Parallel Numerical Algorithms

Chapter 2 – Parallel Thinking
Section 2.2 – Parallel Programming

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

Parallel Programming Paradigms

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

Examples of Parallel Programming Languages and Libraries

- *Data parallel*: Fortran, HPF, Global Arrays
- *Message passing*: MPI
- *Shared-memory*: pthreads, OpenMP, CUDA, OpenCL
- *Object parallel*: Charm++, Legion, HPX
- *Libraries*: BLAS, ScaLAPACK, PETSc, Cyclops

Functional Languages

- Express *what* to compute (i.e., mathematical relationships to be satisfied), but not *how* to compute it or order in which computations are to be performed
- Avoid artificial serialization imposed by imperative programming languages
- Avoid storage references, side effects, and aliasing that make parallelization difficult
- Permit full exploitation of any parallelism inherent in computation

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

Parallelizing Compilers

- Automatic or semi-automatic, loop-based approach has been most successful in exploiting modest levels of concurrency on shared-memory systems
- Many challenges remain before effective automatic parallelization of arbitrary serial code can be routinely realized in practice, especially for massively parallel, distributed-memory systems
- Parallelizing compilers can produce efficient “node code” for hybrid architectures with SMP nodes, thereby freeing programmer to focus on exploiting parallelism across nodes

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

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

Data Parallel

- Data parallel languages provide facilities for expressing array operations for parallel execution, and some allow user to specify data decomposition and mapping to processors
- 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

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

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

Shared Memory

- Most naturally and efficiently implemented on true shared-memory architectures, such as SMPs
- Can also be implemented with reasonable efficiency on NUMA (nonuniform memory access) shared-memory or even distributed-memory architectures, given sufficient hardware or software support
- With nonuniform access or distributed shared memory, efficiency usually depends critically on maintaining locality in referencing data, so design methodology and programming style often closely resemble techniques for exploiting locality in distributed-memory systems

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 — Message-Passing Interface

- Provides 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 — Message-Passing Interface

- 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
- Is performance-efficient because it enables and encourages attention to data locality

MPI-1

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-2

Additional features of MPI-2 include:

- dynamic process management
- input/output
- one-sided operations for remote memory access
- bindings for C++

Additional features of MPI-3 include:

- nonblocking collectives
- new one-sided communication operations
- Fortran 2008 bindings

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 `mpirun`, for spawning MPI processes
    - MPI-2 specifies `mpiexec` for portability
- User selects number of processes and on which processors they will run

Availability of MPI

- Custom versions of MPI supplied by vendors of almost all current parallel computers systems
- Freeware versions available for clusters and similar environments include
  - MPICH: `http://www.mpich.org/`
  - OpenMPI: `http://www.open-mpi.org`
- Both websites provide tutorials on learning and using MPI
- MPI standard (MPI-1, -2, -3) available from MPI Forum `http://www.mpi-forum.org`

Communicator (Groups)

- A *communicator* defines a group of MPI processes
- Each process is identified by its *rank* within given group
- Rank is integer from zero to one less than size of group (`MPI_PROC_NULL` is rank of no process)
- Initially, all processes belong to `MPI_COMM_WORLD`
- Additional communicators can be created by user via `MPI_Comm_split`
- Communicators simplify point-to-point communication on virtual topologies and enable collectives over any subset of processors

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

MPI Data Types

- Available C MPI data types include
  - `char`, `int`, `float`, `double`
- 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

Minimal MPI

Minimal set of six MPI functions we will need

- `int MPI_Init(int *argc, char ***argv);`

  Initiates use of MPI

- `int MPI_Finalize(void);`

  Concludes use of MPI

- `int MPI_Comm_size(MPI_Comm comm, int *size);`

  On return, size contains number of processes in communicator comm

Minimal MPI

-
  ```
  int MPI_Comm_rank(MPI_Comm comm, int *rank);
  ```

  On return, rank contains rank of calling process in communicator comm, with $0 \leq$ rank $\leq$ size-1

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

  On return, msg can be reused immediately

-
  ```
  int MPI_Recv(void *msg, int count, MPI_Datatype datatype,
       int source, int tag, MPI_Comm comm, MPI_Status *stat);
  ```

  On return, msg contains requested message

Example: MPI Program for 1-D Laplace Example

```
#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 = 1.0, alpha = 1.0, beta = 2.0;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  left = me-1; right = me+1;
  if (me == 0) ul = alpha; if (me == p-1) ur = beta;
  for (k = 1; k <= nit; k++) {
    if (me % 2 == 0) {
      if (me > 0) MPI_Send(&u, count, MPI_FLOAT,
         left, tag, MPI_COMM_WORLD);
      if (me < p-1) MPI_Send(&u, count, MPI_FLOAT,
         right, tag, MPI_COMM_WORLD);
      if (me < p-1) MPI_Recv(&ur, count, MPI_FLOAT,
         right, tag, MPI_COMM_WORLD, &status);
      if (me > 0) MPI_Recv(&ul, count, MPI_FLOAT,
         left, tag, MPI_COMM_WORLD, &status);
```

