MPI
AmorphOS

Chris Rossbach

cs378 Fall 2018

11/12/2018
Outline for Today

• Questions?
• Administrivia
  • Keep thinking about projects
  • FPGA lab web updates
    • Required input sizes and memory
  • Michael Wei (VMware) visit
• Agenda
  • Exam
  • MPI
  • AmorphOS (if we have time)

Acknowledgements:
• Portions of the lectures slides were adopted from:
  • Argonne National Laboratory, MPI tutorials.
  • Lawrence Livermore National Laboratory, MPI tutorials.
  • See online tutorial links in course webpage
Project Proposal

CS378: Concurrency

The goal of this assignment is to come up with a plan for your 2 labs. The project is a more open-ended assignment, where you have the freedom to choose your own project. I encourage you to come up with your own project idea, but the options are wide.

You must submit a proposal (1-2 pages long), meeting the guidelines:

- Provide a detailed timeline of how you plan to build the system by the deadline. Give a list of 4 key milestones.
- What infrastructure will you have to build to run the experiment?
- What hardware will you need and where will you get it?
- What kind of experiments do you plan to run?
- How will you know if you have succeeded?
- What kind of performance or functionality problems do you expect to encounter?

Planning is important. So I will review your proposal and give feedback on it.

You can work in groups on your project.

- A very good example

- Update MapReduce Lab to use Spark
- Resurrect cs372 PPoPP lab
- Build an STM (Software Transactional Memory)
- Use Intel HTM for something interesting
- Write a thread pool
- Build a lock-free data structure
- Parallelize an algorithm with [Akka, Spark, Hadoop, Flink, Beam, Storm, etc.]
- Build/Measure a TensorFlow Model
- Parallelize something with [Pony, Julia, Chapel, X10, MPI]
- Do another FPGA algorithm
- More GPU parallelization
- Do the MPI lab (#6) at scale
- ...

Questions?
Exam

• Graded, entered in Canvas
  • There sure are a lot of you!

• Mean: 78.8; Median 82.9; range: [~36...~93]

Favorite subjects:
  Go, GPUs
Least Favorite:
  GPUs, TM
Exam: 1.1.

1. In a uniprocessor system concurrency control is best implemented with
   
   (a) Semaphores
   
   (b) Spinlocks
   
   (c) Interrupts
   
   (d) Atomic instructions
   
   (e) Bus locking
   
   (f) Processes and threads
2. Which of the following are true of threads?

(a) They have their own page tables.

(b) Data in their address space can be either shared with or made inaccessible to other threads.

(c) They have their own stack.

(d) They must be implemented by the OS.

(e) Context switching between them is faster than between processes.
Exam: 1.3.

3. Which of the following conditions are true when deadlock occurs?

(a) Mutual exclusion.
(b) Resources are released while waiting for locks.
(c) Processes cannot be preempted while holding a lock.
(d) Circular wait.
(e) Starvation occurs.

Wait...why not mutual exclusion?
Circular wait can happen without it!
- E.g. can RWLocks deadlock? Yes.
- Can you deadlock on resources other than locks? Yes
- However, I didn’t take/give points for A
Exam: 1.4.

4. If a program exhibits strong scaling,
   
(a) It gets faster really dramatically with more threads.

(b) Increasing the amount of work does not increase its run time.

(c) Its serial phases are short relative to its parallel phases.

(d) Adding more threads decreases the end-to-end runtime for an input.

(e) Adding more threads and more work makes it go about the same speed.
Exam: 1.5.

5. Barriers can be used to implement

(a) Cross-thread coordination.
(b) Mutual exclusion.
(c) Slow parallel programs.
(d) Task-level parallelism.
Exam: 2.1

1. In the context of lock-based mutual exclusion, define safety, liveness, bounded waiting, and failure atomicity.

- Safety
  - Only one thread in the critical region

- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - *Even if other thread takes forever in non-critical region*

- Bounded waiting
  - A thread that enters the entry section enters the critical section within some *bounded number of operations*. *The bound is on the number of times other threads entirety of the critical section before it.*

- Failure atomicity
  - It is OK for a thread to die in the critical region
    - *Dying in critical region → ops all applied or all rolled back*
  - Many techniques do not provide failure atomicity
Exam: 2.4.

