Today's Class

High Throughput vs High Performance:
- Rationale and Examples

Sampling Engines & Tasking Strategies
- Divide-and-Conquer, Producer-Consumer

Message Passing Model
- Review & Limitations

PGAS Model
- UPC++, rationale and examples.
General Info

Grading Review and Feedback
- Moodle: Short statement, just for coarse-grained feedback
- More info: Send your grader an email
- In-depth review: Arrange a meeting with your grader

Homework 4
- Deadline in 3 Weeks
- Will contain bonus items -> +0.25 to any HW if done right

Next Week (8/4)
- There will be practice session -> UPC++ Practice
- There will be NO lecture.

Remember to get your Piz Daint Account!
High Throughput vs High Performance
Single-Core Conduit

Sample Population (Generation 0)

Sample 1  Sample 2  Sample 3  Sample 4  Sample 5  Sample 6  Sample n

Node 0
Core 0

Node 0
Core 1

Node 0
Core 2

Node 0
Core 3

Node 1
Core 0

Node 1
Core 1

Node 1
Core 2

Node 1
Core 3

2 Nodes
4 Cores Each
Single-Core Conduit & Single-Core Model

Node 0 Core 0
Node 0 Core 1
Node 0 Core 2
Node 0 Core 3
Node 1 Core 0
Node 1 Core 1
Node 1 Core 2
Node 1 Core 3

Single-Core Sampler Conduit - Single-Core Model

Sample 0
Sample 1

Wasted Resources
Module: Distributed Memory Conduit

Sample Population (Generation 0)

- Sample 1
- Sample 2
- Sample 3
- Sample 4
- Sample 5
- Sample 6
- Sample n

Two Nodes
Four Cores Each
High Throughput Computing

Parallel Sampler Conduit - Single-Core Model

Node 0 Core 0
Sample 0
Sample 8
Sample 1
Sample 9
Sample 2
Sample 10
Sample 3
Sample 11

Node 0 Core 1
Sample 1
Sample 9
Sample 2
Sample 10
Sample 3
Sample 11
Sample 4
Sample 12
Sample 5
Sample 13
Sample 6
Sample 14
Sample 7
Sample 15

Node 0 Core 2
Sample 2
Sample 10
Sample 3
Sample 11
Sample 4
Sample 12
Sample 5
Sample 13
Sample 6
Sample 14
Sample 7
Sample 15

Node 0 Core 3
Sample 3
Sample 11
Sample 4
Sample 12
Sample 5
Sample 13
Sample 6
Sample 14
Sample 7
Sample 15

Node 1 Core 0
Sample 4
Sample 12
Sample 5
Sample 13
Sample 6
Sample 14
Sample 7
Sample 15

Node 1 Core 1
Sample 5
Sample 13
Sample 6
Sample 14
Sample 7
Sample 15
Sample 8
Sample 9
Sample 10
Sample 11
Sample 12
Sample 13
Sample 14
Sample 15

Node 1 Core 2
Sample 6
Sample 14
Sample 7
Sample 15
Sample 8
Sample 9
Sample 10
Sample 11
Sample 12
Sample 13
Sample 14
Sample 15

Node 1 Core 3
Sample 7
Sample 15
Sample 8
Sample 9
Sample 10
Sample 11
Sample 12
Sample 13
Sample 14
Sample 15

Perfectly (Embarassingly) Parallel - Independent samples & No Communication Costs
High Throughput (Example)

Study the variability and uncertainties of Earthquake Ground Motion models.

Vp peak velocity of coherent pulse, cm/s  NS
Vp peak velocity of coherent pulse, cm/s  EW
Tp period of coherent pulse, s
Nc cycles in coherent pulse
Tpk time to the peak of the pulse
phi phase angle of the pulse
Vr peak velocity of incoherent ground motion, cm/s  NS
Vr peak velocity of incoherent ground motion, cm/s  EW
Tau1 envelope rise time, s
Tau2 constant time, s
Tau3 envelope decay time, s
power spectrum central frequency, Hz
power spectrum bandwidth factor,

Credit: http://people.duke.edu/~hpgavin/groundmotions/

Image: https://www.brgm.eu/project/earthquake-simulations-can-uncertainties-be-quantified-more-accurately
High Performance

E.g, Iterative Solvers.

<table>
<thead>
<tr>
<th>Node 0 Core 0</th>
<th>Rank 0</th>
<th>Node 1 Core 0</th>
<th>Rank 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 0 Core 1</td>
<td>Rank 1</td>
<td>Node 1 Core 1</td>
<td>Rank 5</td>
</tr>
<tr>
<td>Node 0 Core 2</td>
<td>Rank 2</td>
<td>Node 1 Core 2</td>
<td>Rank 6</td>
</tr>
<tr>
<td>Node 0 Core 3</td>
<td>Rank 3</td>
<td>Node 1 Core 3</td>
<td>Rank 7</td>
</tr>
</tbody>
</table>

Parallel Model (No Sampling)

Suffers from costs of communication.
High Performance (Example)

Large-Scale simulation of Earthquake at the San Andreas fault.

