Testing Fine-Grained Parallelism for the ADMM on a Factor-Graph

Nate Derbinsky

Wentworth Institute of Technology

Ning Hao
Oracle

AmirReza Oghbaee
Northeastern

Mohammad Rostami
UPenn

José Bento
Boston College

The Problem

- Large-scale optimization problems
  - Many hard/soft constraints
  - Many discrete/continuous variables
  - Often not smooth/convex
- Diverse applications: vision/imaging, robotics, machine learning, …
- Thus, we need tools that balance
  - Problem independence
  - Ease of use
  - Efficiency/scalability

Our Framework: **parADMM**

- General optimization via the Alternating Direction Method of Multipliers (ADMM)
  - State-of-the-art performance on numerous tasks
- **Automatically** exploit CPU/GPU parallelism given user's *serial* code
  - Written in C; integrates CUDA, OpenMP
- Empirical GPU/CPU speedup results
  - Packing
  - Optimal control
  - Machine learning (SVM)

Optimization Problem

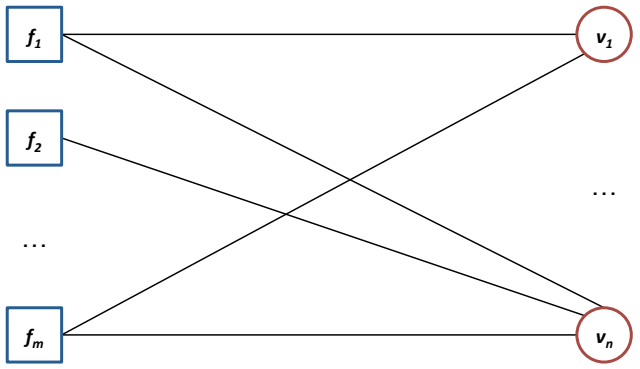
Soft

$$\underset{\boldsymbol{v}\in\mathbb{R}^n}{\text{minimize}} : f(\boldsymbol{v}) = f_1(v_1, v_2, \dots) + f_2(\dots) + \dots$$

$$+ \sum c_k(v_1, v_2, \dots) = \begin{cases} 0 & \text{constraint met} \\ \infty & \text{else} \end{cases}$$

