Parallel Computing Approaches for Model Comparison

Tutorial

Wesley Henderson

4 July 2018

MaxEnt 2018, The Alan Turing Institute

University of Mississippi
Department of Electrical Engineering

Outline

Introduction

Overview

- Model comparison is one level of Bayesian inference
- Model comparison algorithms can take advantage of parallel computing
- Many parallel programming models and hardware platforms exist
- Some problems can be better suited for certain software and hardware combinations
- Tutorial will cover three algorithm/programming model/hardware combinations. Starting place for exploring other combinations.

$$\underbrace{p(\boldsymbol{\Theta}_i|M_i,\boldsymbol{D},I)}_{\text{Posterior}} = \frac{\overbrace{p(\boldsymbol{\Theta}_i|M_i,I)}^{\text{Prior}}\,\overbrace{p(\boldsymbol{D}|M_i,\boldsymbol{\Theta}_i,I)}^{\text{Likelihood}}}{\underbrace{p(\boldsymbol{D}|M_i,I)}_{\text{Evidence or model likelihood}}} \tag{1}$$

$$\overbrace{p(M_i|\boldsymbol{D},I)}^{\text{Model posterior}} = \frac{\overbrace{p(M_i|I)}^{\text{Model prior}}\,\overbrace{p(\boldsymbol{D}|M_i,I)}^{\text{Evidence}}}{p(\boldsymbol{D}|I)} \tag{2}$$

$$O_{ji} = \frac{p(M_j|\boldsymbol{D}, I)}{p(M_i|\boldsymbol{D}, I)} = \underbrace{\frac{p(M_j|I)}{p(M_i|I)}}_{\text{Prior odds}} \times \underbrace{\frac{p(\boldsymbol{D}|M_j, I)}{p(\boldsymbol{D}|M_i, I)}}_{\text{Evidence ratio}} \tag{3}$$

If model priors are equal,

$$O_{ji} = \frac{p(\boldsymbol{D}|M_j, I)}{p(\boldsymbol{D}|M_i, I)} \tag{4}$$

$$p(\boldsymbol{D}|M_i, I) = \int_{\tilde{\boldsymbol{\Theta}}} p(\boldsymbol{\Theta}_i|M_i, I) p(\boldsymbol{D}|M_i, \boldsymbol{\Theta}_i, I) \,\mathrm{d}\boldsymbol{\Theta} \tag{5}$$

Main idea

Break task into pieces that can be worked on concurrently

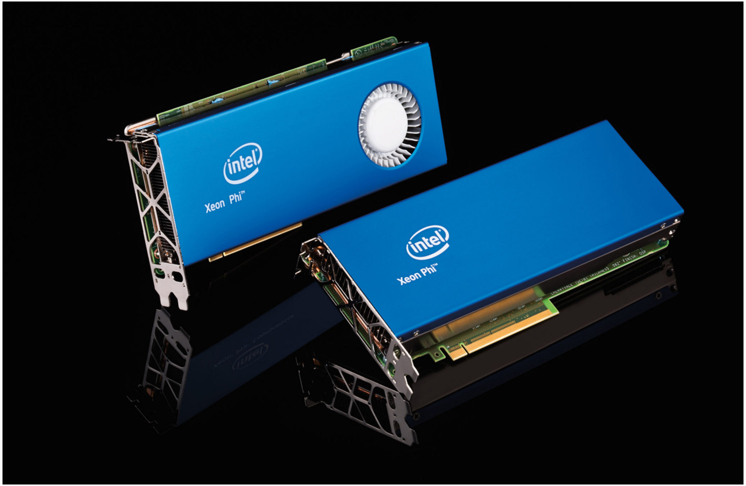
Why?