Mixed Approach

Parallel Sampler - Parallel Model

Sample 0
- Node 0 Core 0
  - Rank 0
  - Rank 1
  - Rank 2
  - Rank 3

Sample 1
- Node 1 Core 0
  - Rank 0
  - Rank 1
  - Rank 2
  - Rank 3

Pop Quiz: For what cases would you use parallel models & parallel sampling?

Time
Sampling Engines & Tasking Strategies
CMA-ES vs TMCMC

**CMA-ES**

All samples for the next generation are known at the end of the previous generation.

**TMCMC**

Samples for the current generation are determined in real-time, based on the evaluation of previous chain steps.
Task Distribution

How do we distribute samples to cores?
Divide-And-Conquer Strategy

Distribute samples equally (in number) among cores at the start of every generation.

Regular communication:
- Happens at the beginning of each generation.
- Message Sizes Well-known.
- Can use separate messages or a Broadcast

Only applicable when the entire workload is known from the beginning.
Load Imbalance

- Happens when cores receive uneven workloads.
- Represents a waste of computational power.

Total Running Time = $\text{Max(Core Time)}$

Load Imbalance Ratio = $\frac{\text{Max(Core Time)} - \text{Average(Core Time)}}{\text{Max(Core Time)}}$
Assign workload opportunistically, as cores/work become available.

Asynchronous Behavior:
- Producer sends samples to workers as soon as they become available.
- Workers report back finished sample and its result.
- Producer keeps a queue of available workers.

Does not require the entire knowing the workload in advance.
Load Imbalance

Total Running Time ≈ Mean(Core Time), as sample size and cores → Infinite

Lost Performance % = \frac{\#ProducerCores}{\#TotalCores}

Pop Quiz: What's the impact on large multi-core systems? (Euler = 24 cores)
Message Passing Model Review
**Message Passing Model**

**MPI:** *De facto* communication standard for high-performance scientific applications.

**Two-sided Communication:** A sender and a receive process explicitly participate in the exchange of a message.

A message encodes two pieces of information:
1. The actual message payload (data)
2. The fact that two ranks reached the exchange point (synchronization).

It does not encode **semantics:** the receiver needs to know what to do with the data.
One-sided Communication: A process can directly access a shared partition in another address space

**Message Passing Model**

- **MPI_Put()**
- **MPI_Get()**

It only encodes one piece of information: **data**.

Allows passing/receiving data without a corresponding send/recv request.

The other end is not notified of the operation (concurrency hazards)

Good for cases in which synchronization / ordering is not necessary.
Structured Grid Stencil Solver
- Iteratively approaches a solution.

2D Grid

Node

Traditional Decomposition
1 Process (Rank) per Core.

Ranks Exchange Halo (Boundary) Cells
Most HPC applications are programmed under the *Bulk-Synchronous Model.*

- Iterates among separate *computation* and *communication* phases.
A NOT so Good Case for MPI: Genome Assembly

Original DNA

- Construct a genome (chromosome) from a pool of short fragments produced by sequencers
- Analogy: shred many copies of a book, and reconstruct the book by examining the pieces
- Complications: shreds of other books may be intermixed, can also contain errors
- Chop the reads into fixed-length fragments (k-mers)
- K-mers form a De Bruijn graph, traverse the graph to construct longer sequences
- Graph is stored in a distributed hash table

Re-assembled DNA

Slide Credit: Scott B. Baden (Berkeley Lab)
Image Credit: http://people.mpi-inf.mpg.de/~sven/images/assembly.png
A NOT so Good Case for MPI: Genome Assembly

Build k-mer graphs from independent segments, sharing their hash numbers.

Initial Segment of DNA: ACTCGATGCTCAATG

GATG->ATGC
ACTC->CTCG->TCGA
TGTC->GCTC-CTCA-TCAA
Hash Table for Rank 1

TGCT->GCTC
TCAA->CAAT->AATG
Hash Table for Rank 0

Rank 0

GATG->ATGC
ACTC->CTCG->TCGA
TGTC->GCTC-CTCA-TCAA->CAAT->AATG
Hash Table for Rank 1

Rank 1

TGCT->GCTC
TCAA->CAAT->AATG
Hash Table for Rank 0

Detect new edge
Update Hash Table
Detect coinciding hash
Align K-mers

Completely Asynchronous:
- Detection of coincident hashes
- Asynchronous Hash Updates

Irregular Communication:
- K-mer chain size can vary
- Need to allocate hash entries in real time (cannot pre-allocate)

Difficult to implement on MPI due to its asynchronicity
Pop Quiz: MPI for Sampling Engines

Exam Question 1:
Is MPI a good model for the divide-and-conquer strategy?

