Parallelism II

John Cavazos and Tristan Vanderbruggen

Dept of Computer & Information Sciences

University of Delaware
Lecture Overview

- Introduction
- OpenMP
  - Model
  - Language extension: directives-based
  - Step-by-step example
- MPI
  - Model
  - Runtime Library
  - Step-by-step example
- Conclusion / Q&A
Codes:
- [http://www.cis.udel.edu/~cavazos/hpc-ll.zip](http://www.cis.udel.edu/~cavazos/hpc-ll.zip)
- [https://github.com/tristanvdb/hpc-lecture](https://github.com/tristanvdb/hpc-lecture)

To connect on Mills and start using it:
- `$> ssh login@mills.hpc.udel.edu$
- `$> workgroup -g your_workgroup$
- `$> vpkg_devrequire gcc$
- `$> vpkg_devrequire openmpi$
2 - OpenMP

- Model
- Language
- Step-by-step Example
- Construct & Clause
- Q&A
2.1 - OpenMP: Model

- Shared Memory Model:
  - multi-processor/core

Source: https://computing.llnl.gov/tutorials/openMP/
2.1 - OpenMP: Model

- **Thread-level Parallelism:**
  - parallelism through threads
  - typically: the number of threads match the number of cores

- **Fork - Join Model:**

Source: https://computing.llnl.gov/tutorials/openMP/
Explicit Parallelism:
- offer full control over parallelization to the programmer
- can be as simple as inserting compiler directives in a serial program
- or, as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks
2.2 - OpenMP: Language

- OpenMP is not exactly a language. It is an extension for C and Fortran.
- OpenMP is a directive-based language.
- It works by annotating sequential code.

Source: https://computing.llnl.gov/tutorials/openMP/
2.2 - OpenMP: Language

- in C, it uses pragmas

```c
#pragma omp construct [clause, ...]
```

- in Fortran, it uses sentinels (!$omp, C$omp, or *$omp):

```fortran
!$OMP construct [clause, ...]
```
● **constructs** are functionalities of the language
● **clauses** are parameters to those functionalities
● **construct + clauses = directive**
Two examples:
- the classic HelloWorld
- a matrix multiplication
```c
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>

int main() {
    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {

        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = \%d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = \%d\n", nthreads);
        }

        } /* All threads join master thread and disband */
}
```
OpenMP: Environment Variables

- OpenMP has a set of environment variables that control the runtime execution
  - `OMP_NUM_THREADS=num`
    - to specify the default number of threads an OpenMP parallel region should contain
  - `OMP_SCHEDULE=algorithm`
    - algorithm = dynamic or static
    - the algorithm to be use for scheduling
2.3 (a) - OpenMP: Hello World

- **Compile:**
  - `$> gcc -fopenmp helloworld-omp.c -o helloworld-omp`

- **Run:**
  - `$> qlogin -pe threads 8`
  - `$> cd hpc-II`
  - `$> export OMP_NUM_THREADS=8`
  - `$> ./helloworld-omp`

```
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 3
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 1
Hello World from thread = 4
```
2.3 (b) - OpenMP: Matrix Multiply

\[(AB)_{i,j} = \sum_{k=1}^{P} A_{ik}B_{kj}\]
2.3 (b) - OpenMP: Matrix Multiply

```c
#include "size-def.h"

float A[N][P]; // op 1
float B[P][M]; // op 2
float C[N][M]; // res

int main() {
    unsigned long i, j, k;

    for (i = 0; i < N; i++)
        for (k = 0; k < P; k++)
            A[i][k] = 0.;
    for (k = 0; k < P; k++)
        for (j = 0; j < M; j++)
            B[k][j] = 0.;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            C[i][j] = 0.;
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            for (k = 0; k < P; k++)
                C[i][j] += A[i][k] * B[k][j];

    return 0;
}
```

\[(AB)_{i,j} = \sum_{k=1}^{P} A_{i,k}B_{k,j}\]
#include <omp.h>

#include "size-def.h"

float A[N][P]; // op 1
float B[P][M]; // op 2
float C[N][M]; // res

int main() {
    unsigned long i, j, k;

    for (i = 0; i < N; i++)
        for (k = 0; k < P; k++)
            A[i][k] = 0.;

    for (k = 0; k < P; k++)
        for (j = 0; j < M; j++)
            B[k][j] = 0.;

