Parallel Programming Paradigms

- Functional languages
- Parallelizing compilers
- Object parallel
- Data parallel
- Shared memory
- Partitioned global address space
- Remote memory access
- Message passing

Functional Languages

- Often implemented using dataflow, in which operations fire whenever their inputs are available, and results then become available as inputs for other operations
- Tend to require substantial extra overhead in work and storage, so have proven difficult to implement efficiently
- Have not been used widely in practice, though numerous experimental functional languages and dataflow systems have been developed

Parallelizing Compilers

- Automatically parallelize programs written in conventional sequential programming languages
- Difficult to do for arbitrary serial code
- Compiler can analyze serial loops for potential parallel execution, based on careful dependence analysis of variables occurring in loop
- User may provide hints (directives) to help compiler determine when loops can be parallelized and how
- OpenMP is standard for compiler directives

Object Parallel

- Parallelism encapsulated within distributed objects that bind together data and functions operating on data
- Parallel programs built by composing component objects that communicate via well-defined interfaces and protocols
- Implemented using object-oriented programming languages such as C++ or Java
- Often based on standards and supporting environments such as CORBA, DCE, CCA
- Examples include Charm++ and Legion
Data Parallel

- Simultaneous operations on elements of data arrays, typified by vector addition
- Low-level programming languages, such as Fortran 77 and C, express array operations element by element in some specified serial order
- Array-based languages, such as APL, Fortran 90, and MATLAB, treat arrays as higher-level objects and thus facilitate full exploitation of array parallelism

Shared Memory

- Classic shared-memory paradigm, originally developed for multitasking operating systems, focuses on control parallelism rather than data parallelism
- Multiple processes share common address space accessible to all, though not necessarily with uniform access time
- Because shared data can be changed by more than one process, access must be protected from corruption, typically by some mechanism to enforce mutual exclusion
- Shared memory supports common pool of tasks from which processes obtain new work as they complete previous tasks

Remote Memory Access

- One-sided, put and get communication to store data in or fetch data from memory of another process
- Does not require explicit cooperation between processes
- Must be used carefully to avoid corruption of shared data
- Included in MPI-2, to be discussed later

Data parallel languages provide facilities for expressing array operations for parallel execution, and some allow user to specify data decomposition and mapping to processors
- High Performance Fortran (HPF) is most visible attempt to standardize data parallel approach to programming
- Though naturally associated with SIMD architectures, data parallel languages have also been implemented successfully on general MIMD architectures
- Data parallel approach can be effective for highly regular problems, but tends to be too inflexible to be effective for irregular or dynamically changing problems

Lightweight Threads

- Most popular modern implementation of explicit shared-memory programming, typified by pthreads (POSIX threads)
- Reduce overhead for context-switching by providing multiple program counters and execution stacks so that extensive program state information need not be saved and restored when switching control quickly among threads
- Provide detailed, low-level control of shared-memory systems, but tend to be tedious and error prone
- More suitable for implementing underlying systems software (such as OpenMP and run-time support for parallelizing compilers) than for user-level applications

Partitioned Global Address Space

- Partitioned global address space (PGAS) model provides global memory address space that is partitioned across processes, with a portion local to each process
- Enables programming semantics of shared memory while also enabling locality of memory reference that maps well to distributed memory hardware
- Example PGAS programming languages include Chapel, Co-Array Fortran, Titanium, UPC, X-10

Message Passing

- Two-sided, send and receive communication between processes
- Most natural and efficient paradigm for distributed-memory systems
- Can also be implemented efficiently in shared-memory or almost any other parallel architecture, so it is most portable paradigm for parallel programming
- “Assembly language of parallel computing” because of its universality and detailed, low-level control of parallelism
- Fits well with our design philosophy and offers great flexibility in exploiting data locality, tolerating latency, and other performance enhancement techniques
Message Passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Facilitates debugging because accidental overwriting of memory is less likely and much easier to detect than with shared-memory
- Sometimes deemed tedious and low-level, but thinking about locality tends to result in programs with good performance, scalability, and portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

MPI was developed in three major stages, MPI-1 (1994), MPI-2 (1997) and MPI-3 (2012)

Features of MPI-1 include:
- point-to-point communication
- collective communication
- process groups and communication domains
- virtual process topologies
- environmental management and inquiry
- profiling interface
- bindings for Fortran and C

MPI Basics

- Closely matches computational model underlying our design methodology for developing parallel algorithms and provides natural framework for implementing them
- Although motivated by distributed-memory systems, works effectively on almost any type of parallel system
- Often outperforms other paradigms because it enables and encourages attention to data locality

