Parallelization Strategies: Hardware and Software
(Two Decades of Personal Experience)

A. Ronald Gallant
Penn State University

Conference on ”Massively Parallel Computing in Economics,” Econometric Institute, Erasmus University, Rotterdam, and Econometrics Department, VU University, Amsterdam, May 11, 2012. Updated May 5, 2014.

These slides:
Which are excerpts from these lectures:
http://www.aronaldg.org/courses/compecon/
Preamble

• The object oriented programming style minimizes a developer’s time by both allowing faster coding, minimizing errors, and reducing collateral damage during maintenance.

• It follows that the better parallel strategies are those that do not impede the object oriented programming style, all else being equal.

• Throughout this talk, compatibility with the object oriented programming style is a desideratum.
Object Oriented Programming

Object oriented programming is a style of programming developed to support modern computing projects. Much of the development was in the commercial sector. The features of interest to us are the following:

- The computer code closely resembles the way we think about a problem.
- The computer code is compartmentalized into objects that perform clearly specified tasks. Importantly, this allows one to work on one part of the code without having to remember how the other parts work: selective ignorance.
- One can use inheritance and virtual functions both to describe the project design as interfaces to its objects and to permit polymorphism. Interfaces cause the compiler to enforce our design, relieving us of the chore. Polymorphism allows us to easily swap in and out objects so we can try different models, different algorithms, etc.
- The structures used to implement objects are much more flexible than the minimalist types of non-object oriented language structures such as subroutines, functions, and static common storage.
Parallel Computing: Overview

- Clusters.
  - Memory is not shared among all CPUs.
  - MPI (Message Passing Interface) can be used.
    - Most common coding strategy is master/slave (aka. administrator/worker or leader/team) branches within a single program.

- Symmetric Multi-Processor (SMP) machines.
  - Memory is shared among all CPUs; cores count as CPUs.
  - MPI can be used.
  - Threads can be used
  - OpenMP can be used.

- Graphics devices.
  - A graphics device is a massively parallel SMP machine.
  - Uses threads that are automatically launched by the device.
  - OpenCL can be used.
    - Nvidia’s CUDA is not portable; becoming obsolete
Typical small cluster configuration
Typical small cluster wiring
A serious SMP machine

Penguin Altus 1804i

Four AMD Opteron 6176, 12 cores, 2.3GHz, 48 cores in total
A serious cluster

NCAR's bluefire

IBM Power 575: 128 nodes, 32 CPUs per node

Each CPU is 4.7GHz, 4096 CPUs in total
Graphics Devices

Tesla C1060 Computing Processor

NVIDIA® Tesla™ C1060 Computing Processor enables the transition to energy efficient parallel computing power by bringing the performance of a small cluster to a workstation.

Now Available!
- Buy now online or from a system builder
- Buy Tesla Personal Supercomputer

Print page

<table>
<thead>
<tr>
<th>Form Factor</th>
<th>10.5 x 4.39 in, Dual Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Tesla GPUs</td>
<td>1</td>
</tr>
<tr>
<td># of Streaming Processor Cores</td>
<td>240</td>
</tr>
<tr>
<td>Frequency of processor cores</td>
<td>1.3 GHz</td>
</tr>
<tr>
<td>Single Precision floating point performance (gops)</td>
<td>933</td>
</tr>
<tr>
<td>Double Precision floating point performance (gops)</td>
<td>78</td>
</tr>
<tr>
<td>Floating Point Precision</td>
<td>IEEE 754 single &amp; double</td>
</tr>
<tr>
<td>Total Unified Memory</td>
<td>4 GB GDDR5</td>
</tr>
<tr>
<td>Memory Speed</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>512-bit</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>128 GB/sec</td>
</tr>
<tr>
<td>Max Power Consumption</td>
<td>187.5 W</td>
</tr>
<tr>
<td>System Interface</td>
<td>PCIe x16</td>
</tr>
<tr>
<td>Auxiliary Power Connectors</td>
<td>6-pin &amp; 8-pin</td>
</tr>
<tr>
<td>Thermal Solution</td>
<td>Active Fan/shroud</td>
</tr>
<tr>
<td>Software Development Tools</td>
<td>C-host CUDA Toolkit</td>
</tr>
</tbody>
</table>
Combinations

- The machines that make up a cluster are usually SMP machines.
  - Newer machines have multiple CPUs with multiple cores.
  - MPI can distribute across all cores in a cluster.
- These SMP machines can, in turn, have graphics devices installed.
- Enables mixtures of coding strategies
  - MPI to distribute across clusters
  - Threads or OpenMP within an SMP machine
  - OpenCL within threads
Coding Strategies

• *Shell Scripts.* Some programs, such as nonlinear optimizers that use multiple, random starts, are so embarrassingly parallelizable, that parallelization can be done with shell scripts alone.