Hard

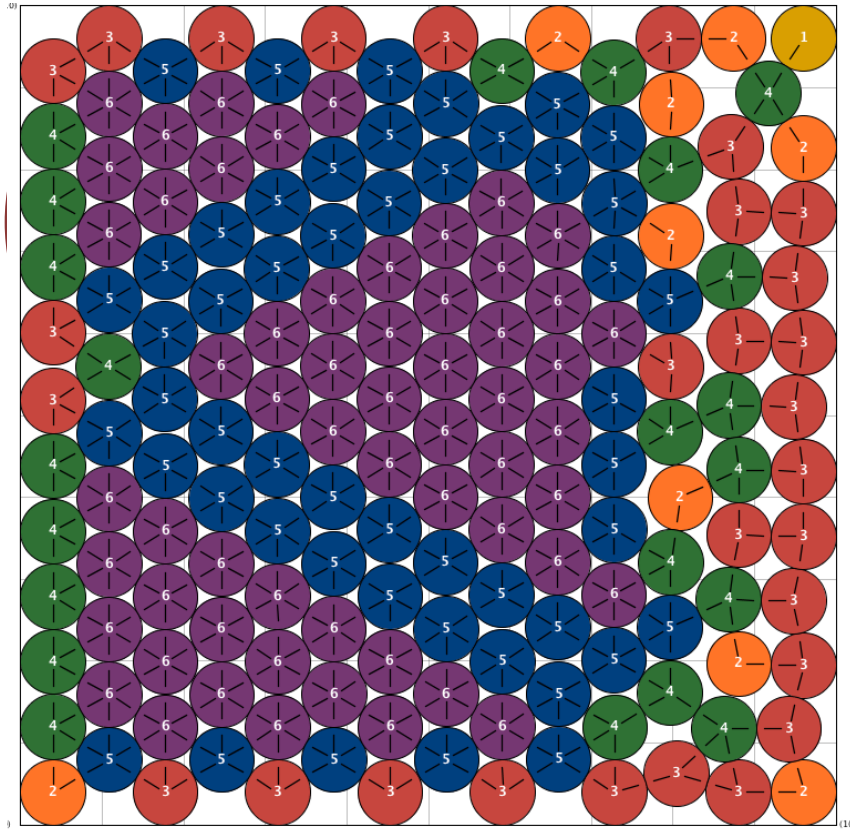
Objective/Cost Function

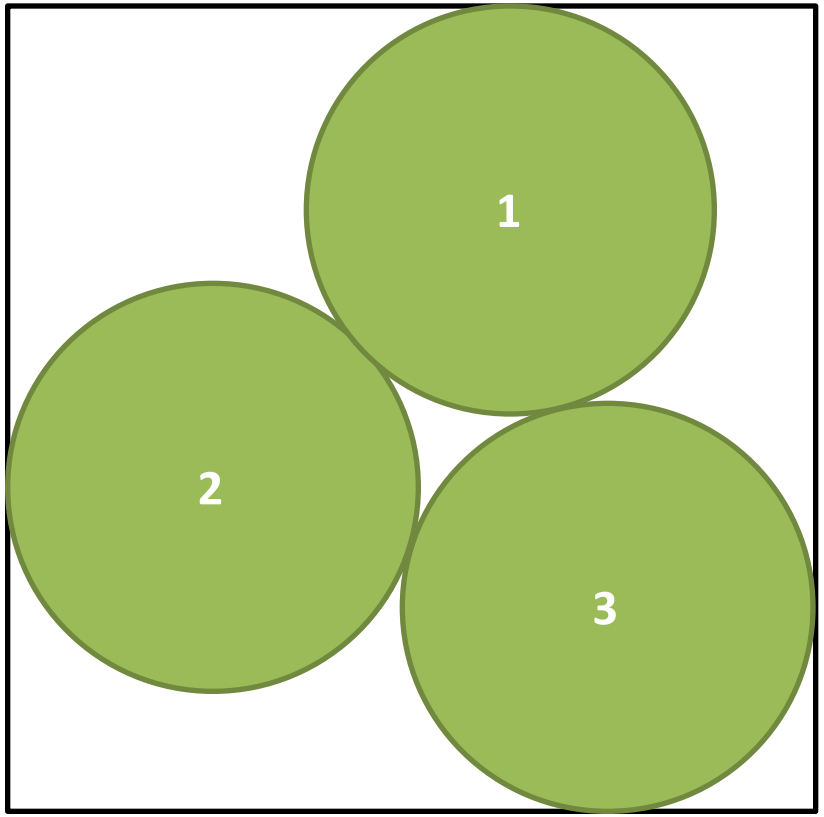


Factor Graph

Example: Packing

Fit *n* circles of radius *r* in a square of side-length *s* without overlap (non-convex, NP-hard, ∞ solutions)