MPI Basics

- Includes more than 125 functions, with many different options and protocols
- Small subset suffices for most practical purposes
- We will cover just enough to implement algorithms we will consider
- In some cases, performance can be enhanced by using features that we will not cover in detail

MPI Basics

- Additional features of MPI-2 include:
  - dynamic process management
  - input/output
  - one-sided operations for remote memory access
  - bindings for C++
- Includes communication among multiple concurrent processes
- Includes several varieties of point-to-point communication, as well as collective communication among groups of processes
- Implemented as library of routines callable from conventional programming languages such as Fortran, C, and C++
- Has been universally adopted by developers and users of parallel systems that rely on message passing

MPI Basics

- Additional features of MPI-3 include:
  - nonblocking collectives
  - new one-sided communication operations
  - Fortran 2008 bindings

MPI Basics

Building and Running MPI Programs

- Executable module must first be built by compiling user program and linking with MPI library
- One or more header files, such as mpi.h, may be required to provide necessary definitions and declarations
- MPI is generally used in SPMD mode, so only one executable must be built, multiple instances of which are executed concurrently
- Most implementations provide command, typically named mpiexec, for spawning MPI processes
  - MPI-2 specifies mpiexec for portability
- User selects number of processes and on which processors they will run

MPI Basics
Custom versions of MPI supplied by vendors of almost all current parallel computer systems

Freeware versions available for clusters and similar environments include

- MPICH: http://www.mpich.org/
- OpenMPI: http://www.open-mpi.org

Both websites also provide tutorials on learning and using MPI

MPI standard (MPI-1, -2, -3) available from MPI Forum
http://www.mpi-forum.org

Specifying Messages

Information necessary to specify message and identify its source or destination in MPI include

- msg: location in memory where message data begins
- count: number of data items contained in message
- datatype: type of data in message
- source or dest: rank of sending or receiving process in communicator
- tag: identifier for specific message or kind of message
- comm: communicator

Available MPI data types include

- C: char, int, float, double
- Fortran: integer, real, double precision, complex

Use of MPI data types facilitates heterogeneous environments in which native data types may vary from machine to machine

Also supports user-defined data types for contiguous or noncontiguous data

Example: MPI Program for 1-D Laplace Example

```c
#include <mpi.h>

int main(int argc, char **argv) {int k, p, me, left, right, count = 1, tag = 1, nit = 10;float ul, ur, u = ... tag, MPI_COMM_WORLD, &status);if (me > 0) MPI_Recv(&ul, count, MPI_FLOAT,left, tag, MPI_COMM_WORLD, &status);}
```

On return,
```c
int MPI_Init(int *argc, char ***argv);
```
Initiates use of MPI

```c
int MPI_Finalize(void);
```
Concludes use of MPI

```c
int MPI_Comm_size(MPI_Comm comm, int *size);
```
On return, size contains number of processes in communicator

A few special MPI routines may be called before MPI_Init or after MPI_Finalize

Example: MPI Program for 1-D Laplace Example

```c
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```
On return, rank contains rank of calling process in communicator, with $0 \leq rank \leq size-1

```c
int MPI_Send(void *msg, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