• *Message Passing Interface (MPI).* The industry-standard protocol for implementing parallel processing. PVM is similar. Allows communication among processes running on different processors. Architecture independent: Code written for a cluster will run on multiple-processor, shared-memory machines. Mildly disruptive to serial code logic.

  ▶ http://www.mpi-forum.org  MPI reference
  ▶ http://www.open-mpi.org  software
  ▶ http://ladon.iqfr.csic.es/docs/MPI Ug_in_FORTRAN.pdf  Fortran
  ▶ ftp://math.usfca.edu/pub/MPI/mpi.guide.ps  C & C++

• *POSIX Threads (Pthreads).* Allows functions with the same name but different instances of the same argument to be run simultaneously. All functions have full access to memory and other machine resources. Can be disruptive to serial code logic and may require care to avoid simultaneous use of the same memory locations or other resources.

  ▶ http://www.llnl.gov/computing/tutorials/pthreads
Coding Strategies (Continued)

• **Parallelized Libraries.** Allows sequential code to have some of the benefits of parallelism. Works best on SMP machines. Can actually impede performance if coupled with MPI.
  ▶ http://www.nag.co.uk/numberic/fd/FDdescription.asp

• **High Performance Fortran.** A sort of hybrid of the strategies above, allows both threads and message passing. Worked poorly for us.
  ▶ http://hpff.rice.edu

• **Open Multi-Processing (OpenMP).** Implements multiprocessing programming in C/C++ and Fortran on SMP machines. It is a set of compiler directives, library routines, and environment variables that influence runtime behavior. Least disruptive to existing serial code. Similar to threads; easier to code. Most compilers have it.
  ▶ https://computing.llnl.gov/tutorials/openMP
Coding Strategies (Continued)

- **Open Computing Language (OpenCL).** A language for programming GPU devices. Can be seriously disruptive to serial logic; especially when it forces dependencies among objects that would otherwise be independent. CUDA is similar and simpler but only works for Nvidia cards.
  
  ▶ http://www.khronos.org

- **ViennaCL.** A scientific computing library that encapsulates OpenCL in the style of the C++ Standard Template Library. Hides all OpenCL unpleasantness from the user. Not disruptive to serial logic. By far the easiest way to use graphics devices.

  ▶ http://viennacl.sourceforge.net
Illustration of Coding Strategies

- Message Passing Interface – MPI
- POSIX Threads – Pthreads
- Open Multi Processing – OpenMP
- Open Computing Language – OpenCL
- Vienna Computing Library – ViennaCL
Introduction to MPI

- Pacheco, Peter S., A User’s Guide to MPI (1995), Manuscript, Department of Mathematics, University of San Francisco. 
  ftp://math.usfca.edu/pub/MPI/mpi.guide.ps

- Open MPI documentation 
  http://www.open-mpi.org/doc
Hello World in MPI

main(int argc, char** argp)
{
    int my_rank;       // Rank of process
    const int buflen = 100; // Max message size
    char buffer[buflen]; // Buffer for messages
    int no_procs;      // Number of processes
    int tag = 50;       // Tag for messages

    MPI_Init(&argc, &argp);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &no_procs);

    if (my_rank != 0) { // Slave
        sprintf(buffer, "%tGreetings from process %d \n", my_rank);
        int dest = 0;
        MPI_Send (buffer, buflen, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { // Master
        for (int source = 1; source < no_procs; ++source) {
            MPI_Status status;
            MPI_Recv(buffer,buflen,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
            cout << buffer;
        }
    }

    MPI_Finalize();
}
MacBook Pro, Intel Core i7, OS 10.7.3

Greetings from process 1
Greetings from process 2
Greetings from process 3
Greetings from process 4
Greetings from process 5
Greetings from process 6
Greetings from process 7
Comments on MPI

• Simple design, minimal disruption of serial code
  ▶ Master reads data, sends data, receives results
  ▶ Slaves receive the data, do the work, send results

• MPI is by far the most useful for my work: simulation estimators for nonlinear models by MCMC
  ▶ http://www.aronaldg.org/webfiles/emm/
  ▶ http://www.aronaldg.org/webfiles/snp/
  ▶ http://www.aronaldg.org/webfiles/gsm/
Posix Threads (Pthreads)

Threads are processes that can run independently and simultaneously within a process.