- More efficient use of resources
- Get results more quickly
- Solve massive problems that are intractable otherwise

Programming models (a.k.a., useful abstractions)

- Shared memory
- Distributed memory
- Hybrid
- Single program, multiple data (SPMD)
- Multiple program, multiple data (MPMD)

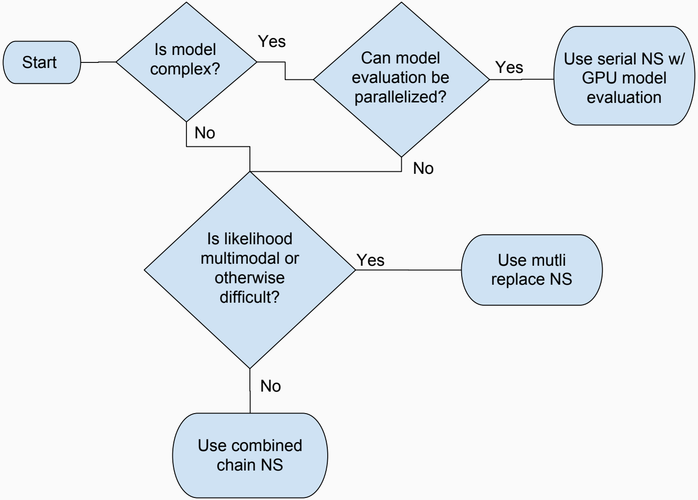
SGI cluster at MCSR



Intel Xeon Phi

Nvidia Tesla GPU Accelerator

Introduction: Choosing a method



Algorithms

Algorithms

- Nested sampling
  - Combined chains
  - Multiple replacement
- Thermodynamic integration
- Reversible jump MCMC
- Sequential Monte Carlo

Algorithms

- Nested sampling
    - Combined chains
    - Multiple replacement
- Thermodynamic integration
- Reversible jump MCMC
- Sequential Monte Carlo

Nested sampling reparameterizes the model evidence integral

$$\mathcal{Z} = \int \pi(\boldsymbol{\Theta}) L(\boldsymbol{\Theta}) \, \mathrm{d}\boldsymbol{\Theta} \tag{6}$$

$$L(\boldsymbol{\Theta}) = \int_0^{L(\boldsymbol{\Theta})} \mathrm{d}\mathcal{L} \tag{7}$$

$$\mathcal{Z} = \int \pi(\boldsymbol{\Theta}) \left[\int_0^{L(\boldsymbol{\Theta})} \mathrm{d}\mathcal{L} \right] \mathrm{d}\boldsymbol{\Theta} \tag{8}$$

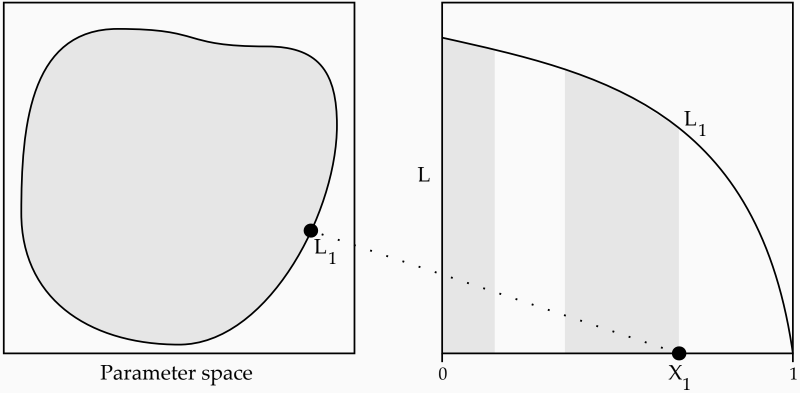
$$\mathcal{Z} = \int_0^\infty \left[\int_{\{\boldsymbol{\Theta}: L(\boldsymbol{\Theta}) > \mathcal{L}\}} \pi(\boldsymbol{\Theta}) \, \mathrm{d}\boldsymbol{\Theta} \right] \mathrm{d}\mathcal{L} \tag{9}$$

$$X(\mathcal{L}) = \int_{\{\boldsymbol{\Theta}: L(\boldsymbol{\Theta}) > \mathcal{L}\}} \pi(\boldsymbol{\Theta}) \, \mathrm{d}\boldsymbol{\Theta} \tag{10}$$