4. In message-passing systems, channel implementations may or may not use buffering/capacity, and may support blocking and/or non-blocking semantics. (A) Can a 0-capacity channel support non-blocking send and receive semantics? Why or why not? (B) How is direct addressing (naming) different from indirect addressing for message passing systems? List a potential advantage and disadvantage for each. (C) What constructs enable Go’s channels to support both blocking and non-blocking semantics? (D) When shouldn’t you close a Go channel from the receiving go routine?

- A) In general no, but receiver can poll
- C) Select!

```go
select {
    case v1 := <-c1:
        fmt.Println("received \%v from \%c1\n", v1)
    case v2 := <-c2:
        fmt.Println("received \%v from \%c2\n", v2)
    case c3 := 23:
        fmt.Println("sent \%v to \%c3\n", 23)
    default:
        fmt.Println("no one was ready to communicate\n")
}
```
Exam: 3.1.

1. Consider the barrier implementation and usage scenario below:

```java
class Barrier {
    protected:
    int m_nArrived;
    int m_nThreads;
    int m_bGo;

    public:
    Barrier(int nThreads) {
        m_nThreads = nThreads;
        m_nArrived = 0;
        m_bGo = 0;
    }

    void Wait() {
        int nOldArr = atomic_inc(&m_nArrived, 1);
        if(nOldArr == m_nThreads - 1) {
            m_nArrived = 0;
            m_bGo = 1;
        } else {
            while(m_bGo == 0) {
                // spin
            }
        }
    } 
}
```

void worker_thread_proc(void *vtid) {
    int tid = (*((int*) vtid));
    for(int i=0; i<100; i++) {
        g_Barrier->Wait();
        compute_my_partition(tid); // compute bound phase
    }
}

Barrier * g_pBarrier = NULL;
int main(int argc, char**argv) {
    int nThreads = 16;
    int tids[nThreads];
    pthread_t threads[nThreads];
    g_pBarrier = new Barrier(nThreads);
    for(int i=0; i<nThreads; i++) {
        tids[i] = i;
        pthread_create(&threads[i], NULL, worker_thread_proc, &tids[i]);
    }
}
```

The implementation has both correctness and performance issues. (A) Suppose the implementation were indeed correct, describe at least one change that could make the implementation more efficient for very short critical sections (e.g. the `compute_my_partition()` function is very fast). (B) Describe at least one change that could make the implementation more efficient for very long critical sections (`compute_my_partition()` takes a very long time). (C) There is a correctness problem with the implementation. What is it, and what is the most natural way to fix it?

- A) spin on local go flag
- B) some kind of blocking
- C) barrier doesn’t reset, some strategy to make it reset
2. (A) How are promises and futures related? As we’ve discussed, there is disagreement on the nomenclature, so don’t worry about which is which; just describe what the different objects are and how they function. (B,C) Consider the following go-like code:

```go
func main() {
    data1 := readAndParseFile(options.getPath1())
    data2 := readAndParseFile(options.getPath2())
    result := computeBoundOperation(data1, data2)
    writeResult(options.getOutputPath())
}
```

(B) Re-write the code to use asynchronous processing where-ever possible, using `go func()` for each of the steps and using WaitGroups to enforce the correct ordering amongst them. Don’t worry about syntax being correct, just focus on the important concurrency-relevant ideas. (C) Suppose `WaitGroup` support were not available. Describe at least one approach that can still ensure the proper ordering between goroutines correctly without requiring `WaitGroups`. (D) Asynchronous systems are often decried as prone to “stack-ripping”. What does this mean? Does go suffer these drawbacks? Why/why not?

- A) something about futures and promises
- B) pretty much anything with go func()
- C) Channels!
- D) Stack-ripping → some creative responses
  - (next slide)
Stack-Ripping

Requests must carry state

Stack-based state out-of-scope!
Exam 3.3

