@ucar.edu
Getting ready

ssh mirageN.ucar.edu

cp -r ~ddvento/MPI .

cp -r ~ddvento/OpenMP .

cd MPI

source mirage.source.{bash, tcsh}

➲ Done!!
Table of Contents

くなりました Parallel Computing

Why we need it

Main paradigms (shared/private resources)

MPI (Message Passing Interface)

OpenMP (Open Multi-Processing)

References
Parallel what?
Parallel what?
Parallel why?

- Problems in science are often "too big" to solve on one processor core
- Power concerns limit performance increases through continued frequency scaling. Multi-core processors are now the norm.
- Moore's Law end is in sight
- Wirth's law
  
  *Software gets slower, faster than hardware gets faster.*
Shared vs Private Resources
Shared vs Private Resources
Concurrency and ordering

6 * 9 + (7 - 2) * (3 + 1) = ?

total ordering: 
6*9=54, 7-2=5, 3+1=4, 5*4=20, 20+54=74

partial ordering (exploiting parallelism):

\[
\begin{align*}
6*9 &= 54 \\
7-2 &= 5 \\
3+1 &= 4 \\
5*4 &= 20 \\
54+20 &= 74
\end{align*}
\]
Serial Matrix Multiplication

INTEGER:: i, j, k
DO i=0, DIM
  DO j=0, DIM
    DO k=0, DIM
      A(j, i) = A(j, i) + B(k, i) * C(j, k)
    ENDDO
  ENDDO
ENDDO
ENDDO

\[
\begin{array}{cccc}
  a & b & c & d \\
  f & e & & \\
  g & & & \\
  h & & & \\
\end{array}
\]
\[
\begin{array}{cccc}
  x & & & \\
  & & & \\
  & & & \\
  & & & \\
\end{array}
\]
\[
\begin{array}{cccc}
  & & & \\
  & & & \\
  & & & v \\
  & & & \\
\end{array}
\]

v = a*e + b*f + c*g + d*h
Amdahl's law example

- if 95% of a program can be parallelized
- ...but remaining 5% cannot
- theoretical maximum speed-up is...
- ...with infinite processors
- ...and no overhead
Amdahl's law example

- if 95% of a program can be parallelized
- ...but remaining 5% cannot
- theoretical maximum speed-up is...
- ...with infinite processors
- ...and no overhead

20
Amdahl's law

- if a proportion (in time) $P$ of a code
- can get a speedup $S$, then
- total speedup will be (less than)

\[
\frac{1}{(1 - P) + \frac{P}{S}}
\]
Amdahl's law

- if a proportion (in time) $P$ of a code
- can get a speedup $S$, then
- total speedup will be (less than)

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

- improving $P$ is more important than improving $S$ (but often more difficult)
Why this is important

- WRF (Weather Research Forecast) ConUS (CONtinental USa) 2.5km 6h benchmark forecast

- On a single P6, in theory could run in 4h – in practice, it takes about 40h
Why this is important

- WRF (Weather Research Forecast) ConUS (CONtinental USa) 2.5km 6h benchmark forecast
- On a single P6, in theory could run in 4h – in practice, it takes about 40h
- 4-nodes (128 cores) it completes in 0.6h
- on 64-nodes (1024 core) in 9 min
Message Passing Interface

➡ Goals:
   - portability
   - scalability
   - high performance

➡ Facts:
   - specification, not implementation
   - standard *de facto*
   - closer (and in principle optimized for) hw
   - programmer splits workload **and data**
MPI APIs

- Emphasis on **messages**
- MPI is huge (about 125 functions in MPI-1, about **400(!)** in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking
Init and Finalize

 ➤ MPI start-up (should be the first thing in a program)

\[
\text{MPI\_INIT}(\text{ierror})
\]
\[
\text{INTEGER } \text{ierror}
\]
! Fortran last argument is the error
! There are not any other arguments for INIT

 ➤ MPI tear-down (should be the last thing in a program – beware of early STOPs!)

\[
\text{MPI\_FINALIZE}(\text{ierror})
\]
\[
\text{INTEGER } \text{ierror}
\]
MPI APIs

- Emphasis on messages
- MPI is huge (about 125 functions in MPI-1, about 400(!) in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking


**Size, Rank and Communicator**

⊙ How many processes are there?
⊙ Important question to decide how to split workload and data on different machines
⊙ Answer:

```
MPI_COMM_SIZE(c, size, ierror)
INTEGER c, size, ierror
```
⊙ Assume for now c=MPI_COMM_WORLD


Size, Rank and Communicator

➲ How do you identify different processes?
➲ The rank is an integer that identifies it inside the communicator

MPI_COMM_RANK(c, rank, ierror)
INTEGER c, size, ierror

➲ Continue to assume c=MPI_COMM_WORLD
Size, Rank and Communicator

- In MPI, it is possible to divide the total number of processes into groups, called COMMUNICATORs.

