Reading for This Lecture

• Roosta, Chapter 4, Sections 1 and 3, Chapter 5
• MPI Introduction and Specification
• OpenMP Introduction, Specification, and Tutorial
Design Issues

• Platform/Architecture
• Task Decomposition
• Task Mapping/Scheduling
• Communication Protocol
Parallelizing Sequential Algorithms

• The most obvious approach to developing a parallel algorithm is to parallelize a sequential algorithm.

• The primary additional concept one must keep in mind is data access patterns.
  – In the case of shared memory architectures, one must be cognizant of possible collisions in accessing the main memory.
  – In the case of distributed memory architectures, one must be cognizant of the need to move data to where it is needed.

• In either case, losses in efficiency result from either idle time or wasted computation due to lack of availability of data locally.
Platforms

• High Performance Parallel Computers
  – Massively parallel
  – Distributed

• "Off the shelf" Parallel Computers
  – Small shared memory computers
  – Multi-core computers
  – GPUs
  – Clusters (of multi-core computers)
Task Decomposition

• Fine-grained parallelism
  – Suited for massively parallel systems with many small processors and fast communication links.
  – These are the algorithms we’ve primarily talked about so far.

• Course-grained parallelism
  – Suited to small numbers of more powerful processors.
  – Data decomposition
    * Recursion/Divide and Conquer
    * Domain Decomposition
  – Functional parallelism
    * Data Dependency Analysis
    * Pipelining
Task Agglomeration

- Depending on the number of processors available, we may have to run multiple tasks on a single processor.

- To do this effectively, we have to determine which tasks should be combined to achieve maximum efficiency.

- This requires the same analysis of communication patterns and data access done in task decomposition.
Mapping

- Concurrency
  - Data dependency analysis

- Locality
  - Interconnection network
  - Communication pattern

- Mapping is an optimization problem.

- These are very difficult to solve in general.
Communication Protocols: Message Passing

- Used primarily in distributed-memory or "hybrid" environments.
- Data is passed through explicit send and receive function calls.
- There is no explicit synchronization.
- In general, this is the most flexible and portable protocol.
- MPI is the established standard.
- PVM is a similar older standard that is still used.
Comunication Protocols: Open MP/Threads

- Used in shared-memory environments.
- Parallelism through "threading".
- Threads communicate through global memory.
- Can have explicit synchronization.
- **OpenMP** is a standard implemented by most compilers.
MPI Basics

• MPI stands for *Message Passing Interface*.

• It is an API for point-to-point communication that hides the platform-dependent details from the user.

• Each platform has its own implementation of MPI.

• The user launches the MPI processes in a distributed fashion and forms one or more “communicators.”

• Data can be sent explicitly between processes using message-passing calls.

• Allows for extremely portable parallel
Messaging Concepts

- Buffer
- Source
- Destination
- Tag
- Communicator
Types of Communication Calls

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Buffered send
- Combined send/receive
- ”Ready” send
## Basic Functions in MPI

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int MPI_Init(int *argc, char ***argv)</code></td>
<td>Join MPI</td>
</tr>
<tr>
<td><code>int MPI_Comm_rank (MPI_Comm comm, int *rank)</code></td>
<td>This process’s position within the communicator</td>
</tr>
<tr>
<td><code>int MPI_Comm_size (MPI_Comm comm, int *size)</code></td>
<td>Total number of processes in the communicator</td>
</tr>
<tr>
<td><code>int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )</code></td>
<td>Send a message to process with rank <code>dest</code> using tag</td>
</tr>
<tr>
<td><code>int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )</code></td>
<td>Receive a message with the specified tag from the process with the rank <code>source</code></td>
</tr>
<tr>
<td><code>int MPI_Finalize()</code></td>
<td>Resign from MPI</td>
</tr>
</tbody>
</table>
```c
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
```
Collective Communication

- **Synchronization**: processes wait until all members of the group have reached the synchronization point.

- **Data Movement**: broadcast, scatter/gather, all to all.

- **Collective Computation (reductions)**: one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.).
Virtual Topologies

- Allows the user to specify the topology of the interconnection network.
- This may allow certain features to be implemented more efficiently.
OpenMP/Threads

Single Process

Global Memory

Thread 1
Thread 2
Thread 3
OpenMP Implementation

- OpenMP is implemented through compiler directives.
- User is responsible for indicating what code segments should be performed in parallel.
- The user is also responsible for eliminating potential memory conflicts, etc.
- The compiler is responsible for inserting platform-specific function calls, etc.
OpenMP Features

• Capabilities are dependent on the compiler.
  – Primarily used on shared-memory architectures
  – Can work in distributed-memory environments (TreadMarks)

• Explicit synchronization

• Locking functions

• Critical regions

• Private and shared variables
Using OpenMP

• Compiler directives
  – parallel
  – parallel for
  – parallel sections
  – barrier
  – private
  – critical

• Shared library functions
  – omp_get_num_threads()
  – omp_set_lock()
OpenMP Example

```c
#pragma omp parallel for default(none) private(i,j,sum) \
for (i=0; i<m; i++){
    sum = 0.0;
    for (j=0; j<n; j++)
        sum += b[i][j]*c[j];
    a[i] = sum;
}
```
Figure 1: OpenMP Performance
OpenMP Concepts and Issues

• Race Conditions
  – Conflicts between processes in updating data.

• Deadlocks

• Critical regions

• Locking functions