    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            C[i][j] = 0.;
#pragma omp parallel shared(A,B,C) private(i,j,k)
{

    #pragma omp for schedule (static)
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            for (k = 0; k < P; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return 0;
}
#pragma omp parallel shared(A,B,C) private(i,j,k)
  ○ create a parallel region, fork a \textit{team} of threads (as many as cores)
  ○ arrays A, B, C are \textit{shared} among the threads
  ○ the "iterators" are \textit{private} to each threads
2.3 (b) - OpenMP: Matrix Multiply

- #pragma omp for schedule (static)
  - the following for-loop have to executed in parallel by the **team**
  - the **schedule** clause precise how the iterations have to be divided
    - static/dynamic
    - chunk size
2.3 (b) - OpenMP: Matrix Multiply

- on Intel i7 4 cores
- for 512x512 float matrices
- Sequential: 0.92s
- OpenMp : 0.24s

Speedup of 3.83
But the speedup depends on the input size:

![Graph showing speedup for OpenMP, function of matrices sizes]
2.4 - OpenMP: Construct

- Constructs:
  a. `barrier` is a synchronisation point for all threads in the `team`
  b. the block following `single` will only be executed by one thread of the `team`
  c. the block following `master` will only be executed by the master thread
  d. only one thread of a team can be in a `critical` block at anytime
  e. `sections` define an area of the code where individual `section` directives delimit independant code to be shared across the threads of the `team`
2.4 - OpenMP: Clause

- clauses:
  a. *shared/private* apply to variables list
  b. *default* policy for variables sharing
     - either *shared* or *none*
  c. *firstprivate* take a list of *private* variables to be initialized
  d. *lastprivate* take a list of *private* variables to be copy out
  e. *reduction* take an operation and a list of scalar variables
  f. *num_thread* either
     - from the team to be used
     - in the team
int main() {

    double startTime;

    #pragma omp parallel private (startTime) num_threads(4)
    {
        startTime = omp_get_wtime();
        // Each thread sleep ID second (master thread sleep 0 s)
        while((omp_get_wtime() - startTime) < (double)(omp_get_thread_num()));
        printf("I (%d) finish to count\n", omp_get_thread_num());
        // Each thread will wait other
        #pragma omp barrier
        printf("I (%d) pass the Barrier\n", omp_get_thread_num());

        #pragma omp single
        {
            printf("I (%d) am the only one executing this code\n", omp_get_thread_num());
        }

        #pragma omp master
        {
            printf("I (%d) am the Master\n", omp_get_thread_num());
        }
    }

    return 0;
}
2.4 - OpenMP: Barrier example

```
tristan@tristan-laptop:~/classes/hpc-lecture/lecture2$ ./barrier-omp
I (0) finish to count
I (1) finish to count
I (2) finish to count
I (3) finish to count
I (3) pass the Barrier
I (3) am the only one executing this code
I (2) pass the Barrier
I (0) pass the Barrier
I (1) pass the Barrier
I (0) am the Master
```
#include <omp.h>

int main() {
    int i, n, chunk;
    float a[100], b[100], result;

    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

#pragma omp parallel for default(shared) private(i) schedule(static,chunk) reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);

    return 0;
}
## 2.4 - OpenMP: Construct & Clause

<table>
<thead>
<tr>
<th>Clause</th>
<th>PARALLEL</th>
<th>DO/for</th>
<th>SECTIONS</th>
<th>SINGLE</th>
<th>PARALLEL DO/for</th>
<th>PARALLEL SECTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHEDULE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
3 - MPI

- Model
- Language
- Step-by-step Example
- API
- Q&A
3.1 - MPI: Model

- Distributed Memory, originally
- today implementation support shared memory SMP

Source: https://computing.llnl.gov/tutorials/mpi/
3.2 - MPI: Language

- MPI is an Interface
  - MPI = Message Passing Interface
- Different implementations are available for C / Fortran