- The processes may belong to more than one communicator, giving greater flexibility.

- The communicator that includes all processes is called **MPI_COMM_WORLD**.
MPI APIs

- Emphasis on **messages**
- MPI is huge (about 125 functions in MPI-1, about **400(!)** in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking
MPI Fortran example (boilerplate)

PROGRAM example
   IMPLICIT NONE
   INCLUDE 'mpif.h'
   INTEGER:: size, rank, ierr

   CALL MPI_INIT(ierr)
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
   PRINT *, 'I am the', rank, 'th process', &
            ' among the', size, 'running'
   CALL MPI_FINALIZE(ierr)
END PROGRAM
How to run MPI programs

- Different MPI implementations have slightly different ways
- Different machines have different scheduling and/or queuing systems
- You usually need an `mpirun` wrapper like the following (this is what we do on mirage)

```bash
mpirun -np 8 ./mpi1_parallel
```
- On bluefire, we use

```bash
mpirun.lsf ./mpi1_parallel
```
and specify the `-np 8` on the queuing system
Hands on
Output:

Assuming the launcher scheduled 4 processes, the output is...

I am the 1-th process among the 4 running.
I am the 0-th process among the 4 running.
I am the 2-th process among the 4 running.
I am the 3-th process among the 4 running.
Output:

➡ Assuming the launcher scheduled 4 processes, the output is...

I am the 1-th process among the 4 running.
I am the 0-th process among the 4 running.
I am the 2-th process among the 4 running.
I am the 3-th process among the 4 running.

➡ Why four lines??
Output:

- Assuming the launcher scheduled 4 processes, the output is...

  I am the 1-th process among the 4 running.
  I am the 0-th process among the 4 running.
  I am the 2-th process among the 4 running.
  I am the 3-th process among the 4 running.

- Why four lines??
- Note that the rank is different! In MPI the resources (variables) are **always** private!!
Output:

- Assuming the launcher scheduled 4 processes, the output is...

  I am the 1-th process among the 4 running.
  I am the 0-th process among the 4 running.
  I am the 2-th process among the 4 running.
  I am the 3-th process among the 4 running.

- Why four lines??
- Note that the rank is different! In MPI the resources (variables) are **always** private!!
- There is not any special order in the I/O!
How to share data?

→ In MPI the resources are always private!!
How to share data?

- In MPI the resources are always private!!
- How can I share data among processors?
How to share data?

➲ In MPI the resources are **always** private!!

➲ How can I share data among processors?

➲ Answer: you **cannot**!
How to share data?

- In MPI the resources are always private!!

- How can I share data among processors?

- Answer: you cannot!

- But you can pass messages, this is MPI, i.e. Message Passing Interface, after all
MPI Messages

Point-to-point
Broadcast (inside the communicator)
You need:
- Data to be sent, includes:
  - Raw data
  - Data type
  - Size
- Destination, includes:
  - rank of the receiving process (if point-to-point)
  - communicator, shared between sender and receiver(s)
- Tag, an identifier for additional checks
MPI APIs

- Emphasis on messages
- MPI is huge (about 125 functions in MPI-1, about 400(!) in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking
Point-to-Point Send

MPI_SEND(buf, count, dt, &
    dest_rank, tag, comm, ierror)
<type> buf(*)
INTEGER count, dt, dest_rank, tag, comm, ierror

➲ Data to be sent: buf, count, dt
➲ Destination: dest_rank, comm
➲ Identifier: tag
➲ Error check: ierror
**Point-to-Point Receive**

MPI_RECV(buf, count, dt, &
       src_rank, tag, comm, status, ierror)
<type> buf(*)
INTEGER count, dt, dest_rank, tag, comm, ierror, status(MPI_STATUS_SIZE)

- Data to be received: buf, count, dt
- Source (not **mailbox-like**!): src_rank, comm
- Identifier: tag
- Check for completion: status
- Error check: ierror
## MPI datatypes in Fortran

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
MPI in action (less trivial)