Exam Question 2:
Is MPI a good model for the producer-consumer strategy?
Asynchronous Communication: UPC++
Get started with UPC++

Read our UPC++ getting started tutorial:

Download UPC++ Programmer's Guide, for reference:
https://bitbucket.org/berkeleylab/upcxx/wiki/Home

Run and analyze the full code of the following slides, download the codes:
- example0.cpp - Lambda Function
- example1.cpp - Hello World
- example2.cpp - Barriers
- example3.cpp - Blocking Broadcast
- example4.cpp - Non-blocking Broadcast
- example5.cpp - Single-Partition Global Memory Access
- example6.cpp - RPC with lambda functions
- example7.cpp - RPC with Quiescense (Fire-and-Forget)
- example8.cpp - Nested RPCs
- example9.cpp - Conjoining Futures
- example10.cpp - RPCs with Return Values
- example11.cpp - Global Memory with Distributed Objects
Before we start: C++ Lambda Expressions

**Also known as:** Anonymous Functions

**What they are:** Functions described as Expressions -> Convenient Notation

```cpp
double square(double x) { return x*x; } // Traditional Function

auto squareLambda = [](double x) -> double { return x*x; }; // Lambda Expression
```

```cpp
int main(int argc, char* argv[]) {
    const double scale = 2.0;
    auto squareLambda = [scale](double x) -> double { return scale*x*x; }; 
    for (double x = 0.0; x < 5.0; x += 1.0)
        printf("X = %f, %f*X^2 = %f\n", (double)x, scale, squareLambda(x));
}
```
#include <stdio.h>
#include <upcxx/upcxx.hpp>

int main(int argc, char* argv[])
{
    upcxx::init();
    rankId = upcxx::rank_me();
    rankCount = upcxx::rank_n();

    printf("Hello, I am rank %d of %d\n", rankId, rankCount);

    upcxx::finalize();
    return 0;
}
```cpp
int main(int argc, char* argv[]) {
    upcxx::init();
    int rankId = upcxx::rank_me();
    int rankCount = upcxx::rank_n();

    printf("Rank %d: Let's do this first.\n", rankId);

    upcxx::barrier();

    printf("Rank %d: Let's do this second.\n", rankId);

    upcxx::finalize();
    return 0;
}
```
```c++
#include <stdio.h>
#include <upcxx/upcxx.hpp>

int main(int argc, char* argv[]) {
    upcxx::init();
    int rankId = upcxx::rank_me();
    int rankCount = upcxx::rank_n();

    int number = 0;
    if (rankId == 0) number = 500;

    upcxx::broadcast(&number, 1 /*Count*/, 0 /*Root*/).wait();

    printf("Rank \%d: Number is \%d\n", rankId, number);
}
```
auto future = upcxx::broadcast(&number, 1 /*Count*/, 0 /*Root*/);

printf("Rank %d: (Before) Number is %d\n", rankId, number);

// Do other stuff

future.wait();

printf("Rank %d: (After) Number is %d\n", rankId, number);
Accessing the Global Address Space (I)

Broadcasting the global pointer to a single partition.

```cpp
upcxx::global_ptr<int> gptr;

if (rankId == 0) gptr = upcxx::new_array<int>(rankCount);

upcxx::broadcast(&gptr, 1, 0).wait();

auto future = upcxx::rput(&rankId, gptr + rankId, 1);

printf("Rank %d - Updating the global allocation.\n", rankId);

future.wait();

upcxx::barrier();

if (rankId == 0)
{
    int lptr = gptr.local();
    printf("{\n");
    for (int i = 0; i < rankCount; i++)
    {
        printf("%d,\n", lptr[i]);
    }
    printf("\n");
}
```

> upcxx-run -n 8 ./example5

```
Rank 0 - Updating the global allocation.
Rank 1 - Updating the global allocation.
Rank 2 - Updating the global allocation.
Rank 3 - Updating the global allocation.
Rank 4 - Updating the global allocation.
Rank 5 - Updating the global allocation.
Rank 6 - Updating the global allocation.
Rank 7 - Updating the global allocation.
{0,1,2,3,4,5,6,7,}
```
Accessing the Global Address Space (II)