```c
int tid1 = txbegin(NULL); // NULL parent transaction
    tid1.write(key1, value1); // Write the value value1 to the entry whose key is key1
int tid2 = txbegin(tid1); // tid1 is the parent transaction
    tid2.write(key2, value2);
    txcommit(tid2);
    tid1.read(key2);
}
    txcommit(tid1);
```

(A) One strategy is for the inner transaction to commit normally, but also produce an “undo” list of updated values that can be used to restore the original values if the outer transaction aborts. Which ACID condition(s) does this approach relax? Why? (B) Another strategy is for each inner transaction to produce a list of deferred updates/actions that the the outer transaction commits for it when the outer transaction commits. For any data item written in any transaction, all transactions read the last update value from this list. Which ACID condition(s) does this approach relax? (C) If the only data flow is that the inner transaction reads from the outer transaction (meaning tid2 reads tid1’s writes but tid1 never reads tid2’s writes), do we still need to relax ACID? Why?

- **A**: Commit normally, produce undo list for abort
  - Durability – other transactions see effects *that get rolled back*
  - Isolation – other transactions see inner effects while outer still in flight
  - Atomicity – effects of txid2 are part of txid1 and they become visible without the rest of txid1

- **B**: Produce deferred actions, all *in-flight* txns read from this list
  - Isolation – uncommitted inner txn writes visible to all (e.g. txid3)
  - Not durability, since commit doesn’t occur until out commits
  - Consistency – can be argued either way. Outer txid1 certainly gets consistent view. Any Txid3 that reads from list sees inconsistent view

- **C**: if inner only reads outer writes, do we need to relax ACID?
  - No, inner sees outers writes but would abort if outer aborts → standard flat nesting
  - Caveat: txid1 writes the same data more than once

- **Caveat**: txid1 writes the same data more than once
Last Thoughts on Exam

• Point was to get people to think
• Don’t worry about your grade
  • Everyone is still “above water”
• Feel free to come talk to me about it if I misread/misunderstood...
Faux Quiz Questions

• What is the difference between horizontal and vertical scaling? Describe a setting or application in which horizontal might be preferable and one in which vertical scaling makes more sense.
• What is PGAS?
• What is bisection bandwidth?
• What is a “shared nothing” architecture?
• What is an All-reduce operation? In MPI can it be implemented with other primitives (e.g. send/recv)? Why does MPI support it as a first class API?
• What is distributed shared memory? Suggest some implementation techniques.
• What is a collective operation? Give an example and explain why it is a collective.
• What is a 3-Tier architecture? What application (s) might be it good for and why?
• What are some advantages and disadvantage of distributed memory architectures?
• What is the difference between 1-sided and cooperative MPI operations? Advantages/disadvantages of each?
Scale out vs Scale up

<table>
<thead>
<tr>
<th>Vertical Scaling</th>
<th>Horizontal Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher Capital Investment</td>
<td>On Demand Investment</td>
</tr>
<tr>
<td>Utilization concerns</td>
<td>Utilization can be optimized</td>
</tr>
<tr>
<td>Relatively Quicker and works with the</td>
<td>Relatively more time consuming and</td>
</tr>
<tr>
<td>current design</td>
<td>needs redesigning</td>
</tr>
<tr>
<td>Limiting Scale</td>
<td>Internet Scale</td>
</tr>
</tbody>
</table>

Vertical Scaling ↑
Make boxes bigger

Horizontal Scaling →
Make more boxes
Architects Wanted

1. User Browses Potential Pets
2. Clicks “Purchase Pooch”
3. Web Server, CGI/EJB + Database complete request
4. Pooch delivered (not shown)

How to handle lots and lots of dogs?
3 Tier architecture

- Web Servers (Presentation Tier) and App servers (Business Tier) scale horizontally.
- Database Server scales vertically.
- Horizontal Scale → “Shared Nothing”

Why is this a good arrangement?

Vertical scale gets you a long way, but there is always a bigger problem size.
Goal
Design Space

- Internet
- Private data center
- Shared memory

Latency

Throughput

Throughput

Latency
Parallel Architectures of Interest for MPI

Distributed Memory Multiprocessor
- Messaging between nodes

- Massively Parallel Processor (MPP)
- Many, many processors

Cluster of SMPs
- Shared memory within SMP node
- Messaging between SMP nodes

- Can also be regarded as MPP if processor number is large

Multicore SMP+GPU Cluster
- Shared mem in SMP node
- Messaging between nodes

What have we left out?
- Uniprocessors
- CMPs
- Non-GPU Accelerators
- Fused Accelerator
What requires extreme scale?

• Simulations—why?
  • Simulations are sometimes more cost effective than experiments

• Why extreme scale?
  • More compute cycles, more memory, etc, lead for faster and/or more accurate simulations

Nuclear Reactors

Image credit: Prabhat, LBNL

Climate Change

Astrophysics
How big is “extreme” scale?

Measured in FLOPs

- **FLOating point operations**
  - 1 GigaFLOP = 1 billion FLOPs
  - 1 TeraFLOP = 1,000 GigaFLOPs
  - 1 PetaFLOP = 1,000 TeraFLOPs
  - Most current supercomputers:
    - 1 ExaFLOP = 1,000 PetaFLOPs
    - Arriving in 2018 (supposedly)

### Top 5 Supercomputers

<table>
<thead>
<tr>
<th>Rank</th>
<th>System</th>
<th>Cores</th>
<th>Rmax [TFlop/s]</th>
<th>Rpeak [TFlop/s]</th>
<th>Power [kW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China</td>
<td>10,649,600</td>
<td>33014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P, NUDT National Super Computer Center in Guangzhou China</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, NVIDIA Tesla P100, Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>4</td>
<td>Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1541, Infiniband EDR, PEZY-SC2 700Mhz, ExaScaler Japan Agency for Marine-Earth Science and Technology, Japan</td>
<td>200 PFLOPS</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
</tr>
<tr>
<td>5</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, Cray Inc. DOE/SC/Oak Ridge National Laboratory United States</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
</tbody>
</table>
Distributed Memory Multiprocessors

... Or how to program 10,000,000 Cores

• Each processor has a local memory
  • Physically separated memory address space

• Processors communicate to access non-local data
  • Message communication (message passing)
    • Message passing architecture
  • Processor interconnection network

• Parallel applications partitioned across
  • Processors: execution units
  • Memory: data partitioning

• Scalable architecture
  • Incremental cost to add hardware (cost of node)

Network

- Nodes: complete computer
  • Including I/O
- Nodes communicate via network
  • Standard networks (IP)
  • Specialized networks (RDMA, fiber)
Performance Metrics: Latency and Bandwidth

• Bandwidth
  • Need high bandwidth in communication
  • Match limits in network, memory, and processor
  • Network interface speed vs. network bisection bandwidth

• Latency
  • Performance affected since processor may have to wait
  • Harder to overlap communication and computation
  • Overhead to communicate is a problem in many machines

• Latency hiding
  • Increases programming system burden
  • Examples: communication/computation overlaps, prefetching

Wait... WTF... bisection bandwidth?

if network is bisected, bisection bandwidth is the bandwidth between the two partitions

Is this different from metrics we’ve cared about so far?
Ostensible Advantages of Distributed Memory Architectures

• Hardware simpler (especially versus NUMA), more scalable
• Communication explicit, simpler to understand
• Explicit communication →
  • focus attention on costly aspect of parallel computation
• Synchronization →
  • naturally associated with sending messages
  • reduces possibility for errors from incorrect synchronization
• Easier to use sender-initiated communication →
  • some advantages in performance

Can you think of any disadvantages?
Outline

• ...  
• Practicalities: running on a supercomputer  
• Message Passing  
• MPI Programming  
• MPI Implementation
Running on Supercomputers

Practicalities in a nutshell

- Programmer plans a **job**; job ==
  - parallel binary program
  - “input deck” (specifies input data)
- Submit job to a **queue**
- Scheduler allocates resources when
  - resources are available,
  - (or) the job is deemed “high priority”

- Sometimes 1 job takes whole machine
  - These are called “hero runs”...
- Sometimes many smaller jobs
- Supercomputers used continuously
  - Processors: “scarce resource”
  - jobs are “plentiful”

- Scheduler runs scripts that initialize the environment
  - Typically done with environment variables
- At the end of initialization, it is possible to infer:
  - What the desired job configuration is (i.e., how many tasks per node)
  - What other nodes are involved
  - How your node’s tasks relates to the overall program
- MPI library interprets this information, hides the details
The Message-Passing Model

- Process: a program counter and address space
- Processes: multiple threads sharing a single address space

- MPI is for communication among **processes**
  - Not threads
- Inter-process communication consists of
  - Synchronization
  - Data movement

How does this compare with CSP?
SPMD

- Data distributed across processes
  - Not shared → shared nothing

“Owner compute” rule: Process that “owns” the data (local data) performs computations on that data
Message Passing Programming

- Defined by communication requirements
  - MPI == *Message-Passing library (Interface) specification*
    - Extended message-passing model
    - Not a language or compiler specification
    - Not a specific implementation or product
  - Targeted for parallel computers, clusters, and NOWs
    - NOWs = network of workstations
  - Specified in C, C++, Fortran 77, F90
  - Message Passing Interface (MPI) Forum
    - [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html)

- Two flavors for communication
  - Cooperative operations
  - One-sided operations
Cooperative Operations

- Data is cooperatively exchanged in message-passing
- Explicitly sent by one process and received by another
- Advantage of local control of memory
  - Change in the receiving process’s memory made with receiver’s explicit participation
- Communication and synchronization are combined

