Work Information

Logan Brown

February 5, 2015

Contents

1 PSUADE

1.1 Installing PSUADE

1.1.1 General Steps

1. tar xzvf PSUADE.tar.gz
2. cd PSUADE_1.7.2
3. mkdir build; cd build
4. cmake .. &> cmake.output &
   (This step will take a minute or so.)
5. make &> make.output &
   (This step will take some time.)

If the make is successful, build/bin should contain the psuade executable. Set $PATH and $LD_LIBRARY_PATH, based on your installation location. To test your build, change to your build directory and type "make test".

1.1.2 Installing on Darter

1. *tar xzvf PSUADE.tar.gz*
2. *cd PSUADE_1.7.2*
3. *mkdir build; cd build*
4. *cmake .. &> cmake.output &*
5. **Go to cmake_install.cmake and change CMAKE_INSTALL_PREFIX from "/usr/local/" to a directory where you have permission.**
   I used "PSUADE_v1.7.2/inst/"
6. *make &> make.output &*

1.1.3 Installing on star1

Note: Requires cmake, you may have to install a local copy.

1. *tar xzvf PSUADE.tar.gz*
2. *cd PSUADE_1.7.2*
3. *mkdir build; cd build*

4. **export LD_LIBRARY_PATH=**

   /home/kwong/LAPACK/lib-shared/:$LD_LIBRARY_PATH

   Alternatively, use another libblas.so, if you have one.

5. *cmake .. &> cmake.output &*

6. **Go to cmake_install.cmake and change CMAKE_INSTALL_PREFIX from "/usr/local/" to a directory where you have permission.**

7. **Go to CMakeCache.txt and change LAPACK_lapack_LIBRARY:FILEPATH to**

   /home/kwong/LAPACK/lib-shared/liblapack.so

   OR

   /home/lbrown/iel-2.0/EXTLIB/LAPACK/lapack-3.4.0-shared/liblapack.so

   or to whatever liblapack.so you prefer

8. *make &> make.output &*

1.2 Compiling PSUADE as a Module

1.2.1 main in Psuade.cpp as a module

I recommend first copying over Psuade.cpp to PsuadeIEL.cpp, and making all of the relevant changes in that file. That way, you can still compile Psuade as a standalone software piece for data analysis and debugging, in addition to compliling the Psuade module library.

If you choose to just modify Psuade.cpp, replace all instances of 'PsuadeIEL.cpp' below with 'Psuade.cpp'. The resulting library will be incapable of running PSUADE without the DIEL, because it will lack a main function.

1. Add include files to PsuadeIEL.cpp

```
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/IEL.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/executive/IEL\_exec\_info.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/tuple\_comm.h"
#include "../../../../iel-2.0/EXECUTIVE/HYBRID/IEL\_comm/arrayList.h"
```

2. Change main(int argc, char** argv) to PSUADEmain(IEL_exec_info_t *exec_info)

3. Add the following lines to CMakeLists.txt

```
SET(CMAKE_CXX_FLAGS "-DMPICH_SKIP_MPICXX")

include_directories("../../../EXECUTIVE/HYBRID/executive/")
```

```
include_directories("../../../EXECUTIVE/HYBRID/IEL_comm/")
include_directories("../../MPICH/mpich-shared/include/")
add_library (PsuadeModule ${LIBRARY_TYPE} "Src/Main/PsuadeIEL.cpp" ${psuade_SRC}
    ${psuade_HDRS} ${PDF_SRC})
```

(The add_library command should be one line, but text wrapping makes it look like two)

1.2.2 Passing command line arguments to PSUADE

Command line arguments should be passed through the argument by the cfg file.

I added these lines to the PsuadeIEL.cpp file to get the arguments. PSUADE assumes that the first argument in argv is "/path/to/psuade/psuade", so we put a filler argument in argv[0].

```
int argc = exec_info->modules[exec_info->module_num].mod_argc + 1;

char* argv[argc];
char* temporary = "PsuadeModule";
argv[0] = temporary;
int i=0;
for(i=0; i<(argc-1); i++)
    {    argv[i+1] = exec_info->modules[exec_info->module_num].mod_argv[i];    }
```

PSUADE has four major functions - Sensitivity Analysis, Uncertainty Quantification, Optimization, and Data Analysis.

In general, PSUADE provides inputs to a simulation and compares it to the outputs from the simulation. It stores all of these results in a file, by default called psuadeData, which has all of the information about the inputs and the outputs. Generally, PSUADE will also output relevant information to the console that was requested by the user (for example, sensitivity analysis information).