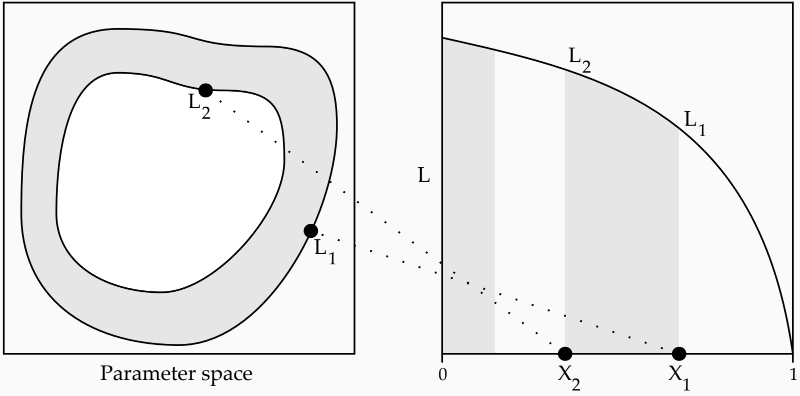
$$\mathcal{Z} = \int_0^\infty X(\mathcal{L}) \, \mathrm{d}\mathcal{L} \tag{11}$$

$$\mathcal{Z} = \int_0^1 \mathcal{L}(X) \, \mathrm{d}X. \tag{12}$$

Nested sampling basics: Prior mass and likelihood

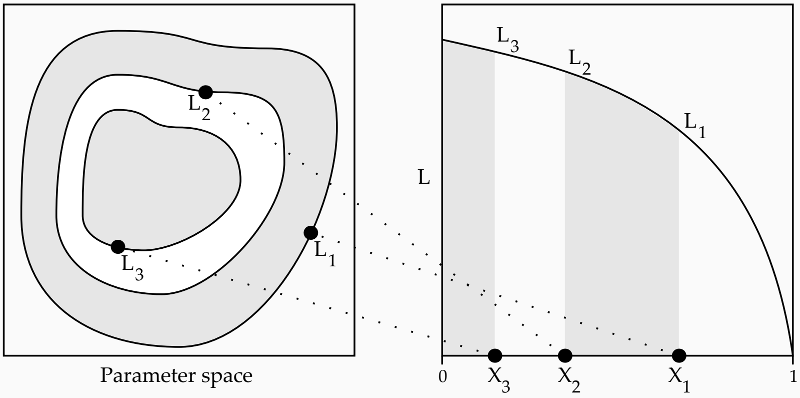


(From Skilling '06)



(From Skilling '06)

Nested sampling basics: Prior mass and likelihood

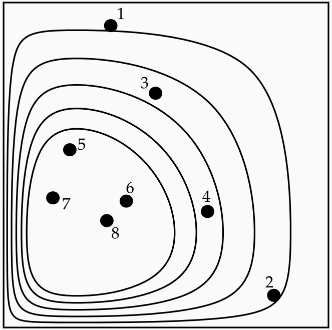


(From Skilling '06)

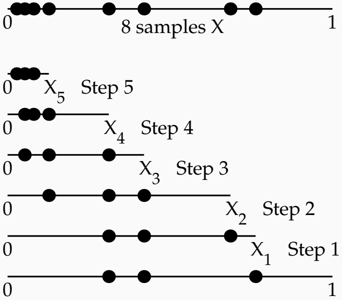
Nested sampling basics: Procedure

1. Generate N live samples from the unconstrained prior
2. Set $Z = 0$ and $X_0 = 1$
3. Record the lowest likelihood among the N live samples as L_i
4. Estimate X_i corresponding to L_i
5. Set $w_i = X_{i-1} - X_i$
6. Increment Z by $L_i w_i$
7. (Optional) Save least-likelihood live sample in list of "dead" samples
8. Replace the least-likelihood live sample with one sampled from the prior, constrained by L_i
9. If halting condition is not met, go to 3