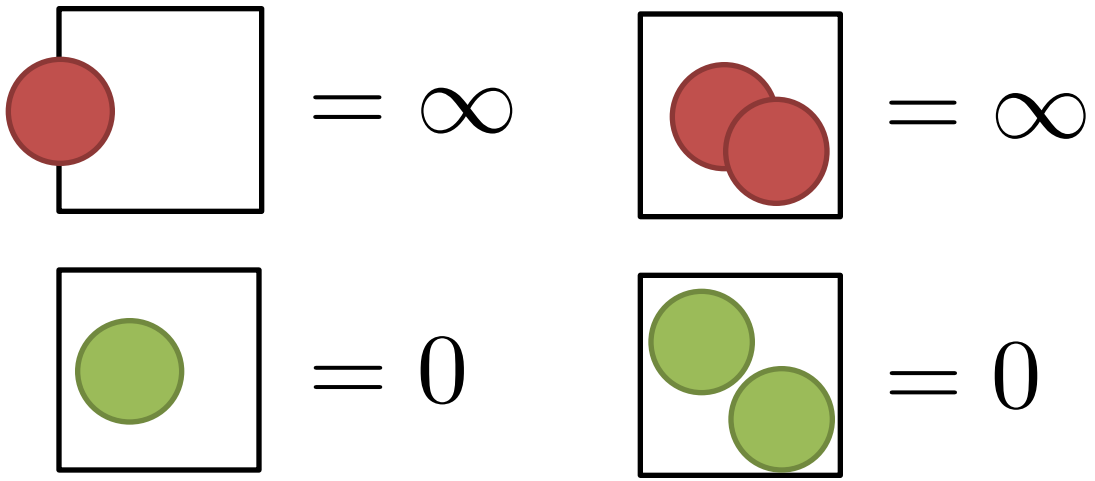


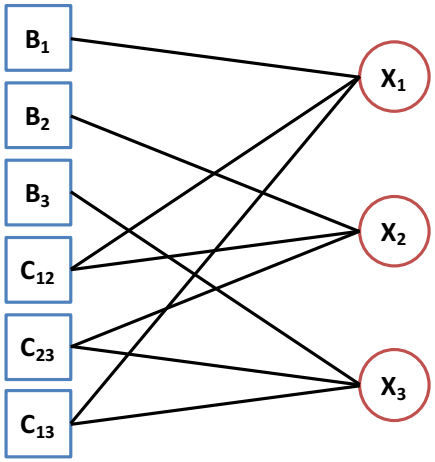


Packing Objective

$$\underset{x_1,x_2,x_3}{\text{minimize}} : \text{Box}(x_1) + \text{Box}(x_2) + \text{Box}(x_3) +$$
$$\text{Collision}(x_1, x_2) + \text{Collision}(x_2, x_3) + \text{Collision}(x_1, x_3)$$

$$\text{Box}(x) \quad \text{Collision}(x, x')$$





ADMM

Alternating Direction Method of Multipliers [Boyd et al. '11]

General

- Arbitrary objective functions, constraints, and variables
- Global minimum for *convex* problems (and demonstrably successful for non-convex as well)
- If converges, produces a *feasible* solution (all hard constraints met)

Interruptible

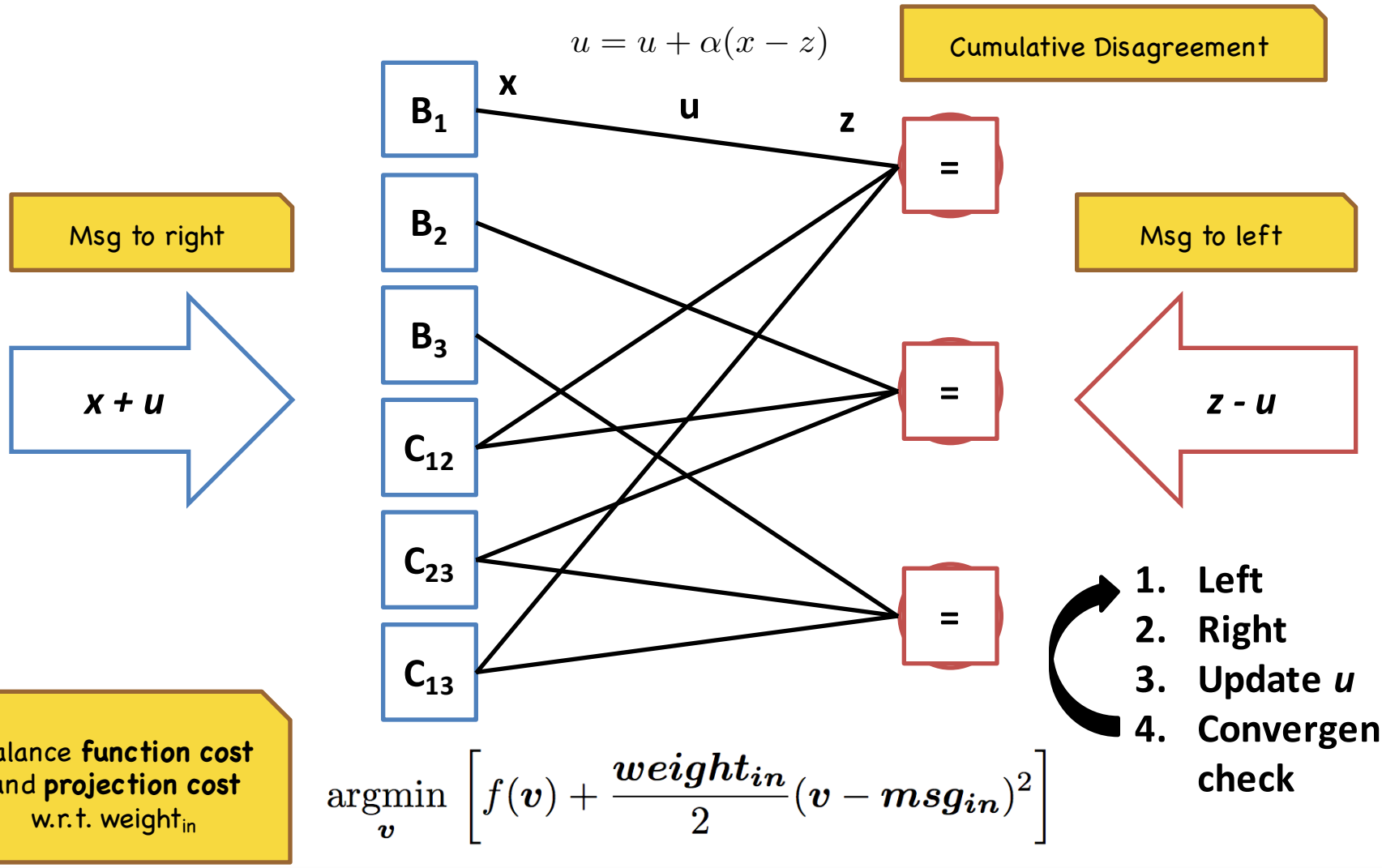
- Iterative algorithm; intermediate results can serve as heuristic start for complementary approaches

