Massively Parallel Programming for Economists: Challenges and Opportunities

Jesús Fernández-Villaverde

University of Pennsylvania

November 21, 2014
Three class of problems of interest

- Life cycle models with important margins:
  1. Endogenous labor supply decisions.
  2. Human capital.
  3. Several assets (liquid, illiquid, bonds,...).

- Mid-size business cycle models with occasionally binding constraints:
  1. Zero lower bound of nominal interest rates.
  2. Financial frictions.
  3. Fiscal limits.
  4. Sudden stops.

- Non-linear estimation of state-space representations.
Microprocessor Transistor Counts 1971-2011 & Moore's Law

The graph shows the growth of transistor count in microprocessors, doubling every two years, from 1971 to 2011. Each point on the graph represents a different processor model, with the date of introduction plotted on the x-axis and the transistor count on the y-axis.

For example, the 8008 processor was introduced in 1971 with a transistor count of approximately 2,300, and the 80486 processor was introduced in 1993 with a transistor count of over 2,600,000,000.

The curve demonstrates the exponential growth of transistor counts over time, which is consistent with Moore's Law.
Why parallel?


- Problems when transistor size falls by a factor $x$:
  1. Electricity consumption goes up by $x^4$.
  2. Heat goes up.
  3. Manufacturing costs go up.

- Inherent limits on serial machines imposed by the speed of light (30 cm/nanosecond) and transmission limit of copper wire (9 cm/nanosecond): virtually impossible to build a serial Teraflop machine with current approach.

- Furthermore, real bottleneck is often memory access (RAM latency has only improved around 10% a year).

- Alternative: having more core/processors!
Cray-1: 1 core
Tianhe-2: 3,120,000 cores
ACCELERATORS / CO-PROCESSORS

- NVIDIA
- ATI
- AMD
- Intel

Systems:
- Clearspeed CSX600
- Cell
- [Other systems]

Years:
- 2006
- 2007
- 2008
- 2009
- 2010
- 2011
- 2012
- 2013
Parallel programming

- Main idea $\Rightarrow$ divide and conquer:

  1. Numerical computation.

  2. Data handling (MapReduce and Hadoop).

- Two issues:

  1. Algorithms.

  2. Coding.
Some references I

- Parallel Programming for Multicore and Cluster Systems by Thomas Rauber and Gudula Rünger.

- An Introduction to Parallel Programming by Peter Pacheco.

- Principles of Parallel Programming by Calvin Lin and Larry Snyder.

- Structured Parallel Programming: Patterns for Efficient Computation by Michael McCool, James Reinders, and Arch Robison.

- Introduction to High Performance Computing for Scientists and Engineers by Georg Hager and Gerhard Wellein.
Some references II

- “Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors.”


- *OpenCL Programming by Example* by Ravishekhar Banger and Koushik Bhattacharyya
When do we parallelize? I

- **Scalability:**
  1. Strongly scalable: problems that are inherently easy to parallelize.
  2. Weakly scalable: problems that are not.

- **Granularity:**
  1. Coarse: more computation than communication.
  2. Fine: more communication.

- Overheads and load balancing.
When do we parallelize? II

- Whether or not the problem is easy to parallelize may depend on the way you set it up.

- Taking advantage of your architecture.

- Trade off between speed up and coding time.

- Debugging and profiling may be challenging.

- You will need a good IDE, debugger, and profiler.
Example I: value function iteration

\[ V(k) = \max_{k'} \left\{ u(c) + \beta V(k') \right\} \]

\[ c = k^\alpha + (1 - \delta) k - k' \]

1. We have a grid of capital with 100 points, \( k \in [k_1, k_2, ..., k_{100}] \).
2. We have a current guess \( V^n(k) \).
3. We can send the problem:

\[ \max_{k'} \left\{ u(c) + \beta V^n(k') \right\} \]

\[ c = k_1^\alpha + (1 - \delta) k_1 - k' \]

to processor 1 to get \( V^{n+1}(k_1) \).
4. We can send similar problem for each \( k \) to each processor.
5. When all processors are done, we gather the \( V^{n+1}(k_1) \) back.
Scope of communication

broadcast

scatter

gather

reduction
Example II: random walk Metropolis-Hastings

- Get a draw from an arbitrary distribution \( P(\cdot) \)

\[
\theta \sim P(\cdot)
\]

- How?

