HPC Workshop – University of Kentucky
May 9, 2007 – May 10, 2007

Part 3 – Parallel Programming
Parallel Programming

- **Concepts**
  - Amdahl’s Law
  - Parallel Programming Models

- **Tools**
  - Compiler (Intel)
  - Math Libraries (Intel)
  - Debuggers (Intel)
  - Profilers (Intel)

- **Standard Benchmarks**
Why do Parallel Programming

- Time to solution
- Resource restrictions
- Only option
- Interesting and Challenging
Amdahl’s Law

- Generalized Amdahl’s Law

\[
\text{SpeedUp} = \frac{1}{\sum_{k=0}^{n} \frac{P_k}{S_k}}
\]

where,
- \( P_k \) is the % of instructions that can be parallelized
- \( S_k \) is the Speed-up due to that improvement
- \( n \) is the number of Tasks

Or

\[
\text{SpeedUp} = \frac{1}{F + \frac{(1-F)}{N}}
\]

- \( F \) is the Fraction of the code that is sequential
- \( N \) is the Number of Processors
What do we learn from Amdahl’s Law

Even small amount of Sequential code hinders parallel efficiency

Efficiency = \frac{\text{SpeedUp}}{N}

- Set some reasonable goals – e.g. 80% parallel efficiency
- Starting from Sequential code may not always be a good idea – consider parallel algorithms and data structures
Parallel Programming Models

**Problem Phase**
- **Task**
  - Embarrassingly Parallel Tasks
  - Replicated Data
  - Divide and Conquer
- **Data**
  - Divide and Conquer
  - Temporal decomposition
  - Spatial/Geometrical decomposition
- **Data Flow**
  - Pipeline
  - Wavefront

**Physical Phase**
- Instruction parallel
- Multi-threading
- SIMD/Vector
- Shared Memory
  - Uniform Memory Access
  - (cc) Non-Uniform Memory Access
- Distributed Memory
  - Closely coupled
  - Loosely Coupled
- Distributed Shared Memory
- Grid

**Software Phase**
- `pThreads`
- `OpenMP`
- `MPI`
- `pvm`
- `Global Array`
- `Linda`
- `Etc.....`
pThreads (POSIX Threads)

- Thread is an independent stream of instructions
- Threads share a process space (process ID)
- Typically used in Shared Memory architectures

Suitable Tasks:
- Use many CPU Cycles
- Respond to asynchronous events
- Several Tasks that can be performed in parallel
pThread API

- **All pThread functions start with “pthread_”**
- **Three broad groups**
  - Thread Management – create, destroy, join, set, inquire attributes
  - Synchronization or Mutexes
  - Condition Variables
Thread Management: (some functions)
- `pthread_create(thread, attr, start_routine, arg)`
- `pthread_exit(status)`
- `pthread_attr_init(attr)`
- `pthread_attr_destroy(attr)`
- `pthread_join(threadid, status)`
- `pthread_detach(threadid, status)`

Mutex: (some functions)
- `pthread_mutex_init(mutex, attr)`
- `pthread_mutex_destroy(mutex)`
- `pthread_mutexattr_init(attr)`
- `pthread_mutexattr_destroy(attr)`
- `pthread_mutex_lock(mutex)`
- `pthread_mutex_unlock(mutex)`

Conditional Variables: (some functions)
- `pthread__cond_init(condition, attr)`
- `pthread__cond_destroy(condition)`
- `pthread__condattr_init(attr)`
- `pthread__condattr_destroy(attr)`
- `pthread__cond_wait(condition, mutex)`
- `pthread__cond_signal(condition)`
- `pthread__cond_broadcase(condition)`
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 8
void *PrintHello(void *threadid) {
    printf("Hello World from Threadid %d\n", threadid);
}
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int retval, tid;
    for(tid=0; tid < NUM_THREADS; tid++) {
        printf("Creating Thread %d\n", tid);
        retval = pthread_create(&threads(tid), NULL, PrintHello, (void *) tid);
        if (retval) {
            printf("Error creating threads %d\n", retval);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
OpenMP (http://www.openmp.org)