<table>
<thead>
<tr>
<th>C Binding</th>
<th>Fortran Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format:</strong></td>
<td><strong>Format:</strong></td>
</tr>
<tr>
<td>rc = MPI_Xxxxxx(param</td>
<td>CALL MPI_XXXXXX(param, ..., ierr)</td>
</tr>
<tr>
<td>Parameter, ...)</td>
<td>call mpi_xxxxxx(param, ..., ierr)</td>
</tr>
<tr>
<td><strong>Example:</strong></td>
<td><strong>Example:</strong></td>
</tr>
<tr>
<td>rc = MPI_Bsend(buf,</td>
<td>CALL MPI_BSEND(buf, count, type, dest, tag, comm,</td>
</tr>
<tr>
<td>count, type, dest,</td>
<td>ierr)</td>
</tr>
<tr>
<td>tag, comm)</td>
<td></td>
</tr>
<tr>
<td><strong>Error code:</strong></td>
<td><strong>Error code:</strong></td>
</tr>
<tr>
<td>Returned as &quot;rc&quot;.</td>
<td>Returned as &quot;ierr&quot; parameter. MPI_SUCCESS if</td>
</tr>
<tr>
<td>MPI_SUCCESS if</td>
<td>successful</td>
</tr>
<tr>
<td>successful</td>
<td></td>
</tr>
</tbody>
</table>

Source: https://computing.llnl.gov/tutorials/mpi/
3.3 - MPI: Step-by-step Example

General MPI Program Structure:

1. **MPI include file**
2. **Declarations, prototypes, etc.**
3. **Program Begins**
   - **Serial code**
4. **Initialize MPI environment**
   - **Parallel code begins**
5. **Do work & make message passing calls**
6. **Terminate MPI environment**
   - **Parallel code ends**
7. **Serial code**
8. **Program Ends**

Source: https://computing.llnl.gov/tutorials/mpi/
#include "mpi.h"

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int numtasks, taskid, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Get_processor_name(hostname, &len);
    printf("Hello from task %d on %s!\n", taskid, hostname);

    if (taskid == 0)
        printf("MASTER: Number of MPI tasks is: %d\n", numtasks);

    MPI_Finalize();

    return 0;
}
3.3 (a) - MPI: Hello World

- Compile
  - `$> mpicc helloworld-mpi.c -o helloworld-mpi`
  - OR
  - `$> gcc -c helloworld-mpi.c -o helloworld-mpi.o`
  - `$> mpicc helloworld-mpi.o -o helloworld-mpi`
  - **Warning:** Select the good toolchain!
3.3 (a) - MPI: Hello World

- Run
  - On one node:
    - `mpirun -n $NB_PROCESS ./helloworld-mpi`
  - On a cluster with qsub (Sun Grid Engine)
    - `qsub -pe mpich $NB_PROCESS mpi-qsub.sh`
    - With mpi-qsub.sh:

```
#!/bin/bash
#
#$ -cwd
#
mpirun -np $NSLOTS ./matmul-mpi
```
#define NRA N /* number of rows in matrix A */
#define NCA P /* number of columns in matrix A */
#define NCB M /* number of columns in matrix B */
#define MASTER 0 /* taskid of first task */
#define FROM_MASTER 1 /* setting a message type */
#define FROM_WORKER 2 /* setting a message type */

int numtasks, /* number of tasks in partition */
taskid, /* a task identifier */
umworkers, /* number of worker tasks */
source, /* task id of message source */
dest, /* task id of message destination */
mttype, /* message type */
rows, /* rows of matrix A sent to each worker */
averow, extra, offset, /* used to determine rows sent to each worker */
i, j, k, rc; /* misc */
double a[NRA][NCA], /* matrix A to be multiplied */
b[NCA][NCB], /* matrix B to be multiplied */
c[NRA][NCB]; /* result matrix C */

MPI_Status status;
MPI initialization:

```c
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
if (numtasks < 2) {
    printf("Need at least two MPI tasks. Quitting...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
}
numworkers = numtasks - 1;
```
Master initialization:

```c
if (taskid == MASTER)
{
    printf("mpi_mm has started with %d tasks.\n",numtasks);
    printf("Initializing arrays...\n");
    for (i=0; i<NRA; i++)
        for (j=0; j<NCA; j++)
            a[i][j] = i+j;
    for (i=0; i<NCA; i++)
        for (j=0; j<NCB; j++)
            b[i][j] = i*j;
}```
3.3 (b) - MPI: Matrix Multiply

/* Send matrix data to the worker tasks */
averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0;
stype = FROM_MASTER;
for (dest=1; dest<=numworkers; dest++)
{
    rows = (dest <= extra) ? averow+1 : averow;
    printf("Sending %d rows to task %d offset=%d\n", rows, dest, offset);
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
3.3 (b) - MPI: Matrix Multiply

/* Receive results from worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++)
{
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
              MPI_COMM_WORLD, &status);
    printf("Received results from task %d\n",source);
}
3.3 (b) - MPI: Matrix Multiply