```
Process 0
  Send(data)
  time
Process 1
  Receive(data)
```

Familiar argument?
One-Sided Operations

• One-sided operations between processes
  • Include remote memory reads and writes
• Only one process needs to explicitly participate
  • There is still agreement implicit in the SPMD program
• Implication:
  • Communication and synchronization are decoupled

Are 1-sided operations better for performance?
A Simple MPI Program (C)

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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```
A Simple MPI Program (C++)

```cpp
#include <iostream.h>
#include "mpi++.h"

int main( int argc, char *argv[] )
{
    MPI::Init(argc,argv);
    cout << "Hello, world!" << endl;
    MPI::Finalize();
    return 0;
}
```
MPI_Init

• Hardware resources allocated
  • MPI-managed ones anyway...

• Start processes on different nodes
  • Where does their executable program come from?

• Give processes what they need to know
  • Wait...what do they need to know?

• Configure OS-level resources

• Configure tools that are running with MPI
  • ...

38

Introduction to Parallel Computing, University of Oregon, IPCC
MPI_Finalize

- Why do we need to finalize MPI?
- What is necessary for a “graceful” MPI exit?
  - Can bad things happen otherwise?
  - Suppose one process exits...
- How do resources get de-allocated?
- How to shut down communication?
- What type of exit protocol might be used?
  - By default, an error causes all processes to abort
  - The user can cause routines to return (with an error code)
    - In C++, exceptions are thrown (MPI-2)
  - A user can also write and install custom error handlers
  - Libraries may handle errors differently from applications

Executive Summary
- Undo all of init
- Be able to do it on success or failure exit
Running MPI Programs

• MPI-1 does not specify how to run an MPI program

• Starting an MPI program is dependent on implementation
  • Scripts, program arguments, and/or environment variables

• % mpirun -np <procs> a.out
  • For MPICH under Linux

• mpiexec <args>
  • Recommended part of MPI-2, as a recommendation
  • mpiexec for MPICH (distribution from ANL)
  • mpirun for SGI’s MPI
Finding Out About the Environment

• Two important questions that arise in message passing
  • How many processes are being use in computation?
  • Which one am I?

• MPI provides functions to answer these questions
  • MPI_Comm_size reports the number of processes
  • MPI_Comm_rank reports the rank
    • number between 0 and size-1
    • identifies the calling process
More “Hello World”

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

What does this program do?
Some Basic Concepts

• Processes can be collected into \textit{groups}
• Each message is sent in a \textit{context}
  • Must be received in the same context!
• A group and context together form a \textit{communicator}
• A process is identified by its \textit{rank}
  • With respect to the group associated with a communicator
• There is a default communicator \texttt{MPI\_COMM\_WORLD}
  • Contains all initial processes
MPI Datatypes

- Message data (sent or received) is described by a triple
  - address, count, datatype
- An MPI *datatype* is recursively defined as:
  - Predefined data type from the language
  - A contiguous array of MPI datatypes
  - A strided block of datatypes
  - An indexed array of blocks of datatypes
  - An arbitrary structure of datatypes

There are MPI functions to construct custom datatypes:
- Array of (int, float) pairs
- Row of a matrix stored columnwise
MPI Tags

• Messages are sent with an accompanying user-defined integer *tag*
  • Assist the receiving process in identifying the message
• Messages can be screened at receiving end by specifying specific tag
  • `MPI_ANY_TAG` matches any tag in a receive
• Tags are sometimes called “message types”
  • MPI calls them “tags” to avoid confusion with datatypes

In my (limited) experience, the least useful MPI feature
MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

• The message buffer is described by:
  • start, count, datatype
• The target process is specified by dest
  • Rank of the target process in the communicator specified by comm
• Process blocks until:
  • Data has been delivered to the system
  • Buffer can then be reused
• Message may not have been received by target process!
Programming MPI with Only Six Functions

• Many parallel programs can be written using:
  • MPI_INIT()
  • MPI_FINALIZE()
  • MPI_COMM_SIZE()
  • MPI_COMM_RANK()
  • MPI_SEND()
  • MPI_RECV()

• Why have any other APIs (e.g. broadcast, reduce, etc.)?

• Point-to-point (send/recv) isn’t always the most efficient...
  • Add more support for communication
if (myID == RootProcess) {

    FILE * fp = fopen(argv[1], "r");
    res = fscanf(fp, "%d\n", &nums);
    for (int p=0; p<num_procs-1; p++) {
        for (int i=0; i<shard_length; i++) {
            res = fscanf(fp, "%d\n", &values[i]);
            assert(res != EOF);
        }
        // _info("ID-%d: sending shard_length:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(&shard_length, 1, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
        // _info("ID-%d: sending shard:%d to ID-%d\n", RootProcess, shard_length, p+1);
        MPI_Send(values, shard_length, MPI_INT, p+1, stag++, MPI_COMM_WORLD);
    }
    /* ... */
} else {

    MPI_Recv(&shard_length, 1, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    values = (int*)calloc(shard_length, sizeof(int));
    MPI_Recv(values, shard_length, MPI_INT, RootProcess, tag, MPI_COMM_WORLD, &status);
    mylength = shard_length;
}
Excerpt: Barnes-Hut