Additionally, in general, if you're looking for advice about PSUADE and how to run it, launch the psuade executable and type help, then help (topic).

```
[lbrown@star1 ~]$ iel-2.0/EXTLIB/PSUADE/psuade_v1.7.2/build/bin/psuade
**************************************************************************
*      Welcome to PSUADE (version 1.7.2)
**************************************************************************
PSUADE - A Problem Solving environment for
         Uncertainty Analysis and Design Exploration (1.7.2)
(for help, enter <help>)
==========================================================================
psuade> help
Help topics:
```

```
info          (information about the use of PSUADE)
io            (file read/write commands)
stat          (basic statistics)
screen        (parameter screening commands)
rs            (response surface analysis commands)
qsa           (quantitative sensitivity analysis commands)
calibration   (Bayesian calibration/optimization commands)
plot          (commands for generating visualization plots)
setup         (commands to set up PSUADE work flow)
edit          (commands to edit sample data in loal memory)
misc          (miscellaneous commands)
advanced      (advanced analysis and control commands)
<command -h>  (help for a specific command)
```

1.3 Sensitivity Analysis / Uncertainty Quantification (SA / UQ)

For Sensitivity Analysis and Uncertainty Quantification, the code runs by generating all of the inputs at once, then running all of them in sequence. For these modes in particular, the code can actually generate all fo the inputs up front if you add "gen_inputfile_only" to the application section of the PSUADE input file.

These sorts of algorithms work by making small, measured changes in the inputs to a function/simulation, then analyzing the resulting output. The exact nature of these algorithms is unnecessary to discuss here. In general, sensitivity analysis algorithms work by making small changes in various inputs for a simulation to judge the relative strength that each input has on the output of the simulation. Uncertainty analysis tends to revolve around making small changes in the input to see how much the output is successfully determined by the inputs.

1.4 Optimization

Optimization runs slightly different. The inputs of the next iteration are determined by the output of the previous iteration, in an effort to optimize some quality of the run. For example, if you measure the error or convergence of the results, by choosing to minimize this quality, the optimization algorithm will attempt to generate inputs that minimize the error or convergence.

One thing we're interested in doing is using the optimization functions to minimize the runtimes of our codes. This has a clear purpose when running codes with a genetic algorithm or MCMC that run to convergence.

1.5 Data Analytics

PSUADE can also be used to analyze data. You can load psuadeData files in psuade by going into the PSUADE API (launching the PSUADE binary) and typing load (see

"psuade¿ help io" details). This allows you to generate plots (psuade¿ help plot), or do other forms of analysis (psuade¿ help rs or psuade¿ help qsa).

We haven't done much with this, but its inclusion as a feature of psuade seems necessary to me.

We have done several examples with PSUADE.

1.6 make test

The most notable one, in my book, has been the "make test" command. PSUADE has 14 tests included with the package. They can be accessed by going into the build directory and typing

```
make test
```

Along with any wrapper things you want to do like making it a background task or capturing the output.

Here's how it looks when I run it. Your results may vary, but should probably only vary in runtimes, not in what succeeds and what doesn't.

1.7 Rosenbrock function

We also did analysis of the rosenbrock function,

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

.

The results themselves were not particularly relevant, but it worked as a proof-of-concept.

```
PSUADE
INPUT
   dimension = 2
   variable   1 x1 =   -4.00000000e+00   4.00000000e+00
   variable   2 x2 =   -4.00000000e+00   4.00000000e+00
#####    Inputs for the example given by PSUADE    #####
##   variable   1 x1 =   4.00000000e+01   6.00000000e+01
##   variable   2 x2 =   6.70000000e+01   7.40000000e+01
##   variable   3 S =   2.00000000e+01   4.00000000e+01
END
OUTPUT
   dimension = 1
   variable   1 H
END
METHOD
   sampling = MC
```

```
   num_samples = 1000
#  for other options, consult manual
END
APPLICATION
   driver = ./simulator
#   gen_inputfile_only
#  driver = ./simulator.py
END
ANALYSIS
#  for analyzer and optimization options, consult manual
#  analyzer method = Moment
#  analyzer method = MainEffect
#  analyzer rstype = MARS
   diagnostics 1
END
END

/*************************************************
 * Main program
 *************************************************/

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char **argv)
{
   int    ii, nInputs;
   double *X, Y=1.0e35;
   FILE   *infile, *outfile;

   infile  = fopen(argv[1], "r");
   if (infile == NULL)
   {
      printf("ERROR - cannot open input file.\n");
      exit(1);
   }
   fscanf(infile, "%d", &nInputs);
   if (nInputs <= 0)
   {
      printf("ERROR - nInputs <= 0.\n");
      exit(1);
   }
   X = (double *) malloc(nInputs*sizeof(double));
```

```
   for (ii = 0; ii < nInputs; ii++)
      fscanf(infile, "%lg", &X[ii]);
   fclose(infile);

   Y = 100*(X[0] - X[1]*X[1])*(X[0] - X[1]*X[1]) - (1 - X[1])*(1 - X[1]);
    //Example generated by PSUADE
//Y = X[0] - 19.6 * X[1] / (1.5 * X[2]);
   outfile  = fopen(argv[2], "w");
   if (outfile == NULL)
   {
      printf("ERROR-cannot write to outfile.\n");
      exit(1);
   }
   fprintf(outfile, " %24.16e\n", Y);
   fclose(outfile);
}
```

1.8 The Future of PSUADE

The next step is to implement PSUADE so that it can work in parallel with other simulations and modules. We also want PSUADE to use tuple communications instead of file I/O for its work.