```c
if (taskid > MASTER)
{
    mtype = FROM_MASTER;
    MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

    for (k=0; k<NCB; k++)
        for (i=0; i<rows; i++)
            {
                c[i][k] = 0.0;
                for (j=0; j<NCA; j++)
                    c[i][k] = c[i][k] + a[i][j] * b[j][k];
            }
    mtype = FROM_WORKER;
    MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
    MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
}
MPI_Finalize();
```
### 3.4 (b) - MPI API

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>MPI_Init (&amp;argc,&amp;argv)</code></td>
</tr>
<tr>
<td>Size of the Communicator</td>
<td><code>MPI_Comm_size (comm,&amp;size)</code></td>
</tr>
<tr>
<td>Rank in the Communicator</td>
<td><code>MPI_Comm_rank (comm,&amp;rank)</code></td>
</tr>
<tr>
<td>Terminate all processes in a communicator</td>
<td><code>MPI_Abort (comm,errorcode)</code></td>
</tr>
<tr>
<td>Name of the current processor</td>
<td><code>MPI_Get_processor_name (&amp;name,&amp;resultlength)</code></td>
</tr>
<tr>
<td>Finalize</td>
<td><code>MPI_Finalize ()</code></td>
</tr>
<tr>
<td>Blocking sends</td>
<td><code>MPI_Send(buffer,count,type,dest,tag,comm)</code></td>
</tr>
<tr>
<td>Non-blocking sends</td>
<td><code>MPI_Isend(buffer,count,type,dest,tag,comm,request)</code></td>
</tr>
<tr>
<td>Blocking receive</td>
<td><code>MPI_Recv(buffer,count,type,source,tag,comm,status)</code></td>
</tr>
<tr>
<td>Non-blocking receive</td>
<td><code>MPI_Irecv(buffer,count,type,source,tag,comm,request)</code></td>
</tr>
<tr>
<td>Wait a request</td>
<td><code>MPI_Wait (&amp;request,&amp;status)</code></td>
</tr>
<tr>
<td>Barrier</td>
<td><code>MPI_Barrier (comm)</code></td>
</tr>
</tbody>
</table>
Using Sun Grid Engine

- Sun Grid Engine is the queuing system used on Mills cluster, let see a few command:
  - **qsub [options] script.qs**
    - -pe para_env nbr_slots
    - -l
      - exclusive=1
      - standby=1
  - **qconf [options]**
    - -sql : list of all queues
    - -sq name : detail of the queue
    - -spl : list of parallel environment
  - **qstat**
  - **qlogin**
3.4 (b) - MPI API

- MPI_CHAR
- MPI_WCHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_LONG_LONG_INT
- MPI_SIGNED_CHAR
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_LONG_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE
- MPI_C_BOOL
- ...

- MPI_Type_contiguous (count, oldtype, &newtype)
- MPI_Type_vector (count, blocklength, stride, oldtype, &newtype)
- MPI_Type_indexed (count, blocklens[], offsets[], old_type, &newtype)
- MPI_Type_commit (&datatype)
- MPI_Type_free (&datatype)

- MPI_Bcast (&buffer, count, datatype, root, comm)
- MPI_Scatter (&s_buf, s_cnt, s_type, &r_buf, r_cnt, r_type, root, comm)
- MPI_Gather (&s_buf, s_cnt, s_type, &r_buf, r_cnt, r_type, root, comm)
- MPI_Reduce (&s_buf, &r_buf, count, datatype, op, root, comm)
  - MPI_MAX
  - MPI_MIN
  - MPI_SUM
  - MPI_PROD
  - ...
- MPI_Scan (&s_buf, &r_buf, count, datatype, op, comm)
- MPI_Allgather (&s_buf, s_cnt, s_type, &r_buf, r_cnt, r_type, comm)
- MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
- MPI_Alltoall (&s_buf, s_cnt, s_type, &r_buf, r_cnt, r_type, comm)