On return, msg can be reused immediately

```c
int MPI_Recv(void *msg, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```
On return, msg contains requested message
Standard Send and Receive Functions

- Standard send and receive functions are **blocking**, meaning they do not return until resources specified in argument list can safely be reused.
- In particular, **MPI_Recv** returns only after receive buffer contains requested message.
- **MPI_Send** may be initiated before or after matching **MPI_Recv** initiated.
- Depending on specific implementation of MPI, **MPI_Send** may return before or after matching **MPI_Recv** initiated.

Other MPI Functions

- MPI functions covered thus far suffice to implement almost any parallel algorithm with reasonable efficiency.
- Dozens of other MPI functions provide additional convenience, flexibility, robustness, modularity, and potentially improved performance.
- But they also introduce substantial complexity that may be difficult to manage.
- For example, some facilitate overlapping of communication and computation, but place burden of synchronization on user.

Communication Functions for Various Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Blocking</th>
<th>Nonblocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard buffered</td>
<td><strong>MPI_Send</strong></td>
<td><strong>MPI_Isend</strong></td>
</tr>
<tr>
<td>synchronous ready</td>
<td><strong>MPI_Ssend</strong></td>
<td><strong>MPI_Isend</strong></td>
</tr>
</tbody>
</table>

- **MPI_Recv** and **MPI_Irecv** are blocking and nonblocking functions for receiving messages, regardless of mode.
- **MPI_Buffer_attach** is used to provide buffer space for buffered mode.

Standard Mode

- Standard mode does not specify whether messages are buffered.
- Buffering allows more flexible programming, but requires additional time and memory for copying messages to and from buffers.
- Given implementation of MPI may or may not use buffering for standard mode, or may use buffering only for messages within certain range of sizes.
- To avoid potential deadlock when using standard mode, portability demands conservative assumptions concerning order in which sends and receives are initiated.
- User can exert explicit control by using buffered or synchronous mode.

For same `source`, `tag`, and `comm`, messages are received in order in which they were sent.
- Wild card values `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be used for source and tag, respectively, in receiving message.
- Actual source and tag can be determined from `MPI_SOURCE` and `MPI_TAG` fields of `status` structure (entries of `status` array in Fortran, indexed by parameters of same names) returned by **MPI_Recv**.

Communication Modes

- Nonblocking functions include request argument used subsequently to determine whether requested operation has completed.
- Nonblocking is different from **asynchronous**.
- **MPI_Wait** and **MPI_Test** for completion of nonblocking communication.
- **MPI_Probe** and **MPI_Iprobe** for incoming message without actually receiving it.
- Information about message determined by probing can be used to decide how to receive it.
- **MPI_Cancel** cancels outstanding message request, useful for cleanup at end of program or after major phase of computation.

Persistent Communication

- Communication operations that are executed repeatedly with same argument list can be streamlined.
- **Persistent** communication binds argument list to request, and then request can be used repeatedly to initiate and complete message transmissions without repeating argument list each time.
- Once argument list has been bound using **MPI_Send_init** or **MPI_Recv_init** (or similarly for other modes), then request can subsequently be initiated repeatedly using **MPI_Start**.
Collective Communication

- MPI_Bcast
- MPI_Reduce
- MPI_Allreduce
- MPI_Alltoall
- MPI_Allgather
- MPI_Scatter
- MPI_Gather
- MPI_Scan
- MPI_Barrier

Virtual Process Topologies

- MPI provides virtual process topologies corresponding to regular Cartesian grids or more general graphs
- Topology is optional additional attribute that can be given to communicator
- Virtual process topology can facilitate some applications, such as 2-D grid topology for matrices or 2-D or 3-D grid topology for discretized PDEs
- Virtual process topology may also help MPI achieve more efficient mapping of processes to given physical network

MPI Profiling Interface

- MPI provides profiling interface via name shift for each function
- For each MPI function with prefix MPI_, alternative function with prefix PMPI_ provides identical functionality
- Profiling library can intercept calls to each standard MPI function name, record appropriate data, then call equivalent alternate function to perform requested action
- MPI_Pcontrol provides profiling control if profiling is in use, but otherwise does nothing, so same code can run with or without profiling, depending on whether executable module is linked with profiling or nonprofiling MPI libraries
- Data collected by profiling can be used for performance analysis or visualization

Virtual Process Topologies

- MPI_Topo_test determines what type of topology, if any, has been defined for given communicator
- Inquiry functions provide information about existing graph topology, such as number of nodes and edges, lists of degrees and edges, number of neighbors of given node, or list of neighbors of given node
- Inquiry functions obtain information about existing Cartesian topology, such as number of dimensions, number of processes for each dimension, and whether each dimension is periodic
- Functions also available to obtain coordinates of given process, or rank of process with given coordinates

Manipulating Communicators

- MPI_Comm_create
- MPI_Comm_dup
- MPI_Comm_split
- MPI_Comm_split
- MPI_Comm_free

Virtual Process Topologies

- MPI_Dist_graph_create creates general graph topology, based on number of nodes, node degrees, and graph edges specified by user
- MPI_Cart_create creates Cartesian topology based on number of dimensions, number of processes in each dimension, and whether each dimension is periodic (i.e., wraps around), as specified by user
- Hypercubes are included, since \( k \)-dimensional hypercube is simply \( k \)-dimensional torus with two processes per dimension
- MPI_Cart_shift provides shift of given displacement along any given dimension of Cartesian topology

Timing MPI Programs

- MPI_Wtime returns elapsed wall-clock time in seconds since some arbitrary point in past
- Elapsed time for program segment given by difference between MPI_Wtime values at beginning and end
- Process clocks are not necessarily synchronized, so clock values are not necessarily comparable across processes, and care must be taken in determining overall running time for parallel program (see MPI_WTIME_IS_GLOBAL)
- MPI_Wtick returns resolution of MPI_Wtime clock

MPICL Tracing Facility

- MPICL is instrumentation package for MPI based on PICL tracing facility and MPI profiling interface
  http://www.csm.ornl.gov/picl/
- In addition to tracing MPI events, MPICL also provides tracing of user-defined events
- Resulting trace file contains one event record per line, with each event record containing numerical data specifying event type, timestamp, process rank, message length, etc
- Trace data output format is suitable for input to ParaGraph visualization tool
Visualizing Trace Data

ParaGraph is graphical display tool for visualizing behavior and performance of parallel programs that use MPI.

ParaGraph provides graphical answers to questions such as:
- Is particular process busy or idle at any given time?
- Which processes are communicating with each other?
- What part of program is executing at any given time?

Performance data can be viewed from many different visual perspectives to gain insights that might be missed by any single view.

ParaGraph provides about thirty different visual displays, most of which are one of three basic types:
- processor utilization
- interprocessor communication
- user-defined tasks

Most displays change dynamically to provide graphical animation of program behavior.

ParaGraph is extensible so users can add new displays, specific to given application.

Example: MPICL Tracing