- Portable and scalable model for shared memory programming
- Lean and Mean
- API is standardized but not the implementation
- Explicit Parallelism (gives developers full control)
- Compiler directive based
- Nested Parallelism supported
- Uses Fork-Join model

No Parallel I/O
OpenMP Directives (basic)

Format

**Fortran: OpenMP directives**

- **!$OMP directive [clause]** - valid for both fixed and free form
- **C$OMP directive [clause]** - valid only for fixed form
- ***$OMP directive [clause]** - valid only for fixed form

**c/c++: OpenMP Pragma**

- `#pragma omp directive [clause] new-line`

**ENVIRONMENT VARIABLE:**

- `OMP_NUM_THREADS` to set the number of OpenMP Threads
  (bash: `export OMP_NUM_THREADS=16`)
- Or you can call the library function `omp_set_num_threads()` within your code
OpenMP Directives (contd.)
Parallel Directive

C/C++
#pragma omp parallel [clause],[clause]...... new-line
    {  
        structured block  
    }  

Fortran
!$omp parallel [clause],[clause]......
    structured block
!$omp end parallel

Clause
If (scalar-expression)
private (list)
firstprivat (list)
default (shared|none)
shared (list)
copyin (list)
reduction (operator:list)
um_threads (integer-expression)
Example 1: HelloWorld.f90

```fortran
PROGRAM HELLO
  INTEGER NTHERDS, TID, OMP_GET_NUM_THREADS,
       + OMP_GET_THREAD_NUM
  C Fork a team of threads
  !$OMP PARALLEL PRIVATE(TID)
  C Obtain and print Number of Threads
    TID = OMP_GET_THREAD_NUM()
    IF (TID .EQ. 0) THEN
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of Threads = ', NTHREADS
    END IF
    PRINT *, ' Hello World From Thread ', TID
  !$OMP END PARALLEL
END
```

`ifort -openmp –o helloworld HelloWorld.f90`
OpenMP – Parallel Do/For directive

**C/C++**

```c
#pragma omp parallel for [clause],[clause]... new-line
 for(int i=0; i < N; i++) {
   work;
 }
```

**Fortran**

```fortran
!$omp parallel do [clause],[clause]......
 DO I = 1, N
   work
 END DO
!$omp end do
```

**Clause**

- `private (list)`
- `firstprivate (list)`
- `lastprivate (list)`
- `reduction (operator:list)`
- `ordered`
- `sched (kind[,chunk size])`
- `nowait`

`kind` is static, dynamic, guided or runtime
Other important OpenMP Constructs and Concepts

- **workshare** – divide the execution block into separate units of work and do them in parallel
- **Master and Synchronization constructions** – master, barrier, critical, atomic, flush, ordered
- **Parallel Sections construct**
- **Nesting**
- **Dynamic**
- **Run time library calls**
MPI (http://www.mpi-forum.org)

- Message Passing Interface
- Specification (rather than a library)
- Practical, Portable, Efficient, Flexible
- Interface specification for both c/c++ and Fortran
- MPI-1 and MPI-2
- Suitable for implementing (almost all) Distributed Memory Model in both distributed and shared memory environments
- Explicit parallelism
- Number of Tasks static in MPI-1 and dynamic in MPI-2 (to some extent)
MPI – Getting Started

C
Header file:       #include “mpi.h”

Format of MPI Calls:
RetVal = MPI_Xxxxxx(parameter,….)
e.g.
RetVal = MPI_Send(msg,
                   strlen(msg)+1,MPI_CHAR,
                   Dest,tag,MPI_COMM_WORLD);

RetValue is MPI_SUCCESS if successful

Fortran
Header file:       include “mpif.h”

Format of MPI Calls:
CALL MPI_Xxxxxxx(parameter,…….)
e.g.
   CALL MPI_SEND(MSG, 12,
                  MPI_CHARACTER, DEST, TAG,
                  MPI_COMM_WORLD,IERR)

IERR is MPI_SUCCESS if successful

C++ bindings only in MPI-2
MPI Program Structure

- MPI Include File
- Initialize MPI environment
- Do Work and use MPI Calls to synch results
- Terminate MPI environment
MPI Communicators and Groups