Nested sampling basics: Procedure



Parameter space



Enclosed prior mass X

Nested sampling basics: Estimating X_i

The likelihood L_i at a given parameter vector $\boldsymbol{\Theta}_i$ can be computed exactly, but the associated prior mass X_i cannot

At every step of nested sampling, the live samples' prior mass are distributed as $U[0, X(L)]$

Order statistics of the uniform distribution show

$$t_i = \frac{X_i}{X_{i-1}} \sim \mathrm{Beta}(N, 1) \tag{13}$$

$$X_i = \prod_{k=1}^{i} t_k \tag{14}$$

Nested sampling basics: Estimating X_i

$$\mathrm{E}(\log t_i) = -1/N \tag{15}$$

$$X_i \approx \exp(-i/N) \tag{16}$$

$$Z \approx \sum_{i=1}^{m} (X_{i-1} - X_i)\, L_i \tag{17}$$

Combined chain nested sampling Combine multiple independent serial nested sampling results

Multiple replacement nested sampling Discard and replace multiple samples at each likelihood threshold

Parallel model evaluation Use serial nested sampling, but parallelize the model computation

First parallel approach

- Run M independent nested sampling processes, each using N live samples
- Resulting discarded samples can be combined and sorted by likelihood; prior mass estimate is same as in case of one nested sampling process with $M \times N$ live samples
- Proofs of the correctness of this approach are available in our 2017 paper.
- Theoretical speed up of $\mathcal{O}(M)$ over serial method
- MIMD

Second parallel approach

- Instead of discarding and replacing one sample, discard and replace R samples at each likelihood constraint
- Initially proposed by Burkoff, et al., in 2012
- Our 2014 paper found that in order to maintain the same level of precision in evidence estimate, $\sqrt{R}N$ live samples must be used when R samples are discarded and replaced for each likelihood constraint
- Theoretical speed up of $\mathcal{O}(\sqrt{R})$ over serial method
- MIMD

Third parallel approach

- Use serial nested sampling
- Parallelize model evaluation
- Ideal for cases with complex model equations
- Allows use of GPU for computation
- SIMD

Shared memory approach

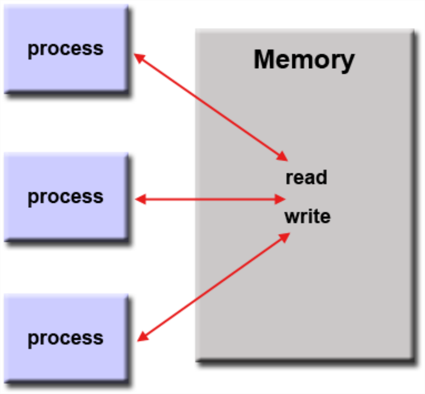
Shared Memory: Overview

- Multiple processors share same memory
- Examples
  - Multi-core CPUs
  - Intel Xeon Phi
- Advantages
  - Processes can easily share data
  - Relatively easy to modify programs to use
- Disadvantages
  - Without proper safeguards, data can be corrupted by competing processes
  - Scalability

Shared Memory: Overview

No threads

- Multiple processes used
- Very straightforward, but no standard approach across platforms



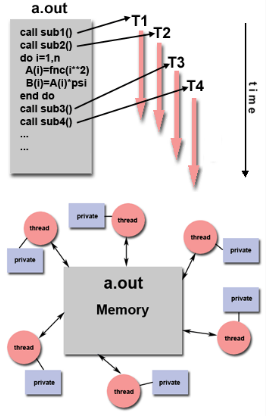
(From `https://computing.llnl.gov/tutorials/parallel_comp/`)

Shared Memory: Overview

With threads

pthreads Library based. Requires manual creation, starting, and synchronization of threads

OpenMP Compiler directive based. Manual controls available, but simple complier directives can be used to parallelize some serial code.