Reference: POSIX Threads Programming
http://www.llnl.gov/computing/tutorials/pthreads/
Thread Properties

- A thread exists within the process that creates it and uses that process's resources.

- A thread has its own independent flow of control.

- A thread has its own stack and registers.

- A thread shares memory and files with the process that creates it and with all other threads.
Consequences of Thread Properties

- Changes made by one thread to shared resources, such as closing a file, will be seen by all other threads.

- Two pointers having the same value point to the same data.

- Reading and writing to the same memory locations is possible and therefore requires explicit synchronization by the programmer.
// Compile with -pthread flag

#include "libscl.h"    // http://www.aronaldg.org/webfiles/libscl
#include <pthread.h>   // Header for pthread
#include <unistd.h>    // Header for sysconf

namespace {

struct arg_type {
    INTEGER threadid;
    std::string message;
};

void* write_arg(void* arg_ptr)
{
    arg_type* arg = (arg_type*)(arg_ptr);
    arg->message += scl::fmt('d',2,arg->threadid)() + "\n";
    std::cout << arg->message;
    pthread_exit(NULL);
}

}
int main(int argc, char** argp, char** envp)
{
    #if defined _SC_NPROCESSORS_ONLN
        INTEGER num_threads = sysconf(_SC_NPROCESSORS_ONLN);
    #else
        INTEGER num_threads = 2;
        std::cerr << "The variable _SC_NPROCESSORS_ONLN is not defined, using "
                  << num_threads << " threads instead\n";
    #endif

    pthread_t threads[num_threads];
    arg_type args[num_threads];
    int rc, t;
    for(t=0; t<num_threads; t++){
        args[t].threadid = t;
        args[t].message = "Hello from thread number ";
        rc = pthread_create(&threads[t], NULL, write_arg, (void*)(&args[t]));
        if (rc) scl::error("Cannot create thread");
    }
    pthread_exit(NULL);
}
Hello from thread number 0
Hello from thread number 1
Hello from thread number 3
Hello from thread number 6
Hello from thread number 5
Hello from thread number 2
Hello from thread number 7
Hello from thread number 4
Comments on Pthreads

- Need not disrupt serial code at all
  - Can be entirely encapsulated within objects
  - Need to take care that no shared resources are used unintentionally

- Application: particle filters
    
OpenMP

Some references

- https://computing.llnl.gov/tutorials/openMP
- https://computing.llnl.gov/tutorials/openMP/exercise.html
OpenMP

- Multithreading master/slave parallelization for SMP machines.

- The code that runs in parallel is marked with a preprocessor directive, i.e. a pragma.

- Pragmas are controlled by clauses for data sharing, synchronization, and scheduling.

- Library functions provide environment information

- After the execution of the parallelized code, the threads "join" back into the master thread.
Hello World

// Compile and link with -fopenmp flag
#include "libscl.h"
#include <omp.h>
using namespace std; using namespace scl;

int main(int argc, char** argp, char** envp)
{
    INTEGER tid, nthreads;
    string msg;
#pragma omp parallel private(nthreads, tid, msg)
    {
        tid = omp_get_thread_num();
        msg = "Hello from thread =" + fmt(’d’,3,tid)() + "\n";
        cout << msg;
        if (tid == 0) {
            bool inpar = omp_in_parallel();
            msg = "\n";
            if (inpar) msg += "Code block running in parallel\n";
            else msg += "Code block running serial\n";
            nthreads = omp_get_num_threads();
            msg += "Number of threads =" + fmt(’d’,3,nthreads)();
            msg += "\n\n";
            cout << msg;
        }
    }
    return 0;
}
MacBook Pro, Intel Core i7, OS 10.7.3

Hello from thread = 1
Hello from thread = 0
Hello from thread = 6
Hello from thread = 5

Code block running in parallel
Number of threads = 8

Hello from thread = 4
Hello from thread = 3
Hello from thread = 2
Hello from thread = 7
Advantages of OpenMP

- Simple.

- Data layout and decomposition is handled automatically by directives.

- Incremental parallelism: Can work on one portion of the program at a time, no dramatic change to code is needed.

- Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.

- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.
Disadvantages of OpenMP

- Risk of introducing difficult to debug synchronization bugs and race conditions
  
  ▶️ A race condition is two or more threads trying to write to the same memory location simultaneously

- Only runs in shared-memory multiprocessor platforms

- Requires a compiler that supports OpenMP.

- Scalability is limited by memory architecture.

- Reliable error handling is missing.