1. Given a state of the chain \( \theta_{n-1} \), we generate a proposal:

\[
\theta^* = \theta_{n-1} + \lambda \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, 1)
\]

2. We compute:

\[
\alpha = \min \left\{ 1, \frac{P(\theta^*)}{P(\theta_{n-1})} \right\}
\]

3. We set:

\[
\begin{align*}
\theta_n &= \theta^* \text{ w.p. } \alpha \\
\theta_n &= \theta_{n-1} \text{ w.p. } 1 - \alpha
\end{align*}
\]

- Problem: to generate \( \theta^* \) we need to \( \theta_{n-1} \).
- No obvious fix (parallel chains violate the asymptotic properties of the chain).
Limitations

- Amdahl’s Law: the speedup $S$ of a program using $N$ processors in parallel computing is limited by the fraction of time $P$ needed for the sequential fraction of the program.

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

- Overheads:
  1. Cost of starting a thread or process.
  2. Cost of communicating shared data.
  3. Cost of synchronizing.

- Load imbalance: often we use less than 10% of capability of a machine.
Memory I

- How is memory handled?
  1. Shared (at least L1 cache).
  2. Distributed.

- Cost vs. speed.

- Rule of thumb: 1 GB of memory is a 10,000x10,000 matrix at single precision.
Memory II

- Temporal and spatial locality.

- Reuse memory in cache as much as possible.

- Avoid cache misses and cache collisions.

- Prefetching.

- Optimization flags and profiler.

- Donald Knuth: Premature optimization is the root of all evil.
Machines

- Your laptop.
- A cluster.
- Cloud computing.
- Supercomputer.
Cloud computing

- “Utility” model of computing.
- Replace a large initial capital cost for a variable cost (use-as-needed).
- For example (https://aws.amazon.com/ec2/pricing/):
  1. 8 processors with 30 GiB, general purpose: $0.56 per hour.
     $403.2 if used full time.
  2. 32 processors with 60 GiB, compute optimized $1.68 per hour.
     $1209.6 per month if used full time.
- You save maintenance time.
Amazon Web Services (AWS)


- You can get plenty of books on Kindle about AWS.
- Access to an Amazon Machine Images (Matlab) or create your own and use X11.
Running an instance

- Go to https://console.aws.amazon.com/console/home (other ways to get there).

- Click on EC2.

- Click on Launch Instance and follow the window links (you may want to get a public key).

- Example: python and C++.

- Open a terminal and connect to the DNS address.

- You can post results online.
Programming modes I

- Automatic parallelization:
  1. AutoParInGCC.
  2. Intel compilers.
  3. Coarrays.

- Partitioned Global Address Space Languages (PGAS): CAF, UPC, X10 (http://x10-lang.org/).

- A small detour: advantage of functional programming for parallelization (Scala, Clojure, Ocaml, Haskell).
int main(int argc, char** argv)
{
    if (MYTHREAD == 0) printf("hello world\n");
    printf("I am thread number %d of %d threads\n", MYTHREAD, THREADS);
    return 0;
}

You can compile the code with GNU UPC
Programming modes II

- Explicit parallelization:
  1. OpenMP.
  2. MPI.
  3. GPU programming: CUDA and OpenCL.

- Pthreads (POSIX threads).

- Other parallel tools (Matlab parallel toolbox, Julia).

- Hybrids.
OpenMP I

- Open specifications for multi-processing.
- It has been around for two decades. Current version 4.0.
- Official web page: http://openmp.org(wp/)
- Tutorial: https://computing.llnl.gov/tutorials/openMP/
- *Using OpenMP: Portable Shared Memory Parallel Programming* by Barbara Chapman, Gabriele Jost, and Ruud van der Pas.
- Fast to learn, reduced set of instructions, easy to code, but you need to worry about contention and cache coherence.
API for multi-processor/core, shared memory machines defined by a group of major computer hardware and software vendors.