IF (rank == sender) THEN
  value = 12.24
  call MPI_SEND(value, 1, MPI_DOUBLE_PRECISION, &
                 receiver, tag, MPI_COMM_WORLD, err)
  PRINT *, 'I am', rank, 'and I sent', value
ELSE IF (rank == receiver) THEN
  call MPI_RECV(value, 1, MPI_DOUBLE_PRECISION, &
                sender, tag, MPI_COMM_WORLD, st, err)
  PRINT *, 'I am', rank, 'and I received', value
ELSE
  PRINT *, 'I am', rank, 'among', size, &
          'not doing anything'
ENDIF
Results

$ mpirun -np 4 mpi2_sndrcv
I am 1 my value is 0.00000000000000
I am 2 my value is 0.00000000000000
I am 2 among 4 not doing anything
I am 3 my value is 0.00000000000000
I am 3 among 4 not doing anything
I am 0 my value is 0.00000000000000
I am 0 and I sent 12.2399997711182
I am 1 and I received 12.2399997711182

✎ Probably the idle processes end first (but it is not guaranteed that the I/O is time-ordered, do **NOT** rely on it for debugging!)
mailbox-like behavior

➢ To receive from any source: use `MPI_ANY_SOURCE` as `src_rank`  

➢ To receive with any tag: use `MPI_ANY_TAG` as `tag`  

➢ If you use this feature, be careful not to confuse which data is which one  

➢ Actual source and tag are returned in the `status` parameter  

➢ Still `MPI_RECV()` call required (not really mailbox)
MPI APIs

- Emphasis on messages
- MPI is huge (about 125 functions in MPI-1, about 400(!) in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking
One-to-All: Broadcast

MPI_BCAST(buf, count, dt, snd_rank, comm, ierror)
<type> buf(*)
INTEGER count, dt, snd_rank, comm, ierror

➤ All the ranks have to call the same MPI_Bcast()

➤ The data in the buf for the snd_rank is sent (and will be found) in all the buf for the other ranks

➤ Closest thing to “shared” that you can find in MPI
MPI in action (broadcast)

IF (rank == sender) THEN
    value = 12.24
ENDIF
PRINT *, 'I am', rank, 'my value is', value

call MPI_BCAST(value, 1, MPI_DOUBLE_PRECISION, &sender, MPI_COMM_WORLD, ierr)

PRINT *, 'I am', rank, 'my new value is', value
Results

$ mpirun -np 4 mpi3_broadcast
I am 1 my value is 0.0000000000000000
I am 2 my value is 0.0000000000000000
I am 3 my value is 0.0000000000000000
I am 0 my value is 12.239997711182
I am 0 my new value is 12.239997711182
I am 1 my new value is 12.239997711182
I am 2 my new value is 12.239997711182
I am 3 my new value is 12.239997711182

⚠️ Data is sent to all, but it is not shared
Results

$ mpirun -np 4 mpi3_broadcast

I am 1 my value is 0.0000000000000000
I am 2 my value is 0.0000000000000000
I am 3 my value is 0.0000000000000000
I am 0 my value is 12.239997711182
I am 0 my new value is 12.239997711182
I am 1 my new value is 12.239997711182
I am 2 my new value is 12.239997711182
I am 3 my new value is 12.239997711182

➲ Data is sent to all, but it is **not shared**
➲ if one task later changes its value, the other tasks **do not** see the change, *unless broadcast is called again*
Reduce

- Reduction: think as replacing the comma, in a list or in an array, with something else

- E.g. the addition fold for the list \([1,2,3,4,5]\) is \(1 + 2 + 3 + 4 + 5\) (replace \(,\) with \(+\))

- Serially

```plaintext
dot=0
DO  i=1,len
    dot = dot + a(i)
ENDDO
```
## Supported MPI Reduce

<table>
<thead>
<tr>
<th>MPI reduce name</th>
<th>MPI supported Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM, MPI_PROD</td>
<td>MPI_REAL, MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_COMPLEX</td>
</tr>
<tr>
<td>MPI_MAX, MPI_MIN</td>
<td>MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>MPI_MAXLOC, MPI_MINLOC</td>
<td>MPI_2REAL, MPI_2INTEGER, MPI_2DOUBLE_PRECISION</td>
</tr>
<tr>
<td>MPI_LAND, MPI_LOR, MPI_LXOR</td>
<td>MPI_LOGICAL</td>
</tr>
<tr>
<td>MPI_BAND, MPI_BOR, MPI_BXOR</td>
<td>MPI_INTEGER, MPI_BYTE</td>
</tr>
</tbody>
</table>

⚠️ You can create custom ones
Reduce calls

MPI_REDUCE(sbuf, rbuf, count, dt, op, dest_rank, comm, ierr)
<type> sbuf(*), rbuf(*)
INTEGER count, dt, op, rcv_rank, comm, ierr