Scalable and Parallelizable

- Formulated as a *decomposition-coordination* problem; leads naturally to concurrency at multiple levels (e.g. MapReduce, multi-core, GPU)

Message-Passing ADMM

[Derbinsky et al. '13]



Example `parADMM` Program

```
graph   Cpu_graph;        // bipartite graph in the CPU
graph   Cpu_Gpu_graph;    // interface between CPU and GPU graph
graph*  Gpu_graph;        // bipartite graph in the GPU
cudaMalloc( (void **) &Gpu_graph   ,   sizeof(graph) );

startG( &Cpu_graph, number_of_dims_per_edge );   // initialize empty CPU graph

// add nodes to the bipartite graph
int index_of_variables_1[] = {1,2,3};   int number_of_variables_1 = 3;
int index_of_variables_2[] = {1,4,5};   int number_of_variables_2 = 3;
int index_of_variables_3[] = {2,5};     int number_of_variables_3 = 2;
int index_of_variables_4[] = {5};       int number_of_variables_4 = 1;

addNode(&Cpu_graph, proximal_operator_1, (void *) parameters_1,
    size_parameters_1,     number_of_variables_1, index_of_variables_1);
addNode(&Cpu_graph, proximal_operator_2, (void *) parameters_2,
    size_parameters_2,     number_of_variables_2, index_of_variables_2);
addNode(&Cpu_graph, proximal_operator_3, (void *) parameters_3,
    size_parameters_3,     number_of_variables_3, index_of_variables_3);
addNode(&Cpu_graph, proximal_operator_4, (void *) parameters_4,
    size_parameters_4,     number_of_variables_4, index_of_variables_4);

// set the rhos and alpha values (all equal)
initialize_RHOS_APHAS(&Cpu_graph, rho, alpha);

// initialize the ADMM variables (at random between lower and upper bound)
initialize_X_N_Z_M_U_rand(&Cpu_graph, lower_bound, upper_bound, lower_bound,
    upper_bound, lower_bound,upper_bound, lower_bound, upper_bound, lower_bound,
    upper_bound);

// initialize the Cpu_Gpu_graph using the GPU graph
copyGraphFromCPUtoGPU(Cpu_graph, &Cpu_Gpu_graph, Gpu_graph);

// iterate the ADMM
for (int i = 0; i< numiterations; i++)
{
    updateXGPU<<< ... , ... >>>(GPU_graph);
    updateMGPU<<< ... , ... >>>(GPU_graph);
    updateZGPU<<< ... , ... >>>(GPU_graph);
    updateUGPU<<< ... , ... >>>(GPU_graph);
    updateNGPU<<< ... , ... >>>(GPU_graph);
}

// copy the variables Z from the GPU graph to the CPU graph
cudaMemcpy(Cpu_graph.Z  ,  Cpu_Gpu_graph.Z   ,  CPU_graph.num_dims *
    CPU_graph.num_vars *   sizeof(double),cudaMemcpyDeviceToHost);

...
```

Example Proximal Operator

```
__device__ double d_max_alpha_radius(double n, double rho, double alpha)
{
        const int newton_iterations = 10;
        double r = d_abs_d( n / 2.0 );

        for ( int i=0; i<newton_iterations; i++ )
        {
                const double num = -pow( r, alpha-1 ) + ( rho / alpha )*( r - n );
                const double denom = -( alpha - 1 ) * pow( r, alpha-2 ) + ( rho / alpha );

                r -= num / denom;

                if ( r < 0 )
                {
                        return 0.0;
                }
        }

        return r;
}

__device__ void d_maximize_sum_of_squared_disk_radii_alpha(double *x, double *n, double *rhos, int numvars, int numdims, void *params)
{
        double* paramsCasted = (double*) params;

        for ( int i=0; i<numvars; i++ )
        {
                double rho_i = rhos[i];
                double rad_i = d_max_alpha_radius(n[ i*numdims ], rho_i, paramsCasted[0]);

                for ( int j=0; j<numdims; j++ )
                {
                        x[ i*numdims + j ] = rad_i;
                }
        }
}
```