- Can’t be used on GPUs
CESM Climate Model on Bluefire

- Uses MPI to distribute across clusters
- Uses OpenMP within a cluster
- Uses special code, PIO, to eliminate MPI I/O bottleneck
OpenCL

References

- [http://www.khronos.org/opencl/registry/cl](http://www.khronos.org/opencl/registry/cl)
  - [OpenCL_1.2_Specification.pdf](http://www.khronos.org/opencl/registry/cl/OpenCL_1.2_Specification.pdf)
  - [OpenCL_1.1_C++_Bindings_Specification.pdf](http://www.khronos.org/opencl/registry/cl/OpenCL_1.1_C++_Bindings_Specification.pdf)
A Kernel to Add Two Vectors

__kernel void squareArray(__global float* input, __global float* output)" 
{
    output[get_global_id(0)] = input[get_global_id(0)]*input[get_global_id(0)];
};
Fig 1. Work Items, Work Groups, and Memory
OpenCL Addresses – 1

Indexes, called work-items, are used both to index the PEs and to determine the memory addresses that a PE uses. There can be up to three such indexes: \((x, y, z)\). I will use two, \(x\) and \(y\), for illustration.

There are \(G_x \times G_y\) PEs available with global work-items \(g_x = 0, \ldots, G_x - 1\) and \(g_y = 0, \ldots, G_y - 1\) indexing them.

The PEs are divided into work-groups of sizes \(S_x\) and \(S_y\) with work-items \(s_x = 0, \ldots, S_x - 1\) and \(s_y = 0, \ldots, S_y - 1\) indexing PEs within a work-group.

There are \(W_x \times W_y\) work-groups with work-items \(w_x = 0, \ldots, W_x - 1\) and \(w_y = 0, \ldots, W_y - 1\) indexing them.
OpenCL Addresses – 2

The relationship among work-items is

\[
\begin{align*}
g_x &= w_x \times S_x + s_x \\
g_y &= w_y \times S_x + s_y
\end{align*}
\]

Within the kernel

\[
\begin{align*}
g_x &= \text{get}\_\text{global}\_\text{id}(0) \\
g_y &= \text{get}\_\text{global}\_\text{id}(1) \\
w_x &= \text{get}\_\text{group}\_\text{id}(0) \\
w_y &= \text{get}\_\text{group}\_\text{id}(1) \\
s_x &= \text{get}\_\text{local}\_\text{id}(0) \\
s_y &= \text{get}\_\text{local}\_\text{id}(1)
\end{align*}
\]

with similar calls to get \( G_x, W_x, S_x \), etc.
OpenCL Speed

The speed of OpenCL code is governed by the same rules as for serial code. They are

- Access memory sequentially.
- Localize computations so that all fetches are from the cache.
  There is no cache on a graphics card so one has to make one’s own from local memory. Local memory is fast; global memory is slow.
- Avoid if statements.
  If you must use them, arrange code so that they evaluate to true more frequently than false because pipelines usually make that assumption.
Some Timings

matrixvecmult.cpp and matrixvecmult.cl at
http://www.aronaldg.org/webfiles/compecon/src/opencl/

exhibit five different coding strategies in illustrate gains due to
better memory management.

- The kernels compute

  \[ W = MV \]

  where \( M \) is a matrix and \( V \) is a vector.

- \( M \) is 100000 by 1100
Telsa C1060 Timing

Tesla C1060

CPU took 0.168037 sec

Testing MatrixVectorMul1
  WorkGroupSize = 64 GlobalSize 100032
  Average kernel execution time 0.137472

Testing MatrixVectorMul2
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.132226

Testing MatrixVectorMul3
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0278443

Testing MatrixVectorMul4
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0219479

Testing MatrixVectorMul5
  WorkGroupSize = 64 GlobalSize 3840
  Average kernel execution time 0.0206547
OpenCL Classes

1. Platform
2. Device
3. Context
4. Program
5. CommandQueue
6. Buffer
7. KernelFunctor

Seriously tedious!
ViennaCL

- A scientific computing library
- Includes a BLAS
- Exceptionally easy to use.

