CISC 372: Parallel Computing

Introduction to MPI

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

siegel@udel.edu

Message Passing Interface: Brief History

- late 1980s
  - every vendor had their own message-passing library
- April 1992
  - workshop led to working group on a message-passing standard
  - involved academia, industry (vendors), users
  - rather than choose one of the existing libraries, "big tent"
- 1994: MPI: A Message Passing Interface Standard (v1.0)
- defines an interface
  - types
  - constants
  - functions
- versions 1.1, 1.2, 1.3, 2.0, 2.1, 2.2, 3.0, 3.1
- MPI 3.1 approved on June 4, 2015
  - `http://www.mpi-forum.org`
  - 868 pages

MPI Program Model

- an MPI program consists of multiple processes
- each process has its own memory (no shared memory)
- think of each process as a program running on its own computer
- the computers can have different architectures
- the programs do not even have to be written in the same language
  - MPI officially supports C and Fortran
- however, in most cases:
  - programmer writes one generic program
  - compiles this
  - at run-time, specifies number of processes
  - run-time system
    - instantiates that number of processes
    - distributes them where they need to go
  - a process can obtain its unique ID ("rank")
    - by branching on rank, each process can execute different code

Communicators and Rank

- a communicator is an MPI abstraction representing a set of processes
  - type: `MPI_Comm`
- processes belonging to a communicator are numbered $0, 1, \ldots n-1$
- n is the size of the communicator
- rank: the number of the process within the communicator
- `MPI_COMM_WORLD`: constant of type `MPI_Comm`
  - pre-defined communicator
  - comprises all processes that exist at start up
- `MPI_Comm_size(MPI_Comm comm, int *size)`
  - stores size of `comm` in `size`
  - returns an error code (0=success)
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - stores rank of calling process in `rank`
  - returns an error code (0=success)

Startup and Shutdown

- `MPI_Init(&argc, &argv)`
  - each process must call this before calling any other MPI functions
  - must be called before reading `argc` or `argv`
  - `MPI_Init(NULL, NULL)`
    - can be used if command line arguments not needed
- `MPI_Finalize()`
  - must be called before process exits
  - no MPI functions can be called after this is called

Hello, world

```
#include<stdio.h>
#include<mpi.h>

int main(int argc, char *argv[]) {
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello from process %d.\n", rank);
  fflush(stdout);
  MPI_Finalize();
}
```

Compiling and Executing

- depends somewhat on the MPI implementation
- standard compilation approach
  - `mpicc [options] -o foo foo.c`
  - just like `cc`
  - results in binary file `foo`
- standard execution approach
  - `mpiexec -n` *numProcs* `./foo`
- on Grendel, Bridges, and other large machines shared by many people:
  - slightly different approach
  - cross-compilation is an option
  - queueing system: SLURM
    - `srun`, `sbatch`, `squeue`, `scancel`, ...

Using the parallel computer Beowulf in CISC 372

- `grendel.cis.udel.edu` is a virtual machine (VM)
- it is not the parallel machine
  - it is used as the interface to the parallel machine Beowulf
- you cannot log on to Beowulf directly
- use Grendel (the VM) to edit, compile, and for other "light" programming tasks
  - or develop/debug on your own machine then use svn to move your work to Grendel
- execute from the VM using SLURM
  - example: `srun -n 10 ./myexecutable`
  - this queues and runs your job on the parallel machine
  - this is the only way you will see performance
  - do not do "big" runs on the VM
  - do not use `mpiexec` on the VM

Example: Boolean Satisfiability

- SAT: The Boolean Satisfiability Problem
- given
  - boolean variables $x_1, \ldots, x_n$
  - a boolean formula ϕ in the $x_1, \ldots, x_n$
    - ϕ may use $\wedge$ (and), $\vee$ (or), and $\neg$ (not)
- determine whether ϕ is satisfiable
  - does there exist a solution?
    - assignments of *true*/*false* to the x_i that lead ϕ to evaluate to *true*
  - additionally: if ϕ is satisfiable, find a/all solution(s)
- example
  - variables x_1, x_2, x_3
  - $\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$
  - ϕ is satisfiable
  - $x_1 =$ *false*, $x_2 =$ *false*, x_3 arbitrary
- example of an unsatisfiable formula: $x_1 \wedge \neg x_1$