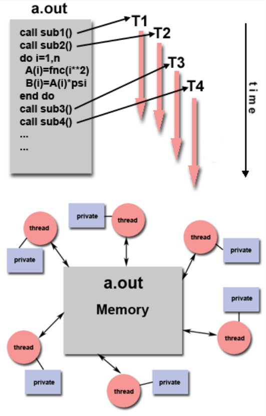
(From `https://computing.llnl.gov/tutorials/parallel_comp/`)

Shared Memory: Overview

With threads

pthreads Library based. Requires manual creation, starting, and synchronization of threads

OpenMP Compiler directive based. Manual controls available, but simple complier directives can be used to parallelize some serial code.



(From `https://computing.llnl.gov/tutorials/parallel_comp/`)

Shared Memory: Xeon Phi

Two existing versions for general applications

Knight's Corner (x100)

- PCI express card, now discontinued
- Runs only on certain motherboards
- 4-way simultaneous multithreading per core
- 57–61 x86-64 cores
- 512-bit SIMD units
- 512 KB L2 cache per core
- Supports offloading or native execution

Knight's Landing (x200)

- Bootable, mainboard chip
- Mostly available from workstation system builders. Expensive
- 64–72 Intel Atom cores
- Two 512-bit vector units per core
- AVX-512 SIMD instructions
- Native execution only

Combined-chain NS on Xeon Phi

- In combined-chain, most of the code can be made trivially parallel, so works well natively on the Phi
- This example code is in C++, but mostly uses C idioms
- Xeon Phi requires Intel C++ compiler. Free for students and in other limited cases

Sample struct and static arrays

```
typedef struct
{
    __declspec(align(64)) double theta[NUM_PARAMS];
    __declspec(align(64)) double logL;
    __declspec(align(64)) double logWt;
} sample;
```

Dynamic arrays (Intel C++ compiler only)

```
#include <malloc.h>
sample ** samples = (sample**)_mm_malloc(SIZE * sizeof(sample), 64);
// Do some things
_mm_free(samples);
```

Start and collect results from each NS instance

```
#pragma omp parallel for
#pragma vector aligned
for (int i = 0; i < N; i++)
{
    nested_sampling(data, samples[i],
        &total_samples[i], i);
}
```

Collecting result samples

```
int q = 0;
for (int i = 0; i < NUM_CHAINS; i++) {
    for (int j = 0; j < total_samples[i]; j++) {
        for (int k = 0; k < NUM_PARAMS; k++) {
            samples[q].theta[k] = sample_list_2d[i][j].theta[k];
        }
        samples[q].logL = sample_list_2d[i][j].logL;
        samples[q].logWt = 0.0;
        q++;
    }
}
```

Sorting samples and computing evidence

```
qsort(samples, *sample_count, sizeof(sample), comparator);
double mean_log_t = 1.0 / (NUM_CHAINS * NUM_LIVE_SAMPLES);
double logwidth = log(1.0 - exp(-mean_log_t));
*logZ = -1.0 * std::numeric_limits<double>::max();
*H = 0.0;
for (int i = 0; i < *sample_count; i++) {
    samples[i].logWt = logwidth + samples[i].logL;
    logZnew = log_plus(*logZ, samples[i].logWt);
    *H = exp(samples[i].logWt - logZnew) * samples[i].logL +
        exp(*logZ - logZnew) * (*H + *logZ) - logZnew;
    *logZ = logZnew;
    logwidth -= mean_log_t;
}
```

Sample comparator function

```
int comparator(const void * lhs, const void * rhs)
{
    double diff;
    diff = ((sample *)lhs)->logL - ((sample *)rhs)->logL;
    if (diff > 0)
        return 1;
    else if (diff < 0)
        return -1;
    else
        return 0;
}
```