Example Packing Solution

Fine-Grained Hypothesis

- When applying ADMM, problem decomposition is user-defined
  - Typically involves relatively few & large [with parallelism within factors], which may be useful in MapReduce-like context

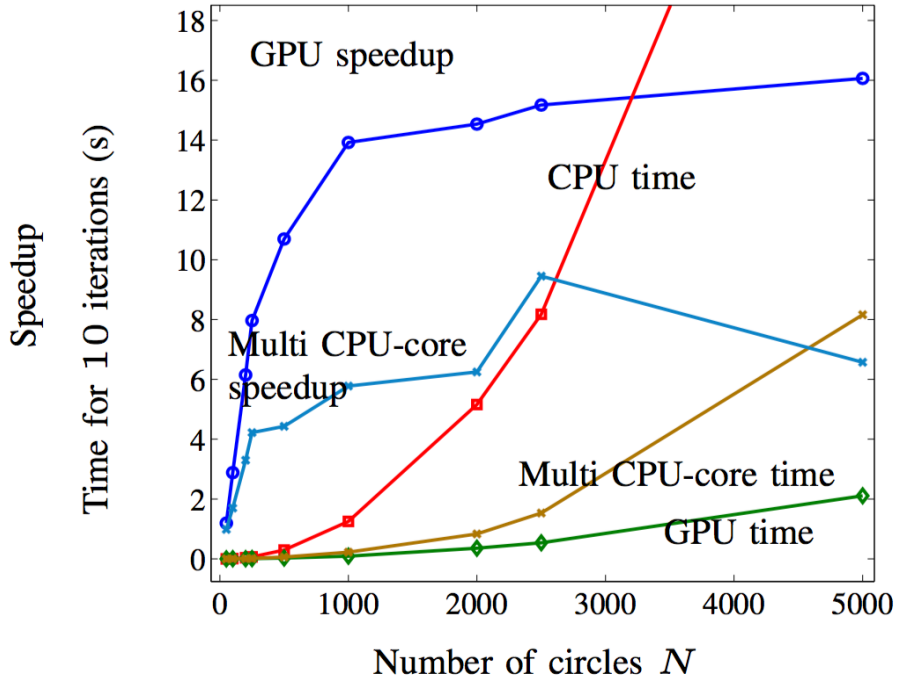
- Our hypothesis: efficient and automatic parallelization via many & small factors (1 thread per factor)
  - Bonus: the user needs only implement factor logic using serial code, typically quite simple

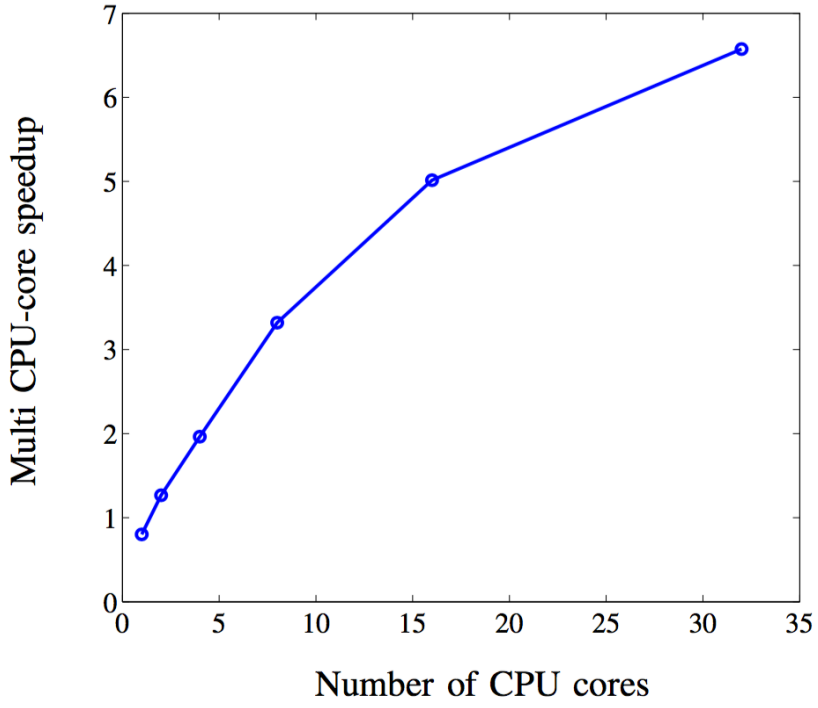
Empirical Evaluation

- AMD Opteron Abu Dhabi 6300 @ 2.8GHz
  - Up to 32 cores, 128GB memory
  - **`pragma openmp parallel for`**
- NVIDIA Tesla K40
- Ubuntu