Example: MPI Program for 1-D Laplace Example

```
        else {
          if (me < p-1) MPI_Recv(&ur, count, MPI_FLOAT,
             right, tag, MPI_COMM_WORLD, &status);
          MPI_Recv(&ul, count, MPI_FLOAT,
             left, tag, MPI_COMM_WORLD, &status);
          MPI_Send(&u, count, MPI_FLOAT,
             left, tag, MPI_COMM_WORLD);
          if (me < p-1) MPI_Send(&u, count, MPI_FLOAT,
             right, tag, MPI_COMM_WORLD);
        }
        u = (ul+ur)/2.0;
      }
      MPI_Finalize();
    }
```

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

Standard Send and Receive Functions

- 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`

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 Modes

- Nonblocking functions include **`request`** argument used subsequently to determine whether requested operation has completed (different from *asynchronous*)
- **`MPI_Isend`** and **`MPI_Irecv`** are nonblocking
- **`MPI_Wait`** and **`MPI_Test`** wait or test for completion of nonblocking communication
- **`MPI_Probe`** and **`MPI_Iprobe`** probe for incoming message without actually receiving it
- Information about message determined by probing can be used to decide how to receive it 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`

Manipulating Communicators

- `MPI_Comm_create`
- `MPI_Comm_dup`
- `MPI_Comm_split`
- `MPI_Comm_compare`
- `MPI_Comm_free`

MPI Performance Analysis Tools

- Jumpshot and SLOG `http://www.mcs.anl.gov/perfvis/`
- Intel Trace Analyzer (formerly Vampir)
  `http://www.hiperism.com/PALVAMP.htm`
- IPM: Integrated Performance Monitoring
  `http://ipm-hpc.sourceforge.net/`
- mpiP: Lightweight, Scalable MPI Profiling
  `http://mpip.sourceforge.net/`
- TAU: Tuning and Analysis Utilities
  `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 default(none) shared() private()
for (...) {
}
```

More OpenMP

- `omp_get_num_threads()` – returns number of active threads within parallel region
- `omp_get_thread_num()` – returns index of thread within parallel region

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

```
#pragma omp parallel
{
}
```

Race Conditions

Example:

```
sum = 0.0;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { sum += u[i]; }
```

Race condition: result of updates to `sum` depend on which thread wins race in performing store to memory

OpenMP provides `reduction` clause for this case:

```
sum = 0.0;
#pragma omp parallel for reduction(+:sum) private(i)
for (i=0; i<n; i++) { sum += u[i]; }
```

Not hypothetical example: on one dual-processor system, first loop computes wrong result roughly half of time

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=u0, * restrict u1p=u1, *tmp;

  u0[0] = u1[0] = alpha;
  u0[MAX_U-1] = u1[MAX_U-1] = beta;
  for (k=0; k<nit; k++) {
    #pragma omp parallel for default(none) shared(u1p,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 */
  }
}
```

References – General

- A. H. Karp, Programming for parallelism, *IEEE Computer* 20(9):43-57, 1987
- B. P. Lester, *The Art of Parallel Programming*, 2nd ed., 1st World Publishing, 2006
- 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
- B. Wilkinson and M. Allen, *Parallel Programming*, 2nd ed., Prentice Hall, 2004

References – MPI

- W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press, 2000
- P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997
- M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference, Vol. 1, The MPI Core*, 2nd ed., MIT Press, 1998
- W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference, Vol. 2, The MPI Extensions*, MIT Press, 1998
- MPI Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*, `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`

References – Other Parallel Systems

- B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP*, MIT Press, 2008
- D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010
- J. Kepner, *Parallel MATLAB for Multicore and Multinode Camputers*, SIAM, Philadelphia, 2009
- P. Luszczek, Parallel programming in MATLAB, *Internat. J. High Perf. Comput. Appl.*, 23:277-283, 2009

References – Performance Visualization

- T. L. Casavant, ed., Special issue on parallel performance visualization, *J. Parallel Distrib. Comput.* 18(2), June 1993
- M. T. Heath and J. A. Etheridge, Visualizing performance of parallel programs, *IEEE Software* 8(5):29-39, 1991
- M. T. Heath, Recent developments and case studies in performance visualization using ParaGraph, G. Haring and G. Kotsis, eds., *Performance Measurement and Visualization of Parallel Systems*, pp. 175-200, Elsevier Science Publishers, 1993
- G. Tomas and C. W. Ueberhuber, *Visualization of Scientific Parallel Programs*, LNCS 771, Springer, 1994
- O. Zaki, E. Lusk, W. Gropp and D. Swider, Toward Scalable Performance Visualization with Jumpshot, *Internat. J. High Perf. Comput. Appl.*, 13:277-288, 1999