SIMD block of likelihood function

```
#pragma omp simd
#pragma vector aligned
for (int i = 0; i < NUM_DATA; i++)
{
#pragma vector aligned
    for (int j = 0; j < NUM_ATOMS; j++)
    {
        mock[i] += A[j] * cos(2 * PI * f[j] * time[i]) +
            B[j] * sin(2 * PI * f[j] * time[i]);
    }
    sq_error[i] = (mock[i] - data[i]) * (mock[i] - data[i]);
}
```

Other implementation considerations

- `main` function loads data and sets nested sampling parameters, then calls `manager` function
- `manager` sets up temporary arrays for collecting results from each nested sampling function, then calls `nested_sampling` within OpenMP
- Make sure that any temporary arrays used (in explore function, likelihood function, etc.) are declared with the correct alignment

Shared Memory: Multi-core CPU

- Common in consumer-grade desktops, laptops, and smartphones
- One processor package, two or more independent processing units
- For development, I used an Intel Xeon E5-1603 v3
  - 2.80 GHz
  - 4 cores
  - 10 MB SmartCache

Multiple replacement NS on CPU

- More serial-only portions of this code, so CPU is perhaps better fit
- This example code is idiomatic C++

Model class

```
class Model
{
    public:
        Model(std::vector<double> data_in);
        double compute_log_likelihood(std::vector<double> theta);
        void compute_sq_error(double * A, double * B, double * f,
                double * sq_error);
        void set_data(std::vector<double> data_in);
        std::vector<double> get_data();
    private:
        std::vector<double> data;
};
```

```
class Sample
{
    public:
        Sample(Model model);
        void set_theta(std::vector<double> theta_in);
        std::vector<double> get_theta();
        void set_logL(double logL_in);
        double get_logL();
        void set_logWt(double logWt_in);
        double get_logWt();
    private:
        std::vector<double> theta;
        double logL;
        double logWt;
};
```

Multiple replacement NS on CPU

Within the `nested_sampling` funcion:

```
std::vector<Sample> live_samples;
for (int i = 0; i < NUM_LIVE_SAMPLES; i++)
{
    Sample live_sample(model);
    live_samples.push_back(live_sample);
}
```

Sort at each likelihood threshold

```
std::sort(live_samples.begin(), live_samples.end(), sample_comp);
```

Comparator function

```
double sample_comp(Sample a, Sample b)
{
    return (a.get_logL() < b.get_logL());
}
```

Choose surviving samples to evolve

```
std::vector<int> surviving_idxs(NUM_LIVE_SAMPLES - NUM_REP);
for (int i = NUM_REP; i < NUM_LIVE_SAMPLES; i++)
{
    surviving_idxs[i - NUM_REP] = i;
}
std::random_shuffle(surviving_idxs.begin(), surviving_idxs.end());
std::vector<int> copy_idxs(NUM_REP);
for (int i = 0; i < NUM_REP; i++)
{
    copy_idxs[i] = surviving_idxs[i];
}
```

Evolve each sample simultaneously

```
std::vector<std::vector<double> > theta_in_vec(NUM_REP);
std::vector<std::vector<double> > theta_out_vec(NUM_REP);
for (int i = 0; i < NUM_REP; i++)
{
    theta_in_vec[i] = live_samples[copy_idxs[i]].get_theta();
}
#pragma omp parallel for
for (int i = 0; i < NUM_REP; i++)
{
    mcmc_explore(theta_in_vec[i], logLstar, model, live_samples,
        theta_out_vec[i]);
}
```

GPU approach

- Main idea: use hardware originally meant for graphics computations for general purpose computation
- Relative to CPUs, GPUs have many more, slower cores
- Parallel portion of program needs to be SIMD for maximum performance
- Several languages/libraries exist for running code on GPUs
    - Khronos Group’s OpenCL
    - Nivida’s CUDA
    - Apple’s Metal
- We’ll focus on OpenCL