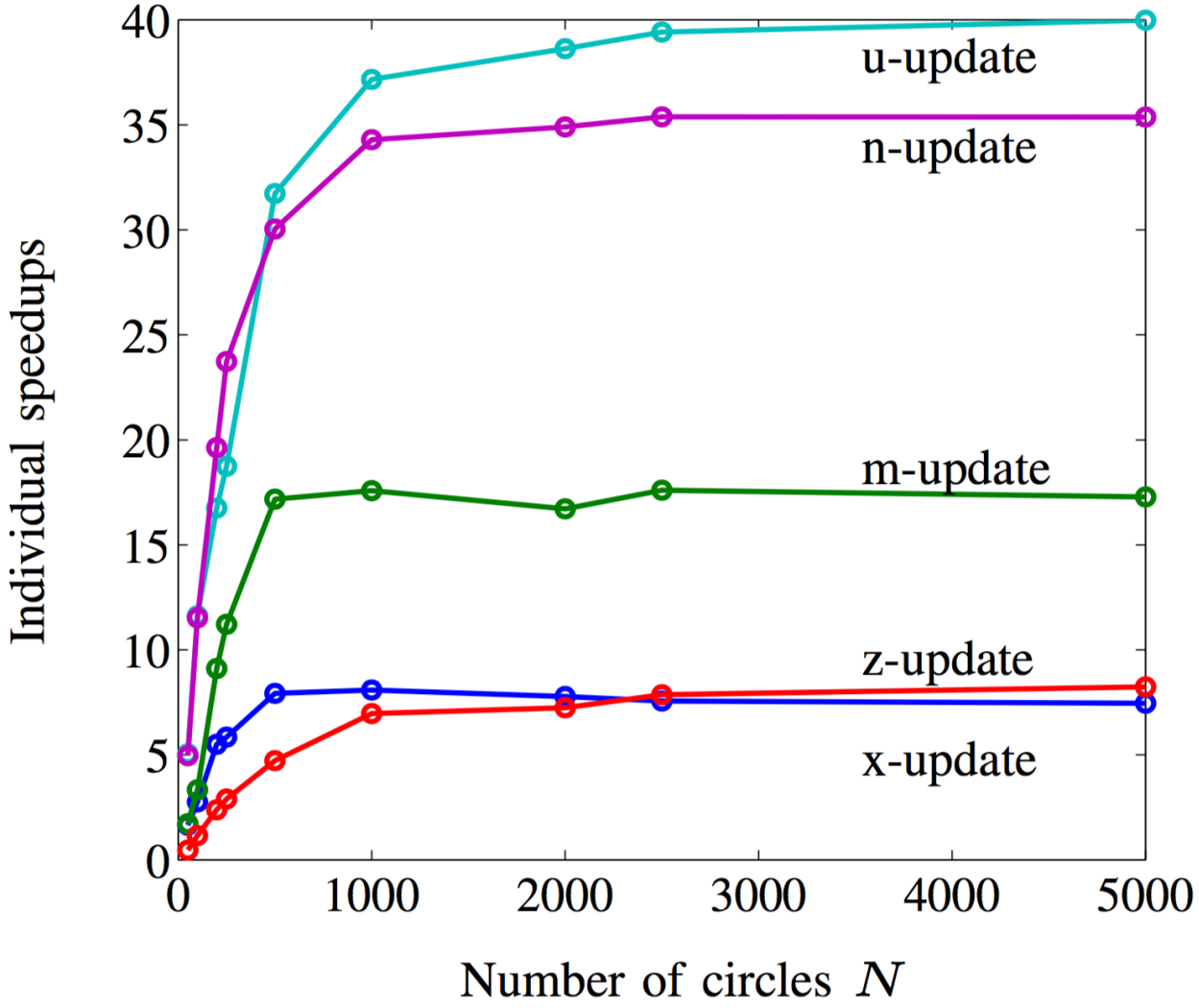
- Numerical results in three domains
  - Focus: speedup with problem size
    - No change to factor code
    - Minimal change to graph-creation code