- Data to be sent/received: sbuf, rbuf, count, dt
- Destination: dest_rank, comm
- Operation to be performed: op
Reduce/Broadcast

MPI_ALLREDUCE(sbuf, rbuf, count, dt, op, comm, ierr)
<type> sbuf(*), rbuf(*)
INTEGER count, dt, op, comm, ierr

➤ No single destination, everybody will have the result

➤ Exercise: write your own example!
**Splitting Workload and Data in MPI**

- Sum of two big arrays, sequential code

```fortran
INTEGER SIZE, I
PARAMETER(SIZE=huge number)
DOUBLE PRECISION a(SIZE), b(SIZE), c(SIZE)

CALL read_from somewherewhere(a, b)

DO I=0, SIZE
    c(i) = a(i) + b(i)
END DO
```

Davide Del Vento
**Splitting Workload and Data in MPI**

```plaintext
INTEGER pieces, rank, start, stop, SIZE, split, I
PARAMETER (SIZE=huge number)
DOUBLE PRECISION a(SIZE), b(SIZE), c(SIZE)
CALL read_from_somewhere(a, b)
CALL MPI_Comm_size(MPI_COMM_WORLD, pieces, ierr)
CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

split = SIZE / pieces
start = rank * split
stop = (rank + 1) * split

DO I=start, stop
   c(i) = a(i) + b(i)
END DO
```

Do you see any problems?
A real solution should

- Include missing details:
  - who loads the data (not everybody)
  - data transfer to everybody
  - transfer the result back

- Remotely allocate the space **only** for the needed data (less memory overhead)

- Probably split the loading of the data and publishing of the result among processes, instead of overload one
MPI APIs

- Emphasis on messages
- MPI is huge (about 125 functions in MPI-1, about \textbf{400(!)} in MPI-2)
- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, buffered, ready, synchronous
    - blocking, non-blocking
Variations: Blocking Sends

- Blocking = returns when the memory where the message was, can be safely (re)used.
- Standard `MPI_Send()` Completion does NOT indicate whether the message has been sent out from the source machine (but it may block until the message is received)
- Buffered `MPI_Bsend()` Guarantees the message being buffered
Variations: Blocking Sends/Receive

- **Ready** `MPI_Rsend()`
  Wait for a matching receive being **posted**

- **Synchronous** `MPI_Ssend()`
  Wait for the matching receive being **started**

- **Standard (Blocking)** `MPI_Recv()`
  Wait for the message being received and copied in the memory, **ready for use**
MPI APIs

- Emphasis on messages

- MPI is huge (about 125 functions in MPI-1, about 400(!) in MPI-2)

- MPI is small
  - 9 concepts (~ functions):
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, buffered, ready, synchronous
    - blocking, non-blocking
Variations: Non-Blocking Sends

- Non-Blocking = initiate the communication and returns immediately; the memory where the message is can **NOT** be (re)used.

- Standard `MPI_Isend(.. MPI_Request *req)`
  Completion does **NOT** indicate whether the message has been buffered (nor sent out or received)

- Buffered `MPI_Ibsend(.. MPI_Request *req)`
  On completion the message is buffered
Variations: Non-Blocking Sends/Receive

» Ready `MPI_Irsend(.. MPI_Request *req)`
   On completion, a matching receive is **posted**

» Synchronous `MPI_Issend(.. MPI_Request *req)`
   On completion, a matching receive is **started**

» Non-Blocking `MPI_Irecv(.. MPI_Request *req)`
   Returns immediately, the message is not yet in the memory, **not ready for use**
Checking for completion

You can wait for completion or just test for it:

```
int MPI_Wait(MPI_Request *rq, MPI_Status *st);
int MPI_Test(MPI_Request *rq, int flag, MPI_Status *st);
```

- **MPI_Wait()** blocks, like the corresponding (standard, buffered, ready, synchronous) blocking send/receive

- **MPI_Test()** returns immediately, with the flag raised on “completion”
Errors

✎ I didn't check errors to have the sources short in the slides. In real life you should!

✎ (almost) all MPI routines “return” an error value as last, optional, argument.

✎ MPI_SUCCESS
No error. MPI routine completed successfully.

