

Bulk Synchronous and SPMD Programming

CS315B Lecture 2

Prof. Aiken CS 315B Lecture 2

1

The Bulk Synchronous Model

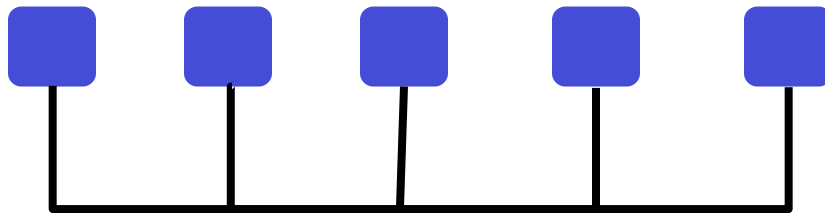
Prof. Aiken CS 315B Lecture 2

2

Bulk Synchronous Model

- A model
 - An idealized machine
- Originally proposed for analyzing parallel algorithms
 - Leslie Valiant
 - "A Bridging Model for Parallel Computation", 1990

The Machine



What are some properties of this machine model?

Computations

- *A sequence of supersteps:*
 - Repeat:
 - *All processors do local computation*
 - *Barrier*
 - *All processors communicate*
 - *Barrier*

What are properties of this computational model?

Basic Properties

- Uniform
 - compute nodes
 - communication costs
- Separate communication and computation
- Synchronization is global

The Idea

- Programs are
 - written for v virtual processors
 - run on p physical processors
- If $v \geq p \log p$ then
 - Managing memory, communication and synchronization can be done automatically within a constant factor of optimal

How Does This Work?

- Roughly
 - Memory addresses are hashed to a random location in the machine
 - Guarantees that on average, memory accesses have the same cost
 - The extra $\log p$ factor of threads are multiplexed onto the p processors to hide the latency of memory requests
 - The processors are kept busy and do no more compute than necessary

SPMD

Terminology

- SIMD
 - Single Instruction, Multiple Data
- SPMD
 - Single Program, Multiple Data

SIMD = Vector Processing

```
if (factor == 0)
    factor = 1.0
A[1..N] = B[1..N] * factor;
j += factor;
```

Picture

```
if (factor == 0)
    factor = 1.0
```

```
A[1] = B[1] *
    factor
```

```
A[2] = B[2] *
    factor
```

```
A[3] = B[3] *
    factor
```

```
...
```

```
j += factor
```

Comments

- Single thread of control
 - Global synchronization at each program instruction
- Can exploit fine-grain parallelism
 - Assumption of hardware support

SPMD = Single Program, Multiple Data

SIMD

```
if (factor == 0)
  factor = 1.0
A[1..N] = B[1..N] *
  factor;
j += factor;
...
```

SPMD

```
if (factor == 0)
  factor = 1.0
A[myid] = B[myid] *
  factor;
j += factor;
...
```


Picture

```
if (factor == 0)
  factor = 1.0
```

```
if (factor == 0)
  factor = 1.0
```

```
A[1] = B[1] *
  factor
```

```
A[2] = B[2]
  * factor
```

```
...
```

```
j += factor
```

```
j += factor
```

Prof. Aiken CS 315B Lecture 2

17

Comments

- Multiple threads of control
 - One (or more) per processor
- Asynchronous
 - All synchronization is programmer-specified
- Threads are distinguished by **myid**
- Choice: Are variables local or global?

Prof. Aiken CS 315B Lecture 2

18

Comparison

- **SIMD**
 - Designed for tightly-coupled, synchronous hardware
 - i.e., vector units
- **SPMD**
 - Designed for clusters
 - Too expensive to do synchronization on each statement, need a looser model

MPI

- **Message Passing Interface**
 - A widely used standard
 - Runs on everything
- A runtime system
- Most popular way to write SPMD programs

MPI Programs

- Standard sequential programs
 - All variables are local to a thread
- Augmented with calls to the MPI interface
 - SPMD model
 - Every thread has a unique identifier
 - Threads can send/receive messages
 - Synchronization primitives

MPI Point-to-Point Routines

- `MPI_Send(buffer, count, type, dest, ...)`
- `MPI_Recv(buffer, count, type, source, ...)`

Example

```
for (...) {
// p = number of chunks of 1D grid, id = process id, h[] = local chunk of the grid
// boundary elements of h[] are copies of neighbors boundary elements

... Local computation ...

// exchange with neighbors on a 1-D grid
if ( 0 < id )
    MPI_Send ( &h[1], 1, MPI_DOUBLE, id-1, 1, MPI_COMM_WORLD );
if ( id < p-1 )
    MPI_Recv ( &h[n+1], 1, MPI_DOUBLE, id+1, 1, MPI_COMM_WORLD, &status );
if ( id < p-1 )
    MPI_Send ( &h[n], 1, MPI_DOUBLE, id+1, 2, MPI_COMM_WORLD );
if ( 0 < id )
    MPI_Recv ( &h[0], 1, MPI_DOUBLE, id-1, 2, MPI_COMM_WORLD, &status );

... More local computation ...

}
```

MPI Point-to-Point Routines, Non-Blocking

- `MPI_Isend(...)`
- `MPI_Irecv(...)`

- `MPI_Wait(...)`

Example

```
for (...) {
// p = number of chunks of 1D grid, id = process id, h[] = local chunk of the grid
// boundary elements of h[] are copies of neighbors boundary elements

... Local computation ...

// exchange with neighbors on a 1-D grid
if ( 0 < id )
    MPI_Isend ( &h[1], 1, MPI_DOUBLE, id-1, 1, MPI_COMM_WORLD );
if ( id < p-1 )
    MPI_Irecv ( &h[n+1], 1, MPI_DOUBLE, id+1, 1, MPI_COMM_WORLD, &status );
if ( id < p-1 )
    MPI_Isend ( &h[n], 1, MPI_DOUBLE, id+1, 2, MPI_COMM_WORLD );
if ( 0 < id )
    MPI_Irecv ( &h[0], 1, MPI_DOUBLE, id-1, 2, MPI_COMM_WORLD, &status );
MPI_Wait(...1...)
MPI_Wait(...2...)

... More local computation ...
}
```

MPI Collective Communication Routines

- MPI_Barrier(...)
- MPI_Bcast(...)
- MPI_Scatter(...)
- MPI_Gather(...)
- MPI_Reduce(...)