```c
int ctr=nLocalOriginal;
int offset=nLocalOriginal-nLocal;
for(i=0;i<worldSize;i++){
    if(i==rank){
        MPI_Bcast(s_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
    }
    else {
        MPI_Bcast(l_particles,N_POS_ELEMS*nLocalMax+1,MPI_DOUBLE,i,MPI_COMM_WORLD);
        for(k=0;k<l_particles[0];k++, ctr++){
            if(l_particles[MASS(k)]<0){
                offset++;
                nparticles--;
            }
            else {
                s_particles[FX(ctr)]=l_particles[FX(k)];
                s_particles[PY(ctr)]=l_particles[PY(k)];
                s_particles[PZ(ctr)]=l_particles[PZ(k)];
                s_particles[MASS(ctr)]=l_particles[MASS(k)];
                indexes[ctr-offset]=ctr;
            }
        }
    }
}
```
Reduce/Allreduce

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

```
reduce
A0+A1+A2  B0+B1+B2  C0+C1+C2
```

```
Allreduce
A0+A1+A2  B0+B1+B2  C0+C1+C2
```

Int MPI::COMM_WORLD.Allreduce(
void* operand /* in */,
void* result /* out */,
int count /* in */,
MPI::Datatype datatype /* in */,
operator /* in */)