✎ MPI_ERR_COMM
Invalid communicator. A common error is to use a null communicator in a call.
- **MPI_ERR_COUNT**
  Invalid count argument. Count arguments must be non-negative (zero is often valid)

- **MPI_ERR_TYPE**
  Invalid datatype argument. May be an uncommitted custom MPI_Datatype

- **MPI_ERR_BUFFER**
  Invalid buffer pointer. Usually a null buffer

- **MPI_ERR_ROOT**
  Invalid root (rank). Ranks must be positive and smaller than the size of the communicator
Scaling up
Scaling up
MPI gotchas

➤ If you use communicators (other than MPI_COMM_WORLD) the same process could have different ranks

➤ Decomposing data and balancing the work-load between processes can be challenging, but is critical to performance

➤ Sometimes difficult to debug

➤ Deadlocks (just concurrency, not only for MPI...)

Davide Del Vento

NCAR Computational & Information Systems Laboratory
Deadlocks in MPI

 peny I'm waiting for you, while you are waiting for me. Who'll give up?

Rank = 0

Rank = 1

sendbuf

recvbuf

recvbuf

sendbuf
Deadlocks in MPI

» I'm waiting for you, while you are waiting for me. Who'll give up?

» Usually nobody, and the program hangs (The Law of the Uneaten Spinach)
Deadlocks in MPI

- The order, in which the send and receive are posted, does matter (that's clear if everyone is receiving before sending, but may happen in other cases too)

- Non-blocking send and receive do help avoiding deadlocks (but might introduce bugs, if not used properly)

- The size of the involved data and the size of the system buffers do matter (but buffers are not a silver bullet)
Last, but not least

- Measure wallclock time (only) with:
  
  ```c
  MPI_Wtime() and MPI_Wtick()
  ! those are the only two without the error argument
  ```

- Other useful functions to check:
  
  ```c
  MPI_Barrier(),
  MPI_Scatter(), MPI_Gather(),
  MPI_Get_count(), MPI_Probe(),
  ```

- Beware of limited implementations:
  
  ```c
  MPI_Comm_spawn(), MPI_Comm_connect(), etc.
  not available on our IBM supercomputers
  ```
OpenMP overview
OpenMP facts

➲ Very easy to use!
➲ Based on preprocessor directives
➲ Same code for both serial and parallel applications (with caveats)
➲ Automatically distributes workload
➲ Synchronization between a subset of threads is not allowed.
➲ Runs **ONLY** in shared-memory
➲ Doesn't help if our problem does not fit in memory
OpenMP library routines

- `OMP_GET_NUM_THREADS()`
  - `OMP_SET_NUM_THREADS()`
  - returns/sets the current number of threads

- `OMP_GET_THREAD_NUM()`
  - returns the thread ID of the current thread

- `OMP_GET_WTIME()`
  - returns the elapsed wallclock time in sec

- There are several more
OpenMP directives

- Syntactically directives are just comments
- Fortran 90
  !$OMP PARALLEL
- Fortran 77
  c$OMP PARALLEL
- C/C++
  #pragma omp parallel
First OpenMP directive

$!$OMP PARALLEL

- forms a team of $N$ threads (where $N$ is set by the OMP_NUM_THREADS environmental variable, or a call to omp_set_num_threads() routine) and starts the execution of the parallel region
- the semantics is (almost) the same as in a serial program
- but there are some differences (can you anticipate any?)
OpenMP overview
First Example

PROGRAM EXAMPLE
INTEGER:: MYTHR, NTHRS

CALL OMP_SET_NUM_THREADS(8)

!$OMP PARALLEL PRIVATE (MYTHR, NTHRS)
! note: PRIVATE will be described later
NTHRS = OMP_GET_NUM_THREADS()
MYTHR = OMP_GET_THREAD_NUM()
PRINT *, 'I am', MYTHR, 'of', NTHRS

!$OMP END PARALLEL
END PROGRAM
**Second OpenMP directive**

- ** !$OMP SINGLE**
  - within a parallel section, the SINGLE construct specifies that the block is executed only by one thread in the team (not necessarily the master)
  - can be useful when you don't want to “stop” the parallel section just to do minor activities (e.g. printing a message)
  - beware (especially for debugging) that the single section may be executed by the first thread that reaches it, while other threads might be left behind doing “previous” work
Second Example

!$OMP PARALLEL PRIVATE (MYTHR, NTHRS)
! note: PRIVATE will be described later
NTHRS = OMP_GET_NUM_THREADS()
MYTHR = OMP_GET_THREAD_NUM()
PRINT *, 'I am', MYTHR, 'of', NTHRS

!$OMP SINGLE
PRINT *, 'I am', MYTHR, '(in single)'
!$OMP END SINGLE

!$OMP END PARALLEL
Most important OpenMP directive

ё !$OMP PARALLEL + !$OMP DO
!$OMP PARALLEL DO

● forms a team of $n$ threads (where $n$ is set by the OMP_NUM_THREADS environmental variable, or a call to omp_set_num_threads() routine) and starts the parallel execution.... like  !$OMP PARALLEL