OpenCL

- OpenCL code runs on many device types (AMD and Nvidia GPUs, CPUs, even the Xeon Phi)
- Kernels contain code that is run by each work item
- Work items are grouped into work groups
- Each work item operates on a different piece of data stream
- Branches within kernel are inefficient
- On Nvidia GPUs, CUDA can perform better for some code, but 1-to-1 comparison is difficult

Parallel model evaluation on GPU

- Nested sampling portion of code is straightforward
- Complexity arises in problem-specific code
- Example code is idiomatic C++ and uses OpenCL
- There is a lot more work involved in adapting code for OpenCL, compared with the previous examples

Model class

```
class Model
{
    public:
        Model(std::vector<double> data_in, cl::Buffer buffer_A,
             cl::Buffer buffer_B, cl::Buffer buffer_f,
             cl::Buffer buffer_data, cl::Buffer buffer_sq_error,
             cl::CommandQueue queue, cl::Context context,
             cl::Kernel kernel);
        double compute_log_likelihood(std::vector<double> theta);
        void set_data(std::vector<double> data_in);
        std::vector<double> get_data();
```

Model class, continued

```
    private:
        std::vector<double> data;
        int num_data;
        cl::Buffer buffer_A;
        cl::Buffer buffer_B;
        cl::Buffer buffer_f;
        cl::Buffer buffer_data;
        cl::Buffer buffer_sq_error;
        cl::CommandQueue queue;
        cl::Context context;
        cl::Kernel kernel_compute_sq_error;
};
```

Setup

```
std::vector<cl::Platform> all_platforms;
cl::Platform::get(&all_platforms);
cl::Platform default_platform = all_platforms[0];
std::vector<cl::Device> all_devices;
default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
cl::Device default_device = all_devices[0];
cl::Context context({ default_device });
cl::Program::Sources sources;
```

Kernel

```
std::string kernel_code =
  "kernel void compute_sq_error("
    "global const double * A, global const double * B,"
    "global const double * f, global const double * data,"
    "const int num_data, const int num_atoms,"
    "global double * sq_error)"
  "{"
  "    double pi = 3.14159265359;"
  "    int i = get_global_id(0);"
  "    double mock = 0.0;"
  "    sq_error[i] = 0.0;"
  "    double time = data[num_data + i];"
```

Kernel, continued

```
"    for (int j = 0; j < num_atoms; j++)"
"    {"
"        mock += A[j] * cos(2 * pi * f[j] * time) +"
"          B[j] * sin(2 * pi * f[j] * time);"
"    }"
"    sq_error[i] = (mock - data[i]) * (mock - data[i]);"
"}";
```

Final setup

```
sources.push_back(
    std::make_pair(kernel_code.c_str(), kernel_code.length()));
cl::Program program(context, sources);
program.build(all_devices);
cl::Buffer buffer_A(context, CL_MEM_READ_WRITE, sizeof(double) * NA);
cl::Buffer buffer_B(context, CL_MEM_READ_WRITE, sizeof(double) * NA);
cl::Buffer buffer_f(context, CL_MEM_READ_WRITE, sizeof(double) * NA);
cl::Buffer buffer_data(context, CL_MEM_READ_WRITE,
    sizeof(double) * (2 * num_data + 1));
cl::Buffer buffer_sq_error(context, CL_MEM_READ_WRITE,
    sizeof(double) * num_data);
```

Final setup, continued

```
cl::CommandQueue queue(context, default_device);
queue.enqueueWriteBuffer(buffer_data, CL_TRUE, 0,
    sizeof(double) * (2 * num_data + 1), data.data());
cl::Kernel kernel_compute_sq_error = cl::Kernel(program,
    "compute_sq_error");
Model model(data, buffer_A, buffer_B, buffer_f, buffer_data,
    buffer_sq_error, queue, context, kernel_compute_sq_error);
```