```
Ilreduce
A0+A1+A2  B0+B1+B2  C0+C1+C2
```

```
A0+A1+A2  B0+B1+B2  C0+C1+C2
A0+A1+A2  B0+B1+B2  C0+C1+C2
A0+A1+A2  B0+B1+B2  C0+C1+C2
```
MPI_Reduce

- **MPI_Reduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - IN sendbuf (address of send buffer)
  - OUT recvbuf (address of receive buffer)
  - IN count (number of elements in send buffer)
  - IN datatype (data type of elements in send buffer)
  - IN op (reduce operation)
  - IN root (rank of root process)
  - IN comm (communicator)

- MPI_Reduce combines elements specified by send buffer and performs a reduction operation on them.
- There are a number of predefined reduction operations: MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC
### Reduce_scatter/Scan

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

**reduce-scatter**

| A0+A1+A2 | B0+B1+B2 | C0+C1+C2 |

**scan**

<table>
<thead>
<tr>
<th>A0</th>
<th>B0</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0+A1</td>
<td>B0+B1</td>
<td>C0+C1</td>
</tr>
<tr>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
MPI_Scan

- **MPI_Scan** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - **IN** sendbuf  (address of send buffer)
  - **OUT** recvbuf  (address of receive buffer)
  - **IN** count  (number of elements in send buffer)
  - **IN** datatype  (data type of elements in send buffer)
  - **IN** op  (reduce operation)
  - **IN** comm  (communicator)

- Note: **count** refers to total number of elements that will be received into receive buffer after operation is complete
To use or not use MPI?

• **USE**
  • You need a portable parallel program
  • You are writing a parallel library
  • You have irregular or dynamic data relationships
  • You care about performance

• **NOT USE**
  • You don’t need parallelism at all
  • You can use libraries (which may be written in MPI) or other tools
  • You can use multi-threading in a concurrent environment
    • You don’t need extreme scale
AmorphOS Motivation

Bigger, faster FPGAs deployed in the cloud
  • Microsoft Catapult/Azure
  • Amazon F1

• FPGAs: Reconfigurable Accelerators
  • ASIC Prototyping, Video & Image Proc., DNN, Blockchain
  • Potential solution to *accelerator provisioning challenge*

Our position: FPGAs will be shared
  • Sharing requires protection
  • Abstraction layers provide compatibility
  • Beneficiary: provider → consolidation
FPGA Background

- Field Programmable Gate Array (FPGA)
  - Reconfigurable interconnect → custom data paths
  - FPGAs attached as coprocessors to a CPU

- FPGA Build Cycle
  - Synthesis: HDL → Netlist (~seconds)
  - Place and Route: Netlist → Bitstream (~min--hours)
  - Reconfiguration/Partial Reconfiguration (PR)

- Production systems: No multi-tenancy

- Emerging/Research Systems use fixed slots/PR
  - Fixed-sized slots → fragmentation (50% or more)
  - Elastic resource management needed
AmorphOS Goals

• Protected Sharing/Isolation
  • Mutually distrustful applications

• Compatibility / Portability
  • HDL written to AmorphOS interfaces
  • 14 benchmarks run unchanged on Microsoft Catapult and Amazon F1

• Elasticity
  • User logic scales with resource availability
  • Sharing density scales with availability
AmorphOS Abstractions

- **Zone**: Allocatable Unit of Fabric
  - 1 Global zone
  - N dynamically sized, sub-dividable PR zones
- **Hull**: OS/Protection Layer
  - Memory Protection, I/O Mediation
  - Interfaces form a compatibility layer
- **Morphlet**: Protection Domain
  - Extends Process abstraction
  - Encapsulate user logic on PR or global zone
- **Registry**: Bitstream Cache
  - Hides latency of place-and-route (PaR)

<table>
<thead>
<tr>
<th>Registry</th>
<th>Morphlets</th>
<th>Bitstream</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;A,B&gt;</td>
<td>0x0a1...</td>
</tr>
<tr>
<td></td>
<td>&lt;A,B,C&gt;</td>
<td>0x0fb01...</td>
</tr>
<tr>
<td></td>
<td>&lt;B,C&gt;</td>
<td>0x11ad...</td>
</tr>
</tbody>
</table>
Scheduling Morphlets

• Tradeoff
  • Fixed zones + PR $\rightarrow$ fast, fragmentation
  • Global zone + PaR $\rightarrow$ eliminates fragmentation, slow

• AmorphOS: best of both worlds
  • Low Latency Mode
    • Fixed zones + PR
    • Default Morphlet bitstream
  • High Throughput Mode
    • Combine multiple Morphlets