C++ and Fortran. Extensions to other languages.

For example, you can have OpenMP in Mex files in Matlab.

Supported by major compilers (GCC) and IDEs (Eclipse).

Thus, it is usually straightforward to start working with it.
OpenMP III

- Multithreading with fork-join.

- Rule of thumb: One thread per processor.

- Job of the user to remove dependencies and synchronize data.

- Heap and stack (LIFO).

- Race conditions: you can impose fence conditions and/or make some data private to the thread.

- Remember: synchronization is expensive and loops suffer from overheads.
Compiler directives to tell what to parallelize:

```plaintext
#pragma omp parallel default(shared) private(beta,pi)
```

Compiler generates explicitly threaded code when OpenMP flag is invoked (`-fopenmp`).

We can always recompile without the flag and compiler directives are ignored.

Most implementations (although not the standard!) allow for nested parallelization and dynamic thread changes.
General code structure

#include <omp.h>

main () {
    int var1, var2, var3;
    /// Serial code

    ...

    // Parallel section
    // Fork a team of threads and specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3){
        // Parallel section executed by all threads
        // Other OpenMP directives
        // Run-time Library calls
        // All threads join master thread and disband
    }

    // Resume serial code
}

Example code

#include <omp.h>

main() {
    int i, N = 1000, chunk = 100;
    float a[N], b[N], c[N];
    // Some initializations
    for (i=0; i < N; i++) {
        a[i] = i*1.0;
        b[i] = i*1.0;
    }
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++) {c[i] = a[i] + b[i];}
    } // end of parallel section
}
MPI I

- Message Passing Interface is a standardized and portable message-passing system based on the consensus of the MPI Forum.

- Official web page (and for downloading binaries):
  http://www.open-mpi.org/

- Tutorial: https://computing.llnl.gov/tutorials/mpi/

- A couple of references:


  2. MPI: The Complete Reference - Volumes 1 and 2, by several authors.
MPI II

- MPI is organized as a library performed with routine calls.

- Bindings for C++ and Fortran. Also for Python, OCaml, and R.

- For example, you can have MPI in Mex files in Matlab.

- Harder to learn (MPI 3.0 standard has more than 440 routines) and code, but extremely powerful⇒used for state-of-the-art computations.

- Multiple processes (thread with its own controller).

- Thus, better for coarse parallelization.
Invoked with a compiler wrapper

```bash
>>mpic++ -o ClassMPI ClassMPI.cpp.
```

Plenty of libraries (PLAPACK, Boost.MPI).