Typical Structure

```
communicate_get_work_to_do();  
barrier;           // not always needed  
do_local_work();  
barrier;  
communicate_write_results();
```

What does this remind you of?

PGAS Model

- *PGAS = Partitioned Global Address Space*
- There is one global address space
- But each thread owns a partition of the address space that is more efficient to access
 - i.e., the local memory of a processor
- Equivalent in functionality to MPI
 - But typically presented as a programming language
 - Examples: Split-C, UPC, Titanium

PGAS Languages

- No library calls for communication
- Instead, variables can name memory locations on other machines

```
// Assume y points to a remote location  
// The following is equivalent to a send/receive  
x = *y
```

PGAS Languages

- Also provide collective communication
- Barrier
- Broadcast/Reduce
 - 1-many
- Exchange
 - All-to-all

PGAS vs. MPI

- Programming model very similar
 - Both provide SPMD
- From a pragmatic point of view, MPI rules
 - Easy to add MPI to an existing sequential language
- For productivity, PGAS is better
 - Programs filled with low-level details of MPI calls
 - PGAS programs easier to modify
 - PGAS compilers can know more/do better job

Prof. Aiken CS 315B Lecture 2

31

Summary

- SPMD is well-matched to cluster programming
 - Also works well on shared memory machines
- One thread per core
 - No need for compiler to discover parallelism
 - No danger of overwhelming # of threads
- Model exposes memory architecture
 - Local vs. Global variables
 - Local computation vs. sends/receives

Prof. Aiken CS 315B Lecture 2

32

Analysis

Control

SIMD

```
if (factor == 0)
  factor = 1.0
forall (i = 1..N)
  A[i] = B[i] * factor;
j += factor;
...
```

SPMD

```
if (factor == 0)
  factor = 1.0
A[myid] = B[myid] *
  factor;
j += factor;
...
```

Control, Cont.

- SPMD replicates the sequential part of the SIMD computation
 - Across all threads!
- Why?
 - Often cheaper to replicate computation in parallel than compute in one place and broadcast
 - A general principle . . .

Global Synchronization Revisited

- In the presence of non-blocking global memory operations, we also need memory fence operations
- Two choices
 - Have a separate classes of memory and control synchronization operations
 - E.g., `barrier` and `memory_fence`
 - Have a single set of operations
 - E.g., `barrier` implies memory and control synchronization

Message Passing Implementations

- Idea: A memory fence is a special message sent on the network; when it arrives, all the memory operations are complete
- To work, underlying message system must deliver messages in order
- This is one of the key properties of MPI
 - And most message systems

Bulk Synchronous/SPMD Model

- Easy to understand
- Phase structure guarantees no data races
 - Barrier synchronization also easy to understand
- Fits many problems well

But ...

- Assumes 2-level hierarchy
 - Local/global, a flat collection of homogenous sequential processors
- No overlap of communication and computation
- Barriers scale poorly with machine size
 - (# of operations lost) * (# of processors)

Hierarchy

- Current & future machines are more hierarchical
 - 3-4 levels, not 2
- Leads to programs written in a mix of
 - MPI (network level)
 - OpenMP (node level) + vectorization
 - CUDA (GPU level)
- Each is a different programming model

No Overlap of Computation/Communication

- Leaves major portion of the machine idle in each phase
- And this potential is lost at many scales
 - Hierarchical machines again
- Increasingly, communication is key
 - Data movement is what matters
 - Most of the execution time, most of the energy

Global Operations

- Global operations (such as barriers) are bad
 - Require synchronization across the machine
 - Especially bad when there is performance variation among participating threads
- Need a model that favors asynchrony
 - Couple as few things together as possible

Global Operations Continued

- MPI has evolved to include more asynchronous and point-to-point primitives
- But these do not always mix well with the collective/global operations

I/O

- How do programs get initial data? Produce output?
- In many models answer is clear
 - Passed in and out from root function
 - Map-Reduce
 - Multithreaded shared memory applications just use the normal file system interface

I/O, Cont.

- Not clear in SPMD
- Program begins running and
 - Each thread is running in its own address space
 - No thread is special
 - Neither “master” nor “slave”

I/O, Cont.

- Option 1
 - Make thread 0 special
 - Thread 0 does all I/O on behalf of the program
 - Issue: Awkward to read/write large data sets
 - Limited by thread 0's memory size
- Option 2
 - Parallel I/O
 - Each thread has access to its own file system
 - Containing distributed files
 - Each file “f” a portion of a collective file “f”

I/O Summary

- Option 2 is clearly more SPMD-ish
- Creating/deallocating files requires a barrier
- In general, parallel programming languages have not paid much attention to I/O

Next Week

- Intro to Regent programming
- First assignment
 - Please get your accounts on Certainty