```c
#include <mpi.h>
void main(int argc, char **argv) {
  int k, myid, nprocs, ntasks, work;
  MPI_Init(&argc, ... task k, including MPI calls}
  work = work done in task k;
  traceevent("exit", k, i, work);
  MPI_Finalize();
}
```

Example: MPICL Tracing Using MPI Pcontrol

```c
#include <mpi.h>
#include <pcontrol.h>
void main(int argc, char **argv) {
  int k, myid, nprocs, ntasks, work;
  MPI_Init(&argc, ... task k, including MPI calls}
  work = work done in task k;
  traceevent("exit", k, i, work);
  MPI_Finalize();
}
```

MPI Performance Analysis Tools

- Jumpshot and SLOG
- Intel Trace Analyzer (formerly Vampir)
  [http://www.hiperism.com/PALVAMP.htm](http://www.hiperism.com/PALVAMP.htm)
- IPM: Integrated Performance Monitoring
- mpiP: Lightweight, Scalable MPI Profiling
- TAU: Tuning and Analysis Utilities
  [http://www.cs.uoregon.edu/research/tau/home.php](http://www.cs.uoregon.edu/research/tau/home.php)
OpenMP

- Shared memory model, SPMD
- Extends C and Fortran with directives (annotations) and functions
- Relies on programmer to provide information that may be difficult for compiler to determine
- No concurrency except when directed; typically, most lines of code run on single processor/core
- Parallel loops described with directives
  
  ```
  #pragma omp parallel for reduction(+:sum) private(i)
  ```

OpenMP — Portable Shared Memory Programming

Example:

```!
$OMP PARALLEL DO DEFAULT(NONE) SHARED() PRIVATE()
```

```!
for (...) {
  ...

$OMP END PARALLEL
```!

Example: OpenMP Program for 1-D Laplace Example

```!
#include <omp.h>
int main(int argc, char **argv) {
int k, i, nit=10;
float alpha = 1.0, beta = 2.0;
float u0[MAX_U], u1[MAX_U];
float * restrict u0p, * restrict u1p, * restrict tmp;

for (k=0; k<nit; k++) {
  #pragma omp parallel for default(none) shared(u0p) private(i)
  for (i=1; i < MAX_U-1; i++) {u1p[i] = (u0p[i-1]+u0p[i+1])/2.0;}
  tmp = u1p; u1p = u0p; u0p = tmp; /* swap pointers*/
}
```!


Michael T. Heath

Parallel Programming Paradigms

More OpenMP

- `omp_get_thread_num()` — returns number of active threads within parallel region
- `omp_get_num_procs()` — returns number of available cores

General parallel blocks of code (executed by all available threads) described as

```!
#pragma omp parallel
{
...
$OMP END PARALLEL
```!

References — MPI

- P. S. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1997

References — Performance Visualization

- G. Tomas and C. W. Ueberhuber, Visualization of Scientific Parallel Programs, LNCS 771, Springer, 1994

References — Other Parallel Systems

- J. Kepner, Parallel MATLAB for Multicore and Multinode Computers, SIAM, Philadelphia, 2009

References — General

- C. Lin and L. Snyder, Principles of Parallel Programming, Addison-Wesley, 2008
- P. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2011
- M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2003

References — Performance Visualization

- G. Tomas and C. W. Ueberhuber, Visualization of Scientific Parallel Programs, LNCS 771, Springer, 1994

References — General

- C. Lin and L. Snyder, Principles of Parallel Programming, Addison-Wesley, 2008
- P. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann, 2011
- M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2003