Parallel I/O features.
Example code

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " " of " << size << "\n";
    MPI::Finalize();
    return 0;
}
```
Routines

- Communication:
  1. Send and receive: between two processors.
  2. Broadcast, scatter, and gather data on all processors.
  3. Compute and move (sum, product, max of, ...) data on many processors.

- Synchronization.

- Enquiries:
  1. How many processes?
  2. Which process is this one?
  3. Are all messages here?
MPI derived types

- MPI predefines its primitive data types:
  1. MPI_CHAR
  2. MPI_DOUBLE_PRECISION
  3. MPI_C_DOUBLE_COMPLEX

- Also for structs and vectors.

- Particularly important for top performance.
VampirTrace II

- A program tracing package that can collect a very fine grained event trace of the sequential or parallel program.

- The traces can be visualized by the Vampir tool and a number of other tools that read the Open Trace Format (OTF).

- VampirTrace is included in Open MPI v1.3 and later.


- Invoke with mpic++-vt.

- Can also be used for OpenMP.

- Tracing disadvantages.
Seymour Cray

If you were plowing a field, which would you rather use:
Two strong oxen or 1024 chickens?
Distributed Neural Networks with GPUs in the AWS Cloud

by Alex Chen, Justin Basilico, and Xavier Amatriain

As we have described previously on this blog, at Netflix we are constantly innovating by looking for better ways to find the best movies and TV shows for our members. When a new algorithmic technique such as Deep Learning shows promising results in other domains (e.g., Image Recognition, Neuro-imaging, Language Models, and Speech Recognition), it should not come as a surprise that we would try to figure out how to apply such techniques to improve our product. In this post, we will focus on what we have learned while building infrastructure for experimenting with these approaches at Netflix. We hope that this will be useful for others working on similar algorithms, especially if they are also leveraging the Amazon Web Services (AWS) infrastructure. However, we will not detail how we are using variants of Artificial Neural Networks for personalization, since it is an active area of research.

Many researchers have pointed out that most of the algorithmic techniques used in the trendy Deep Learning approaches have been known and available for some time. Much of the more recent innovation in this area has been around making these techniques feasible for real-world applications. This involves designing and implementing architectures that can execute these techniques using a reasonable amount of resources in a reasonable amount of time. The first successful instance of large-scale Deep Learning made use of 16000 CPU cores in 1000 machines in order to train an Artificial Neural Network in a matter of days. While that was a remarkable milestone, the required infrastructure, cost, and computation time are still not practical.

Andrew Ng and his team addressed this issue in follow up work. Their implementation used GPUs as a powerful yet cheap alternative to large clusters of CPUs. Using this architecture, they were able to train a model 6.5 times larger in a few days using only 3 machines. In another study, Schwenk et al. showed that training those models on GPUs can improve performance dramatically, even when comparing to high-end multicore CPUs.
Big difference

- Latency: amount of time required to complete a unit of work.

- Throughput: amount of work completed per unit of time.

- Latency devices: CPU cores.

- Throughput devices: GPU cores (these cores, known as thread processors, are grouped together into multiprocessors.):
  
  1. Have a lower clock speed.
  
  2. More transistors for arithmetic operations, fewer to control flow and data caching.
  
  3. Less memory.

- Intermediate: Coprocessors.

- Nature of your application?
Coprocessor
When to go to the GPU?

1. Problem is easily scalable because computation is massively parallel.

2. Much more time spent on computation than on communication:
   - CPU memory is “host memory.”
   - GPU memory is “device memory.”
   - One needs to transfer data objects between one and the other.

Remember: a GPU is attached to the CPU via a PCI (Peripheral Component Interconnect) Express bus.
CUDA vs. OpenCL

- **CUDA (Compute Unified Device Architecture):**
  1. Developed by NVIDIA for its GPUs.
  2. Extension to C++. Wrappers to Fortran and other languages available.

- **OpenCL (Open Computing Language) is an open standard.**

- Advantages and disadvantages.
Structure of the code

- **Algorithm**: a set of threads.

- Each thread is assigned to a core.

- Each thread has a kernel+user-defined data structure.

- Threads are grouped in blocks. Each block is executed in 1 GPU.

- However, several blocks can run in 1 GPU.

- Single instruction, multiple thread execution mode.
Example code

//Compute vector sum C=A+B
//Each thread performs one pairwise addition
    global__ void vecAdd(float* a, float* b, float* c) {
        int i = blockIdx.x*blockDim.x+threadIdx.x;
        c[i] = a[i]+b[i];
    }

int main() {
    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256,256>>>(d_a, d_b, d_c);
}
Libraries

- Working with CUDA directly is costly and not time effective.

- Libraries: cuBLAS, Cusp, CUDA Math Library, etc.


- Two libraries you can look at:

  1. Thrust: parallel algorithms library which resembles the C++ Standard Template Library (STL). Comes with the CUDA toolkit.