▷ http://viennacl.sourceforge.net
ViennaCL Regression Example – 1

- using libscl

```
realmat X(n,p);
realmat y(n,1);
realmat C = T(X)*X;
realmat b = invpsd(C)*(T(X)*y);
```

- memory layout is same as realmat if tag = column_major

```
viennacl::matrix<float,viennacl::column_major> gpu_X(n,p);
viennacl::matrix<float,viennacl::column_major> gpu_C(p,p);
viennacl::vector<float> gpu_y(n);
viennacl::vector<float> gpu_b(p);
```
ViennaCL Regression Example – 2

- copy X and y from CPU to GPU

  ```
  viennacl::fast_copy(X.begin(), X.end(), gpu_X);
  viennacl::fast_copy(y.begin(), y.end(), gpu_y.begin());
  ```

- compute \( b = \text{invpsd}(T(X)\times X) \times (T(X)\times y) \) on the GPU

  ```
  gpu_C = viennacl::linalg::prod(trans(gpu_X), gpu_X);
  gpu_b = viennacl::linalg::prod(trans(gpu_X), gpu_y);
  viennacl::linalg::lu_factorize(gpu_C);
  viennacl::linalg::lu_substitute(gpu_C, gpu_b);
  ```

- copy b from GPU to CPU

  ```
  viennacl::fast_copy(gpu_b.begin(), gpu_b.end(), b.begin());
  ```
ViennaCL Timing, Linux, libscl

const INTEGER p = 30;
const INTEGER n = 100000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache,
Telsa C1060, libscl

libscl least squares time = 0.205742
viennacl X & y copy time = 0.013628
viennacl least squares time = 0.001215
viennacl b copy time = 0.085871   <-- GPU to CPU expensive
viennacl total time = 0.100714
GPU/CPU total time   = 48.9516 per cent
GPU/CPU least squares time  = 0.5905 per cent
ViennaCL Timing, Linux, libscclcb

const INTEGER p = 30;
const INTEGER n = 100000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libscclcb

libsccl least squares time = 0.043693
viennacl X & y copy time = 0.013632
viennacl least squares time = 0.001202
viennacl b copy time = 0.08596   <- GPU to CPU expensive
viennacl total time = 0.100794
GPU/CPU total time = 230.6868 per cent
GPU/CPU least squares time = 2.7510 per cent
ViennaCL $C = AB$ Timing, Linux

Arows = 1000
Acols = 10000
Bcols = 1000;

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libsc1
libsc1_float mult time = 14.9863
viennacl A & B copy time = 0.661133i
viennacl mult time = 0.001064
viennacl C copy time = 0.632763
viennacl total time = 1.29496
GPU/CPU time = 8.64098 per cent

Linux 2.6.18, Intel Xeon 3.16GHz 6144 KB cache (unified),
Telsa C1060, libsc1cb
libsc1cb_float mult time = 1.68685
viennacl A & B copy time = 0.660871
viennacl mult time = 0.001028
viennacl C copy time = 0.633325
viennacl total time = 1.29522
GPU/CPU time = 76.7837 per cent

Moral: Minimize copies between CPU and GPU
Better: Use pthreads to do something simultaneously
ViennaCL $C = AB$ Timing, Apple

Arows = 1000  
Acols = 10000  
Bcols = 1000;

Mac OS X 10.6.8, Intel i7 2.66GHz 256 KB L2 (per core), 4MB L3  
GeForce GT 330M libsonl  
libscl_float mult time = 10.8579  
viennacl A & B copy time = 0.68326  
viennacl mult time = 0.063123  
viennacl C copy time = 3.37502  
viennacl total time = 4.1214  
GPU/CPU time = 37.9576 per cent

Mac OS X 10.6.8, Intel i7 2.66GHz 256 KB L2 (per core), 4MB L3  
GeForce GT 330M libsonlcb  
libscl_float mult time = 3.85385  
viennacl A & B copy time = 0.643956  
viennacl mult time = 0.064301  
viennacl C copy time = 3.39043  
viennacl total time = 4.09868  
GPU/CPU time = 106.353 per cent

Moral: Minimize CPU$\leftrightarrow$GPU copies, esp. GPU$\rightarrow$CPU  
Better: Use pthreads to do something simultaneously
ViennaCL Summary

- ViennaCL is exceptionally easy to use.

- No need to use the OpenCL classes
  1. Platform,
  2. Device,
  3. Context,
  4. Program,
  5. CommandQueue,
  6. Buffer,
  7. KernelFunctor,

- But you can to get more control or to run your own kernels
  - A ViennaCL device can be an SMP machine’s CPU’s
  - There can be more than one device
Memory Transfer Bottleneck

- Chris Gregg and Kim Hazelwood (2011) “Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer”
  - http://people.virginia.edu/ chg5w/resources/Publications

- We have benchmarked a broad set of GPU kernels on a number of platforms with different GPUs and our results show that when memory transfer times are included, it can easily take between 2 to 50x longer to run a kernel than the GPU processing time alone.

- Not a serious problem when solving GE problems