SAT

- numerous applications
  - cryptography
  - circuit design: are two digital circuits equivalent?
  - automatic test generation for software or hardware
  - model checking: automatic verification of programs
  - artificial intelligence: planning, ...
- asymptotic complexity?
  - all known algorithms have exponential worst-case time complexity in n
  - it is not known whether you can do better than exponential
  - it is possible a polynomial-time algorithm exists!
  - SAT is an example of a problem in NP: nondeterministic polynomial time
  - it is unknown whether P=NP — the big unsolved problem in computer science
  - if SAT is in P, then P=NP
- many effective SAT solvers exist
  - can solve problems with millions of variables, clauses
  - widely-used in many applications
  - active research area with numerous journals, conferences, competitions

A simple brute-force SAT solver

- iterate over all 2^n assignments to the n boolean variables
  - for each, plug into ϕ and evaluate
- example formula in C:

```
      (v[0]  || v[1])
&& (!v[1] || !v[3])  && (v[2]  || v[3])
&& (!v[3] || !v[4])  && (v[4]  || !v[5])
&& (v[5]  || !v[6])  && (v[5]  || v[6])
&& (v[6]  || !v[15]) && (v[7]  || !v[8])
&& (!v[7] || !v[13]) && (v[8]  || v[9])
&& (v[8]  || !v[9])  && (!v[9] || !v[10])
&& (v[9]  || v[11])  && (v[10] || v[11])
&& (v[12] || v[13])  && (v[13] || !v[14])
&& (v[14] || v[15])
```

Brute force SAT solver: example

	a	b	c	$(\neg a) \wedge (b \vee \neg c)$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

- iterate over integers and extract the base-2 representation of each
- see `sat.c`

The Core Principles of Parallel Computing

Every algorithm applies some operations to some data.

To parallelize the algorithm, you must:

1. divide up the data, and
2. divide up the operations.

Two Goals:

- locality: most operations performed by process P require only the data assigned to P
  - minimize communication!
- load balance: the work is distributed equally among the processes
  - a parallel program is only as fast as the longest-running process

Parallelizing SAT solver

	a	b	c	$(\neg a) \wedge (b \vee \neg c)$
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

- each row is a piece of work — divide these up equally
- locality?
  - each proc operates on its own data; no communication necessary — "embarrassingly parallel"
- load balance?
  - possible issue: some cases can be solved faster than others ("short circuit" nature of *and*, *or*)
  - in the example, last 4 cases are quick
  - note these quick cases tend to be "clumped" together

Dividing up the work

Suppose we have two procs. How to divide up the work between them?

1. Method 1 (block distribution)
   - Proc 0: rows 0,1,2,3
   - Proc 1: rows 4,5,6,7
   - Problem: Proc 1 finishes quickly, then has nothing to do.
   - *program is only as fast as the slowest process*
2. Method 2 (cyclic distribution)
   - Proc 0: rows 0,2,4,6
   - Proc 1: rows 1,3,5,7
     - Probably closer to equal division of work

Load Balancing

Cyclic Distribution

Generalize

Given any number of tasks.

Given p processes.

Distribute the tasks cyclically:

- proc 0: $0, p, 2p, \ldots$
- proc 1: $1, p+1, 2p+1, \ldots$
- proc 2: $2, p+2, 2p+2, \ldots$
- etc.

I.e., proc i gets tasks t, where $t\%p = i$.

See `sat1.c`, `Makefile`.