Accessing multiple partitions via a distributed object.

```cpp
auto myPartition = upcxx::new_<double>(rankId);
upcxx::dist_object<upcxx::global_ptr<double>> partitions(myPartition);

*myPartition.local() = sqrt((double)rankId);

upcxx::barrier();

if (rankId == 0) for (int i = 0; i < rankCount; i++)
{
    auto partition = partitions.fetch(i).wait();
    double val = upcxx::rget(partition).wait();
    printf("SquareRoot(%f) = %f\n", (double)i, val);
}
```

```
upcxx-run -n 8 ./example11
SquareRoot(0.000000) = 0.000000
SquareRoot(1.000000) = 1.000000
SquareRoot(2.000000) = 1.414214
SquareRoot(3.000000) = 1.732051
SquareRoot(4.000000) = 2.000000
SquareRoot(5.000000) = 2.236068
SquareRoot(6.000000) = 2.449490
SquareRoot(7.000000) = 2.645751
```
Asynchronous Execution with RPCs

Remote Procedure Call - Using Lambda Expressions.

```cpp
upcxx::init();
rankId = upcxx::rank_me();
int rankCount = upcxx::rank_n();

finished = false;
upcxx::barrier();

if (rankId == 0)
    for (int i = 1; i < rankCount; i++)
        upcxx::rpc(i, [](int par){
            printf("Rank %d: Received RPC with Parameter: %d\n", rankId, par);
            finished = true;
        }, i*rankCount).wait();

if (rankId > 0) while (!finished) upcxx::progress();
```
Conjoining Futures

So that we can continue doing stuff while they finish

```cpp
upcxx::future<> futs = upcxx::make_future();

if (rankId == 0)
    for (int i = 1; i < rankCount; i++)
    {
        auto fut = upcxx::rpc(i, [](int par){
            printf("Rank %d: Received RPC with Parameter: %d\n", rankId, par);
            finished = true;
        }, i*rankCount);
        futs = upcxx::when_all(futs, fut);
    }

if (rankId > 0) while (!finished) upcxx::progress();

if (rankId == 0) printf("Not all finished yet.\n");
```

```
if (rankId == 0) futs.wait();
if (rankId == 0) printf("All finished now.\n");
```

> upcxx-run -n 8 ./example9
Rank 1: Received RPC with Parameter: 8
Rank 2: Received RPC with Parameter: 16
Rank 3: Received RPC with Parameter: 24
Rank 4: Received RPC with Parameter: 32
Rank 5: Received RPC with Parameter: 40
Rank 6: Received RPC with Parameter: 48
Rank 7: Received RPC with Parameter: 56

All finished now.
```
if (rankId == 0)
{"futs = upcxx::make_future();
for (int i = 1; i < rankCount; i++)
{"auto f1 = upcxx::rpc(i, [](int par){
    printf("Rank %d: Received RPC with Parameter: %d\n", rankId, par);
    finished = true;
}, i*rankCount);
    f1.then([i]() {printf("Rank 0: Rank %d Came back.\n", i);});
    futs = upcxx::when_all(futs, f1);
}futs.wait();\}
RPC with Return Values

```c
int calculateSquare(int x) { return x*x; }

int main(int argc, char* argv[])
{
  upcxx::init();
  int rankId    = upcxx::rank_me();
  int rankCount = upcxx::rank_n();

  if (rankId == 0)
    for (int i = 1; i < rankCount; i++)
      printf("Value: %d - Square %d\n", i, upcxx::rpc(i, calculateSquare, i).wait());
```

```
>upcxx-run -n 8 ./example10
Value: 1 - Square 1
Value: 2 - Square 4
Value: 3 - Square 9
Value: 4 - Square 16
Value: 5 - Square 25
Value: 6 - Square 36
Value: 7 - Square 49
```
Quiescense (Fire and Forget)

When we don't need a response

```c
int main(int argc, char* argv[])
{
    upcxx::init();
    rankId    = upcxx::rank_me();
    int rankCount = upcxx::rank_n();

    finished = false;
    upcxx::barrier();

    if (rankId == 0)
        for (int i = 1; i < rankCount; i++)
            upcxx::rpc_ff(i, [](int par){
                printf("Rank %d: Received RPC with Parameter: %d\n", rankId, par);
                finished = true;
            }, i*rankCount);

    if (rankId > 0) while (!finished) upcxx::progress();
}
```

When we don't need a response, we can use `upcxx::rpc_ff` to send a request without expecting a response. This is a form of quiescence called Fire and Forget.
Nested RPCs

When the receiver also needs the sender to do something

```cpp
tuple::barrier();

if (rankId == 0)
    for (int i = 1; i < rankCount; i++)
        tuple::rpc_ff(i, [] (int par) {
            printf("Rank %d: Received RPC with Parameter: %d\n", rankId, par);
            finished = true;
        });

responses++;

printf("Rank 0: Received responses: %d\n", responses);
if (responses == rankCount-1) finished = true;
});
}

i*rankCount);
```