MPI_COMM_WORLD

GROUP 1

GROUP 2

COMM1

COMM2
MPI Environment Management Routines (some)

- **MPI_INIT**
  
  ```
  CALL MPI_INIT(IERR)
  MPI_Init(&argc,&argv)
  ```

- **MPI_FINALIZE**
  
  ```
  CALL MPI_FINALIZE(IERR)
  MPI_Finalize()
  ```

- **MPI_COMM_SIZE**
  
  ```
  CALL MPI_COMM_SIZE(COMM,SIZE,IERR)
  MPI_Comm_size(comm,&size)
  ```

- **MPI_COMM_RANK**
  
  ```
  CALL MPI_COMM_RANK(COMM,RANK,IERR)
  MPI_Comm_rank(comm,&rank)
  ```
MPI Point-To-Point Communications

- Only between 2 MPI Ranks
- Send and Receive Routines
  - Synchronous Send
  - Blocking Send / Receive
  - Non-Blocking Send / Receive
  - Buffered Send
  - Combined Send/Receive
  - “Ready” Send
- Any Send can be paired with any Receive
- Other MPI Routines to wait for message or Test status
Blocking vs Non-Blocking

**Blocking**
- Send will return only after it is safe to modify application buffer
- Can be synchronous (handshaking)
- Can be asynchronous (system buffer holds the data)
- Receive will only return after it is safe for using the data

**Non-Blocking**
- Send and Receive will return immediately
- It is a simple “request” to the MPI Library to perform the operation
- Unsafe to modify application buffer before testing the status of the message
- Performance gains (potential) by overlapping communications with computations
Some Point-to-Point Send / Receive Routines

Blocking Send

MPI_Send(buffer,count,type,dest,tag,comm)
CALL MPI_SEND(buffer,count,type,dest,tag,comm,ierr)

Blocking Receive:

MPI_Recv(buffer,count,type,source,tag,comm,status)
CALL MPI_RECV(buffer,count,type,source,tag,comm,status,ierr)

Non-Blocking Send:

MPI_Isend(buffer,count,type,dest,tag,comm,request)
CALL MPI_ISEND(buffer,count,type,dest,tag,comm,request,ierr)

Non-Blocking Receive:

MPI_Irecv(buffer,count,type,source,tag,comm,request)
CALL MPI_IRECV(buffer,count,type,source,tag,comm,request,ierr)
### MPI Data Types (some)

<table>
<thead>
<tr>
<th>C Data Types</th>
<th>Fortran Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>MPI_CHARACTER</td>
</tr>
<tr>
<td>signed char</td>
<td>character(1)</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>signed short int</td>
<td>integer</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>signed int</td>
<td></td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>MPI_INTEGER</td>
</tr>
<tr>
<td>signed long int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td></td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td></td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td></td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>MPI_REAL</td>
</tr>
<tr>
<td>float</td>
<td>real</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>MPI_DOUBLE_PRECISION</td>
</tr>
<tr>
<td>double</td>
<td>double precision</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>long double</td>
<td></td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>MPI_BYTE</td>
</tr>
<tr>
<td>8 binary digits</td>
<td>8 binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>MPI_PACKED</td>
</tr>
<tr>
<td>data packed or unpacked with MPI_Pack 0/ MPI_Unpack</td>
<td>data packed or unpacked with MPI_Pack 0/ MPI_Unpack</td>
</tr>
</tbody>
</table>
MPI – Collective communications (some)

- All or None
- Synchronization
  - MPI_Barrier
- Data Movement
  - MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Alltoall
- Collective Computations
  - MPI_Reduce, MPI_Scan, MPI_Reduce_scatter
- Collective operations are Blocking
- No derived data types
MPI-2 Key New Functionalities

- **Dynamic Process**
- **One-Sided Communications**
  - Provides routines for one directional communications
  - Include Shared memory operations (put/get) and remote accumulate
- **Extended Collective Operations**
  - Allows non-blocking collective operations
- **External Interfaces to debuggers and profilers**
- **Additional Language Bindings – C++, F90**
- **Parallel I/O**
(Andrew) Parallel Compilers, Math Libs, etc