Adding things up

- now we want to print the total number of solutions found
- each process can count its solutions
- then we need to add up these numbers across all processes
- this obviously requires communication
- an example of a collective operation
  - a communication operation involving all processes in a communicator
- to carry out a collective operation in MPI:
  - each process calls the same function
  - some arguments will be the same for all processes
  - some will differ
- the collective function `MPI_Reduce` can be used to
  - add vectors across all processes
  - store the resulting vector in the memory of one process

MPI_Reduce

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)
```

`sendbuf`	address of send buffer (`void*`)
`recvbuf`	address of recv buffer (`void*`)
`count`	number of elements in send buffer (`int`)
`datatype`	data type of elements in send buffer (`MPI_Datatype`)
`op`	reduce operation (`MPI_Op`)
`root`	rank of root process (`int`)
`comm`	communicator (`MPI_Comm`)

Rank 0 `sendbuf`	x_{00}	x_{01}	x_{02}
Rank 1 `sendbuf`	x_{10}	x_{11}	x_{12}
Rank 2 `sendbuf`	x_{20}	x_{21}	x_{22}

Root `recvbuf`	$x_{00} + x_{10} + x_{20}$	$x_{01} + x_{11} + x_{21}$	$x_{02} + x_{12} + x_{22}$

`MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)`

- all processes in the communicator must call it
- all pass same value for `root`, `comm`, `op`
- in most cases, all pass same values for `count`, `datatype`
- there is no requirement on the pointer values
  - each lives in a "different world" (no shared memory)
- the `recvbuf` argument is only used on the root process
  - all other processes ignore this argument
- if you break any of the rules
  - anything could happen
  - you might get an error message
  - your program might run and just return erroneous results
  - you might get a deadlock
  - you might get a crash with an indecipherable error message
  - the MPI Standard does not specify

Reduction Operations

Predefined reduction operations:

MPI_Op	binary operation	C operation
MPI_SUM	addition	+
MPI_PROD	multiplication	*
MPI_MAX	maximum	x>=y ? x : y
MPI_MIN	minimum	x<y ? x : y
MPI_LAND	logical *and*	&&
MPI_LOR	logical *or*	||
MPI_LXOR	logical *exclusive or*	
MPI_BAND	bit-wise *and*	&
MPI_BOR	bit-wise *or*	|
MPI_BXOR	bit-wise *exclusive or*	^

Can also make user-defined reduction operations.

Datatypes

Some common MPI datatypes:

MPI_Datatype	C type
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char

- See MPI Standard 3.1, Section 3.2.2, "Message Data", Table 3.2
  - "Predefined MPI datatypes corresponding to C datatypes"
- food for thought
  - why did MPI Forum re-invent the data structure wheel?
- now examine `sat2.c`
  - see how `MPI_Reduce` is used

Creating global synchronization points: `MPI_Barrier`

`MPI_Barrier(comm)`

`comm` communicator (`MPI_Comm`)

- another collective operation
- blocks calling process until all processes in `comm` call `MPI_Barrier`
- "no one can leave until everyone enters"
  - the motto of the barrier
- if one process in `comm` calls `MPI_Barrier(comm)`, all should
  - else deadlock ensues
- example: all procs say "hello", barrier, then "goodbye"
- example: write a "hello, world" program, but:
  - messages are printed in order of increasing rank
  - solution: loop with barrier

Keeping track of time: `MPI_Wtime()`

Stands for wall time. From MPI Standard 2.2:

> *MPI defines a timer. A timer is specified even though it is not "message-passing," because timing parallel programs is important in "performance debugging" and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers.*

- returns a floating-point number
  - the number of seconds elapsed since some fixed time in the past
  - "time in the past" is not specified, but is fixed for the life of the process
  - e.g.: midnight on Jan. 1, 1970
- typical usage
  - `t0 = MPI_Wtime();`
  - do some computation
  - `t1 = MPI_Wtime();`
  - it took `t1-t0` seconds to do the computation

MPI_Wtime, cont.

Issue:

- when is a task that involves multiple processes completed?
  - when the first process finishes? the average process?
  - **when the last process finishes**
- you cannot time just one process

Solution (see `sat3.c`):

1. isolate the region of code you want to time (e.g.: you might want to exclude I/O)
2. `MPI_Barrier(comm);`
3. `t0 = MPI_Wtime();`
4. do some computation
5. `MPI_Barrier(comm);`
6. `t1 = MPI_Wtime();`
7. elapsed time is `t1-t0`
   - result should be roughly the same on every process