Packing Circles in a Triangle

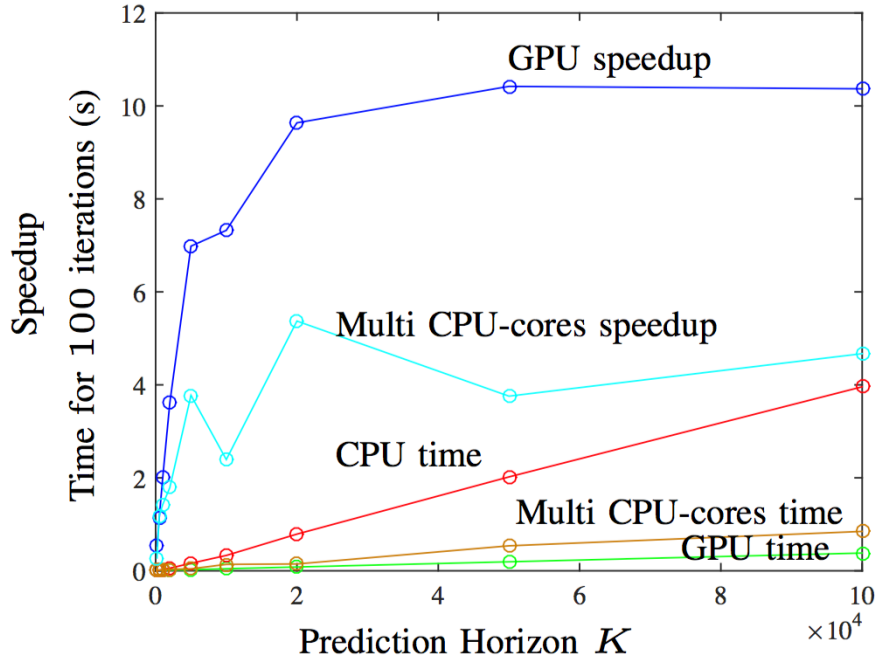


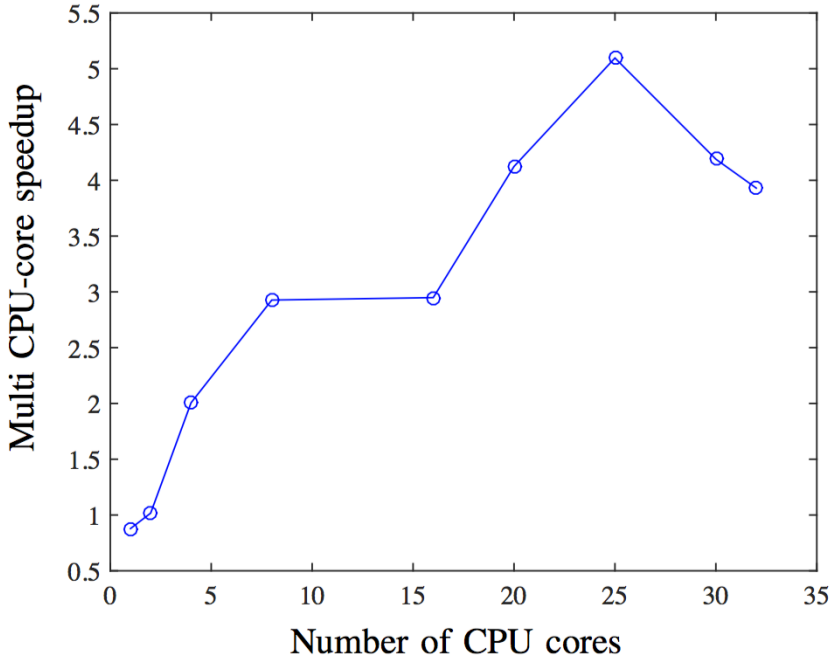


Phase Breakdown

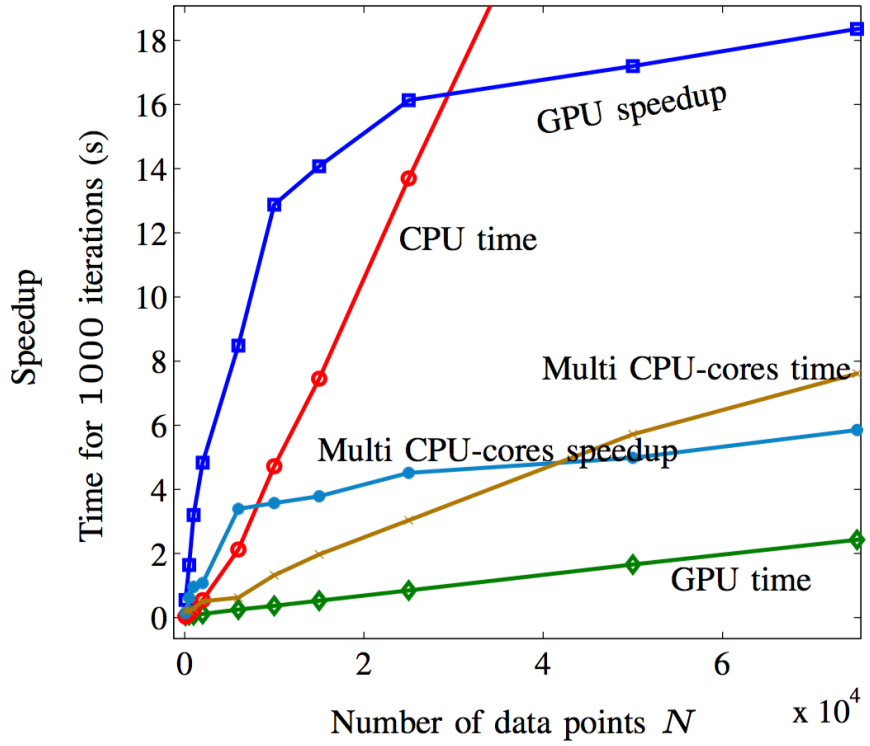


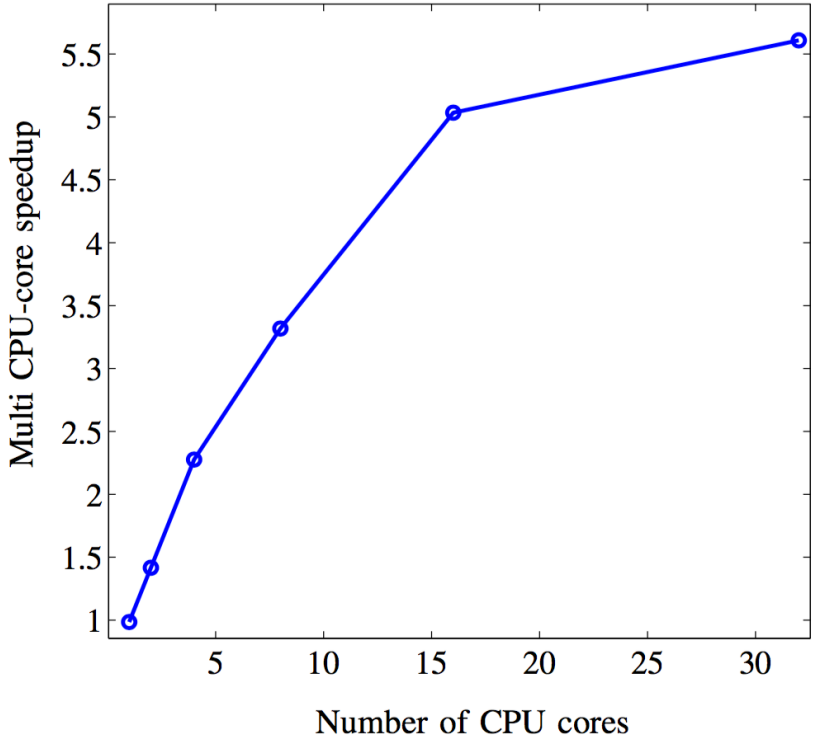
Model Predictive Control (MPC)





Binary Classification via SVM in $\mathbb{R}^2$





Conclusions

- We achieve problem-independent scaling with user-supplied **serial** code
  - Multi-core CPU (5-6x with 32-cores)
  - GPU (10-18x; comparable with other GPU-accelerated libraries)

- Future work
  - Improved work-scheduling/topology
  - Multiple GPU/computer, asynchronous ADMM

Additional Details in the Paper

- Classic ADMM formulation
- **`parADMM`** internals
  - Includes CUDA, OpenMP parameterization
- Survey of related tools/frameworks
- Empirical evaluation
  - Problem formulation
  - Analysis of results

The End :)

Questions?

Ning Hao
Oracle

AmirReza Oghbaee
Northeastern

Mohammad Rostami
UPenn

José Bento
Boston College

https://github.com/parADMM/engine