Log-likelihood function

```
double Model::compute_log_likelihood(std::vector<double> theta)
{
    // Scale the parameters before sending them to the device
    double A[NUM_ATOMS];
    double B[NUM_ATOMS];
    double f[NUM_ATOMS];
    const int param_per_atom = NUM_PARAMS / NUM_ATOMS;
    for (int i = 0; i < NUM_ATOMS; i++)
    {
        A[i] = (AMAX - AMIN) * theta[i * param_per_atom + 0] + AMIN;
        B[i] = (AMAX - AMIN) * theta[i * param_per_atom + 1] + AMIN;
        f[i] = (FMAX - FMIN) * theta[i * param_per_atom + 2] + FMIN;
    }
```

Log-likelihood function, continued

```
queue.enqueueWriteBuffer(buffer_A, CL_TRUE, 0,
    sizeof(double) * NUM_ATOMS, A);
queue.enqueueWriteBuffer(buffer_B, CL_TRUE, 0,
    sizeof(double) * NUM_ATOMS, B);
queue.enqueueWriteBuffer(buffer_f, CL_TRUE, 0,
    sizeof(double) * NUM_ATOMS, f);
kernel_compute_sq_error.setArg(0, buffer_A);
kernel_compute_sq_error.setArg(1, buffer_B);
kernel_compute_sq_error.setArg(2, buffer_f);
kernel_compute_sq_error.setArg(3, buffer_data);
kernel_compute_sq_error.setArg(4, num_data);
kernel_compute_sq_error.setArg(5, NUM_ATOMS);
kernel_compute_sq_error.setArg(6, buffer_sq_error);
```

Log-likelihood function, continued

```
queue.enqueueNDRangeKernel(kernel_compute_sq_error,
    cl::NullRange, cl::NDRange(num_data), cl::NullRange);
queue.finish();
double sq_error[num_data];
queue.enqueueReadBuffer(buffer_sq_error, CL_TRUE, 0,
    num_data * sizeof(double), sq_error);
```

Log-likelihood function, continued

```
    double q2;
    double logL;
    const double sigma = 0.5;
    q2 = 0.0;
    for (int i = 0; i < num_data; i++)
    {
        q2 += sq_error[i];
    }
    logL = -1 * q2 / (2 * sigma * sigma);
    return logL;
}
```

Further suggestions and conclusion

Suggestion: Cloud computing

If your access to parallel computing hardware is insufficient, consider cloud computing

- Google Cloud Platform
- Amazon Elastic Cloud Compute (EC2)
- Microsoft Azure

Each platform has a free trial with limited access to HPC resources

Example: NVIDIA Tesla K80, in Europe, costs $0.49 per hour per die

Common features

- Create VMs with a variety of resource configurations
- Connect using SSH to administer and run code
- Create instances with GPUs
- Create clusters
- Only pay for what you use

If you want to write HPC code on a minimal system like a Chromebook, Amazon has a somewhat minimal cloud IDE, Cloud9

Conclusion

- Model comparison comprises an important class of inference problems
- Existing techniques, e.g., nested sampling, can be parallelized in a variety of ways using a variety of hardware
- It is critically important to match algorithm to hardware
- Examples have been shown that address implementation on the Xeon Phi, CPUs, and GPUs
- Cloud computing platforms can provide an attractive alternative to dedicated hardware

Resources

- Lawrence Livermore National Lab's "Introduction to Parallel Computing" `https://computing.llnl.gov/tutorials/parallel_comp/`
- Best practice guide for older Xeon Phi models (Knight's Corner): `http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/`
- Best practice guide for newer Xeon Phi models (Knight's Landing): `http://www.prace-ri.eu/best-practice-guide-knights-landing-january-2017/`
- Introduction to OpenCL: `http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854`