```c
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>

int main(void){
    // H has storage for 4 integer
    thrust::host_vector<int> H(4);
    // initialize individual elements
    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;
    // H and D are deleted when the function returns
    return 0; }
```
Example code

#include <stdio.h>
#include <math.h>
#include <arrayfire.h>
using namespace af;

int main(void) {
    // sample 40 million points on the GPU
    array x = randu(20e6), y = randu(20e6);
    array dist = sqrt(x * x + y * y);
    // pi is ratio of how many fell in the unit circle
    array pi = 4.0 * sum(dist < 1) / 20e6;
    print(pi);
}
Multiple GPUs

- CUDA will handle the details of a particular GPUs (without too much efficiency loss).

- Thus, you can port your code from one GPU to another without too much tinkering.

- Nowadays you can have several GPUs installed in one machine (or a cluster).

- Communication among them achieved with MPI.
Matlab parallel toolbox I

- Core MATLAB implements multithreading in its BLAS and in its element-wise operations.

- But parallel toolbox allows for explicit parallelization.

- Works for:
  1. Multicore processors.
  2. GPUs.
  3. Clusters.

- Related but different from Matlab distributed computing server: run works on clusters, clouds, and grids.
Matlab parallel toolbox II

- Supports:
  1. Parallel loops.
  2. CUDA on NVIDIA GPUs.
  3. Interactive and batch processing.
  4. Distributed arrays and spmd.

- Less powerful than dealing directly with OpenMP or MPI.
Parallel pool

>> parpool

- It creates a parallel pool of workers on a cluster and returns a pool object.

- Access from the GUI.

- A parallel pool will be created automatically when one parallel command is executed (can be changed in settings).

- Related keywords:

  1. poolobj=gcp.

  2. delete(poolobj).
Parfor I

>> parfor loopvar = initval:endval
>>    loop body
>> end

- Basic parallel control structure: see basic_parfor.m.
- Roughly equivalent to the OpenMP pragma for.
- Mlint understands parfor.
Four features:

1. `loopvar` must be monotonically increasing integer values.

2. Order of execution is not guaranteed.

3. Restrictions apply to statements in `loop` body (for example, array references must be direct).

4. If you do not have multiple workers, it will be executed as a sequential loop.

5. You cannot nest parfors.
Examples

- Typical problem: linear solver.

- Examples:

  1. Example_parfor_1.m.

  2. Example_parfor_2.m.
Parfeval

\[ F = \text{parfeval}(\text{fcn}, \text{numout}, \text{in1}, \text{in2}, \ldots) \]

- Execute function asynchronously on parallel pool worker.

\[
\begin{align*}
\gg f &= \text{parfeval}(@\text{magic}, 1, 10); \\
\gg \text{value} &= \text{fetchOutputs}(f);
\end{align*}
\]

- Variation: parfevalOnAll
Distributed arrays

\[ \text{AA} = \text{distributed}(A) \]

- Creates a distributed array.
- This allows a parallel pool to accommodate an array that might be too large to fit into the memory of a single machine.

\[ \text{pmode start 4} \]
\[ \text{a} = \text{rand}(2,2) \]
\[ \text{a} = \text{distributed}(\text{rand}(2,2)) \]
\[ \text{pmode quit} \]
Codistributed arrays

```matlab
>> spmd
>> A = zeros(80, 1000);
>> D = codistributed(A)
>> end
```

- A codistributed array is divided into segments (called local parts), each residing in the workspace of a different worker.

- Over 150 Matlab functions are overloaded for codistributed arrays.
GPUs

>>&gpuDevice

- Inquires about the existence of a GPU device.

>>&gpuDeviceCount

- Counts how many GPU devices we have.

>>&reset(gpu)

- Resets GPU.

>>&gpuBench(gpu)

- Benchmarks GPU (you need to download extra files).
GPU arrays I

>> G = gpuArray(X)

- Sends a numerical array to the GPU.

>> G = gathers(X)

- Retrieves a numerical array from the GPU.

>> >> G = existsOnGPU(X)

- Returns a logical value indicating whether X is on the GPU.

- Example: Example_gpuArray_1.m.
GPU arrays II

- Transferring data to and from the GPU is costly: MeasuringGPU.m.

- When feasible, it is a better idea to build the data directly in the GPU.

- Some examples (among several others):

  ```matlab
  >> A = gpuArray.zeros(M,N)
  >> A = gpuArray.ones(M,N)
  >> A = gpuArray.rand(M,N)
  ```
When does it work?

- Scalable problems: `Example_gpuArray_1.m`.
- Less scalable problems: `Example_gpuArray_2.m`.
- Use `arrayfun` and `bsxfun`: `myTestClass.m`.
- A more general point: vectorization is particularly important for the GPU.
- Need to take special care of complex numbers.