    • Co-schedule on a global zone
Scheduling Case Study

**Low-Latency Mode**
- Host
- DRAM
- App A
  - Morphlet A
  - FPGA Fabric

**High-Throughput Mode**
- Host
- DRAM
- App A
  - Morphlet A
- App B
  - Morphlet A

**Low-Latency Mode**
- Host
- DRAM
- App A
  - Morphlet A
- App B
  - Morphlet B

**High-Throughput Mode**
- Host
- DRAM
- App A
  - Morphlet A
- App B
  - Morphlet B
- App C
  - Morphlet C
- App D
  - Morphlet D

**T_0**

**T_1**

**T_2**

**T_3**
AmorphOS Hull

• Hardens and extends vendor Shells
  • Microsoft Catapult
  • Amazon F1

• AmorphOS Interfaces
  • Control: \textit{CntrlReg}
  • Virtual Memory: \textit{AMI}
  • Bulk Data Transfer: Simple-\textit{PCIe}
AmorphOS Hull

- Hardens and extends vendor Shells
  - Microsoft Catapult $\rightarrow$ Higher Level
  - Amazon F1 $\rightarrow$ Lower Level

- AmorphOS Interfaces
  - Control: \texttt{CntrlReg}
  - Virtual Memory: \texttt{AMI}
  - Bulk Data Transfer: Simple-\texttt{PCIe}

- Multiplexing of interfaces
  - Isolation/data protection
  - Scalable, 32 accelerators
    - Tree of multiplexers
Implementation & Methodology

• Catapult Prototype
  • Altera Mt. Granite Stratix V GS 2x4GB DDR3, Windows Server
  • Segment-based protection, partial reconfiguration (PR)

• Amazon F1 Prototype
  • Xilinx UltraScale+ VU9P, 4x16GB GDDR4, CentOS 7.5
  • No PR, but much more fabric than Catapult

• Workloads
  • DNNWeaver – DNN inference
  • MemDrive – Memory Bandwidth
  • Bitcoin – blockchain hashing
  • CHStone – 11 accelerators (e.g. AES, jpeg, etc)
Scalability

- MemDrive: 2X throughput
- Bitcoin: compute-bound
- DNNWeaver: • 32X density • 23X throughput

Takeaway: Massive throughput/density improvement possible, awareness of contended resources necessary

- **F1:** Xilinx UltraScale+ VU9P, 4x16GB GDDR4, CentOS 7.5
- **Higher is better, Homogenous Morphlets**
Throughput

- **8 Bitcoin Morphlets**
- **Catapult Altera Stratix V GS 2x4GB DDR3, Windows**
- Registry pre-populated: ctxt sw. 200ms
- Log Scale, Lower is better

**Fixed Zones: worse than no sharing due to down-scaling!**

**Takeaway:** Co-scheduling on a global zone can perform better than fixed-sized slots and PR
Partitioning Policies

Single-Zone
- Multiple Morphlets in a single PR zone
- Divide the Morphlet PR zone into smaller PR zones
- Divide the PR zone into smaller PR zones
- Single Morphlet in a single PR zone
- Everything runs serially
- Single context

Global Zone
- Multiple Morphlets
- No fixed size zones
- Single PR zone

Subdivide
- Divide top-level PR zone into smaller PR zones
Partitioning Policies

- Bitcoin Morphlets
- Catapult: Altera Mt. Granite Stratix V GS 2x4GB DDR3, Windows
- Registry pre-populated: ctxt sw. 200ms
- Higher is better

Takeaway:
- Hierarchical PR on limited HW not worth it
- See paper for projections on F1
Related Work

• **Access to OS-managed resources**
  • Borph: So [TECS ’08, Thesis ’07]
  • Leap: Adler [FPGA ‘11]
  • CoRAM: Chung [FPGA ‘11]

• **First-class OS support**
  • HThreads: Peck [FPL’06], ReconOS: Lübbers [TECS ’09] -- extend threading to FPGA SoCs
  • MURAC: Hamilton [FCCM ’14] – extend process abstraction to FPGAs

• **Single-application Frameworks**
  • Catapult: Putnam [ISCA ‘14] / Amazon F1

• **Fixed-slot + PR**
  • OpenStack support: Chen [CF ‘14], Byma [FCCM ’14]; Fahmy [CLOUDCOM ‘15];
  • Disaggregated FPGAs: Weerasinghe [UIC-ATC-ScalCom ‘15]

• **Overlays**
  • Zuma: Brant [FCCM ‘12],
  • Hoplite: Kapre [FPL ‘15],
  • ReconOS+Zuma: [ReConfig ’14]
Conclusions & Future Work

• Compatibility Improved
  • without restricting programming model
  • Comprehensive set of stable interfaces
  • Port AmorphOS *per platform not each accelerator per platform*

• Scalability achieved *within and across accelerators*
  • AmorphOS transparently scales morphlets up/down
  • Powerful combination of slots/Partial Reconfiguration and full FPGA bitstreams

• Future work
  • Transparently scale across multiple FPGAs
  • Scale across more than just FPGAs
  • Open source AmorphOS/port to more platforms
MPI_Allreduce

```c
int MPI::COMM_WORLD.Allreduce(
    void*,
    void*,
    int,
    MPI::Datatype,
    MPI::Op,
    operand /* in */,
    result /* out */,
    count /* in */,
    datatype /* in */,
    operator /* in */
)
```