● ...but instead of duplicating the work, it splits the load of the loop among the threads

● incredibly easy to parallelize an existing serial program
Most important OpenMP directive

`!$OMP PARALLEL + !$OMP DO
!$OMP PARALLEL DO`

- forms a team of $N$ threads (where $N$ is set by the `OMP_NUM_THREADS` environmental variable, or a call to `omp_set_num_threads()` routine) and starts the parallel execution.... like `!$OMP PARALLEL`
- ...but instead of duplicating the work, it splits the load of the loop among the threads
- incredibly easy to parallelize an existing serial program
- and incredibly easy to get it wrong :-(}
Third Example

PROGRAM EXAMPLE
INTEGER:: MYTHREAD, I
CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL PRIVATE (MYTHR)
! note: PRIVATE will be described later
MYTHR = OMP_GET_THREAD_NUM()
!$OMP DO
DO I=1,10
       PRINT *, 'Looping', I 'by', MYTHREAD
ENDDO
!$OMP END DO
!$OMP END PARALLEL
END PROGRAM
Serial Matrix Multiplication

DO i = 1, N
  DO J = 1, M
    DO K = 1, P
      C(i,K) = C(i,K) + A(i,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO

\[ v = ae + bf + cg + dh \]
Exercise:

Make a parallel version of Matrix Multiplication in OpenMP (the serial version is already on your computer)
Matrix Multiplication in OpenMP

CALL OMP_SET_NUM_THREADS(whatever)
!$OMP PARALLEL DO
! note: loops on J and K are serial
DO i = 1, N
   DO J = 1, M
      DO K = 1, P
         C(i,K) = C(i,K) + A(i,J) * B(J,K)
      ENDDO
   ENDDO
ENDDO
ENDDO
ENDDO
!$OMP END PARALLEL DO
Matrix Multiplication in OpenMP

CALL OMP_SET_NUM_THREADS(whatever)
 !$OMP PARALLEL DO
 ! note: loops on J and K are serial
 DO i = 1, N
   DO J = 1, M
     DO K = 1, P
       C(i,K) = C(i,K) + A(i,J) * B(J,K)
     ENDDO
   ENDDO
 ENDDO
 ENDDO
 !$OMP END PARALLEL DO

➔ Drawback: parallelize only the loop on i
CALL OMP_SET_NUM_THREADS(whatever) 
!$OMP PARALLEL DO 
! note: loops on J and K are serial 
DO i = 1, N 
  DO J = 1, M 
    DO K = 1, P 
      C(i,K) = C(i,K) + A(i,J) * B(J,K) 
    ENDDO 
  ENDDO 
ENDDO 
ENDDO 
!$OMP END PARALLEL DO 

➲ Drawback: parallelize only the loop on i 
➲ Possible solution: merge the loops e.g. H=1,N*M*P (a more convenient one later – only for OpenMP 3.0 )
Exercise: parallelize the following

DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO

a( 1,:)     1.  101.  201.
a( 2,:)     2.  102.  202.
a( 3,:)     3.  103.  203.
a( 4,:)     4.  104.  204.
a( 5,:)     5.  105.  205.
a( 6,:)     6.  106.  206.
a( 7,:)     7.  107.  207.
Wrong solution

CALL OMP_SET_NUM_THREADS(4)

!$OMP PARALLEL DO
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO

!$OMP END PARALLEL DO

a( 1,:) 1. 101. 0.
a( 2,:) 2. 102. 0.
a( 3,:) 0. 103. 0.
a( 4,:) 4. 0. 204.
a( 5,:) -1. 105. 205.
a( 6,:) 6. 106. 206.
a( 7,:) -1. 0. 207.
Wrong solution

CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
!$OMP END PARALLEL DO

- By default TEMP and J are SHARED (among all threads)
Wrong solution

CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
DO I = 0, N*M-1
  TEMP = I/M + 1
  J = MOD(I, M) + 1
  CALL SLEEP(1)
  A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
!$OMP END PARALLEL DO

- By default TEMP and J are SHARED
- Then modified
Wrong solution

CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
!$OMP END PARALLEL DO

- By default TEMP and J are SHARED
- Then modified
- I do some other work (here I'm cheating)
Wrong solution

CALL OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL DO
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
 !$OMP END PARALLEL DO

- By default TEMP and J are SHARED
- Then modified
- I do some other work (here I'm cheating)
- While I work, another thread can change my values
Wrong solution

CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
!$OMP END PARALLEL DO

- By default TEMP and J are SHARED
- Then modified
- I do some other work (here I'm cheating)
- While I work, another thread can change my values
- And I finally use the variables, with possibly the wrong values!
Right solution

CALL OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL DO PRIVATE(TMP, J)
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TMP, J) = TEMP + 100*(J-1)
ENDDO
!$OMP END PARALLEL DO

- I is automatically PRIVATE
- A, N and M are automatically SHARED
OpenMP clauses

- Clauses are options for the directives

- SHARED and PRIVATE clauses for the PARALLEL directive

- There is DEFAULT clause (with useful NONE option)

- other useful clauses are REDUCTION and COLLAPSE

- ...but there are other ones
DEFAULT (NONE) example

CALL OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL DO DEFAULT(NONE)
DO I = 0, N*M-1
    TEMP = I/M + 1
    J = MOD(I, M) + 1
    CALL SLEEP(1)
    A(TEMP,J) = TEMP + 100*(J-1)
ENDDO
 !$OMP END PARALLEL DO

● This does not compile
● You must explicitly specify all variables (TEMP, J and A) as SHARED or PRIVATE
● Like IMPLICIT NONE, it forces you to think
Another example

Suppose I have to do something like:

```fortran
INTEGER :: x, size
x = 0
size = 12
OMP_SET_NUM_THREAD(size)
 !$OMP PARALLEL SHARED(x)
   x = x + 1
 !$OMP END PARALLEL
PRINT *, x
```

What's the result?
Another example

Suppose I have to do something like:

```fortran
INTEGER :: x, size
x = 0
size = 12
OMP_SET_NUM_THREAD(size)
 !$OMP PARALLEL SHARED(x)
   x = x + 1
 !$OMP END PARALLEL
PRINT *, x
```

What's the result? $x = size = 12$?
Wrong! The result is unknown!

- Could be everything between 1 and the number of running threads (i.e. \text{size})
Wrong! The result is unknown!

Could be everything between 1 and the number of running threads (i.e. size)
The **CRITICAL** directive makes only a single thread at a time running in the section:

```fortran
INTEGER :: x, size
x = 0
size = 12
OMP_SET_NUM_THREAD(size)
!$OMP PARALLEL SHARED(x)
  !$OMP CRITICAL
  x = x + 1
  !$OMP END CRITICAL
!$OMP END PARALLEL
```

What's the result, now?
CRITICAL Directive

- Only one thread at time in the critical section

Processor 2

1
+1
= 2
2

Memory

0 0 0
1 1 1
1 2

Processor 1

0
+1
= 1
1

Processor 3 ...

Davide Del Vento
The **CRITICAL** directive makes only a single thread at a time running in the section:

```plaintext
INTEGER :: x, size
x = 0
size = 12
OMP_SET_NUM_THREADS(size)
!$OMP PARALLEL SHARED(x)
   !$OMP CRITICAL
      x = x + 1
   !$OMP END CRITICAL
!$OMP END PARALLEL
```

Is it any faster than serial code?
Reduction

✦ So, is it possible to parallelize something like the following?

```fortran
REAL :: dot, a(len), b(len)
dot = 0.0

DO i=1,len
  dot = dot + a(i) * b(i)
ENDDO
```
Reduction

➲ So, is it possible to parallelize something like the following?

REAL :: dot, a(len), b(len)
dot = 0.0

DO i=1,len
    dot = dot + a(i) * b(i)
ENDDO

➲ Idea: do the products in parallel, then the sums in critical (or maybe some sub-sums to exploit more parallelism)
Reduction

So, is it possible to parallelize something like the following?

```plaintext
REAL :: dot, a(len), b(len)
dot = 0.0
 !$OMP PARALLEL DO REDUCTION(+:dot)
 DO i=1,len
    dot = dot + a(i) * b(i)
 ENDDO
 !$OMP END PARALLEL DO
```

Exactly what REDUCTION does for you

Works with: MAX, MIN, IAND, IOR, Ieor, +, *

-, AND, OR, EQV, NEQV
Rules of thumb

➤ Loop indexes are automatically PRIVATE
➤ Everything “local or temporary” should be PRIVATE (or FIRSTPRIVATE or LASTPRIVATE if its value is used outside the loop, before or after respectively)
➤ Everything “persistent” and/or used for different values of the loop index should be SHARED
➤ a SHARED variable that is not an array accessed with the loop indexes, should be written only in a CRITICAL region (serialize, so it's slow)
➤ If you are using CRITICAL, see if REDUCTION is an option (maybe changing the math a little bit)
COLLAPSE clause

- Available in OpenMP 3.0 and later
- Clause for the PARALLEL DO directive
- Cause an automatic “collapse” (merge) of the loops, and thus automatic parallelization of inner loops too
- Most things taken care automatically, but user have to be careful (see next slides)
Matrix Multiplication in OpenMP 3.0

CALL OMP_SET_NUM_THREADS(whatever)
!$OMP PARALLEL DO SHARED(A, B, C) COLLAPSE(3)
! note: i,j,k are automatically private
DO i = 1, N
  DO J = 1, M
    DO K = 1, P
      C(i,K) = C(i,K) + A(i,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO
!OMP END PARALLEL

➲ Now everything is parallel, but...
Matrix Multiplication in OpenMP 3.0

CALL OMP_SET_NUM_THREADS(whatever)
!$OMP PARALLEL DO SHARED(A, B, C) COLLAPSE(3)
! note: i,j,k are automatically private
DO i = 1, N
  DO J = 1, M
    DO K = 1, P
      C(i,K) = C(i,K) + A(i,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO
ENDDO
!OMP END PARALLEL

➲ Now everything is parallel, but...
➲ C is shared and possibly modified by different threads
Matrix Multiplication in OpenMP 3.0

CALL OMP_SET_NUM_THREADS(whatever)
!$OMP PARALLEL DO SHARED(A, B, C) COLLAPSE(2)
! note: i,k are automatically private
DO i = 1, N
  DO K = 1, P
    DO J = 1, M ! J now is the innermost
      C(i,K) = C(i,K) + A(i,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO
!OMP END PARALLEL

平行化新可能，确保正确性（且不使用 CRITICAL）
Fortran common OpenMP directives

- **OMP PARALLEL**
  forms a team of threads and starts parallel execution (with **SHARED** and **PRIVATE**)

- **OMP DO**
  as above and distributes loop's iterations among the threads (with **REDUCTION** and **COLLAPSE**)

- **OMP CRITICAL**
  only a single thread at time will execute this section (they all will execute it at different times)
Less common OpenMP directives

➲ OMP SINGLE
this section will be executed only one time, by one of the threads

➲ OMP BARRIER
guarantees that all the threads will pass the barrier at the same time (early birds will wait)

➲ OMP SECTIONS / OMP SECTION
more control on work distribution among threads

➲ There are other directives not shown here
Dining Philosophers Problem
Deadlocks in OpenMP

- The fork (a chopstick in a variant of the problem) is a semaphore: something you need to process your data, and also others need it at the same time!
- You need two forks to eat, but you have to get one at a time
- If everybody gets one, and just wait for the other... nobody will eat
Deadlocks in OpenMP

!$OMP PARALLEL SHARED(FORKS)  
!$OMP CRITICAL  
IF a-fork-is-available  
    get-the-fork  
    !the same twice: these are not Italians  
END IF  
!$OMP END CRITICAL  
IF I-have-both-fork  
    eat  
ELSE  
    release-the-fork-if-you-have-one  
    think  
ENDIF  
!$OMP END PARALLEL
Homework: write a good solution

- Good means:

- Without deadlocks, namely everybody waiting forever for something

- Without starvation, namely all the philosophers must eventually eat, otherwise someone would die
MPI-OpenMP hybrid programming

- MPI and OpenMP can and actually often are used together
- Loop-level OpenMP parallelism on the node
- Message Passing parallelism among nodes
**PGAS**

- Partitioned Global Address Space: global memory address space, partitioned in portions local to each processor
- Co-array Fortran program is replicated a number of times (asynchronously) in *images*. The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to provide a concise representation of references to data that is spread across images.
References
(plus wikipedia and Google)

➡ MPI:
http://www.msi.umn.edu/tutorial/scicomp/general/MPI/mpi_intro.html
http://www.mpi-forum.org/
http://www-unix.mcs.anl.gov/mpi/
http://www.redbooks.ibm.com/abstracts/sg245380.html
http://webct.ncsa.uiuc.edu:8900/public/MPI/ (registration needed)

➡ OpenMP
http://openmp.org/
https://computing.llnl.gov/tutorials/openMP/

➡ Illustrations from
http://www.opencliparts.org
You are free:

- to Share — to copy, distribute, display, and perform the work

- to Remix — to make derivative works

Under the following conditions:

- Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

- Noncommercial. You may not use this work for commercial purposes.

- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

- Any of the above conditions can be waived if you get permission from the copyright holder.

- Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.

Your fair use and other rights are in no way affected by the above.
This is a human-readable summary of the Legal Code (the full license).