Fortunately, it appears we can do both at once.

```
I don't know whether this will useful : there is a psuade option called
'driver = psLocal' which will call a resident function inside psuade called
psLocalFunction. If you have a fixed function, you can compile it in psuade.
The psLocalFunction is in Src/DataIO/FunctionInterface.cpp.

If you have a fixed function that you call, you can put it
inside the psLocalFunction subroutine in Src/DataIO/FunctionInterface.cpp (that is,
replace everything inside that function with your code).
If you want to use your code every time a simulation is requested, you will
instantiate FunctionInterface and then call loadFunctionData

loadFunctionData(length, names)

and you set length = 5
and
names is a char**
with names[0] = 'PSUADE_LOCAL';
        names[1] = 'NONE'
        names[2] = 'NONE'
        names[3] = 'NONE'
```

```
        names[4] = 'NONE'

Then whenever you call FunctionInterface->evaluate, it will
call your local function.
```

It should be possible to substitute in the IEL_tget function as the 'fixed function' that will be called. Then PSUADE can treat other modules as fitness functions.

The next step will be to replace PSUADE's I/O function with IEL_tput. This shoudln't be too difficult.

`psuade_v1.7.2/Src/DataIO`

For the tput and tget changes to PSUADE, it should follow a similar process to the section in this document describing my changes to the main function.

2 R, RC

The RC module is fairly straightforward.

```
extern int R_running_as_main_program;   /* in ../unix/system.c */

//For debugging purposes, use main instead of RC, then run make executable
//int main(int ac, char **av)
int RC(IEL_exec_info_t *exec_info)
{
putenv("R_HOME=../../EXTLIB/R/inst/lib64/R");

char *a0 = "RCcar";
char *a1 = "--slave";
char *a2 = malloc(sizeof(char));

//Option to get the R script name from the command line, instead of just script.r
if(exec_info->modules[exec_info->module_num].mod_argc > 0){
char tmp[40];  strcpy (tmp, "--file=");
strcat (tmp, exec_info->modules[exec_info->module_num].mod_argv[0]);
a2 = tmp;
}

char** arguments = malloc(3*sizeof(char*));
arguments[0] = a0;
arguments[1] = a1;
arguments[2] = a2;

printf("Executing %s\n", arguments[2]);

SEXP aleph, beth;
R_running_as_main_program = 0; //May do nothing?
Rf_initialize_R(3, arguments);  //R --slave --file=(filename)
Rf_mainloop();

return 0;
}
```

Mostly what it does is set up the necessary environment, then calls the function Rf_mainloop(). The IEL configuration file will have to have the information about where the R script is. The first argument in the args section of the configuration file must be the relative path to the R script that you are interested in running. Here is a sample RCmodule configuration file that will launch a file called "script.r" that is held in the same directory as the IEL driver code.

```
shared_bc_sizes = [1];
tuple_space_size=0;

modules = (
{
function="RC";
args=("script.r");
//Recommend that the only argument is relative path to the R script, E.g. "script.r"
libtype="static";
library="libRC.a";
size=1;
points=(
    (())
    );
}
);
```

The RC module currently doesn't use any tuple communication, and does not interact with other modules in any way. Fortunately, this should be easily remedied. R can use the .C() function to call just about any C function. Converting C datatypes to R datatypes should be somewhat of a challenge, but there shouldn't be many challenges beyond that. It should be a very direct process to compile any C libraries into the R library, or compile them as a separate library to be loaded into R.

3 Sphere Packing

The sphere packing code is fairly complicated, and I don't know how well I'll be able to explain it here. The original explanation was after months of work by 6 people in a 40 minute powerpoint with a Q&A session afterwards, and it still didn't fully land. So with that in mind, here's the best way I can start to explain the sphere packing code.

The code consists of two major parts. One part is the genetic algorithm, which doesn't need much introduction. If you know genetic algorithms, you know this code. In short, a population of inputs is made, where each input is treated like a living organism. Its fitness is judged (by a given function), and the more fit individuals can mate with others to generate offspring that are a combination of the parents (plus some mutation). The whole population undergoes selection pressure, by which the weaker individuals are removd from the population while the more fit individuals continue to live on and reproduce.

The sphere packing code was originally a matlab code that we converted to C code. It is an attempt to address the question of hexagonal sphere packings on cylinders. Star with a sphere of radius R, and two integers P and Q. We want to create two strings of unit spheres, one of P unit spheres and one of Q unit spheres (imagine a string of

pearls). We then want to wrap those two strings of spheres around the cylinder such that the final two spheres in the chain have the same center, but all spheres except the last one are treated as solid objects. By playing with the radius of the sphere and the height difference between any two spheres in the second chain, we can solve for all of the other variables necessary to calculate the distance between the centers of the final two spheres. When that distance is zero (or sufficiently close to zero), our packing is complete.

The code's goal is to work towards the idea that for any P and Q, we can find a radius and height difference (called R and z) to complete the sphere packing. Of course, no amount of running the code will suffice as a proof, but it could find a counterexample that we did not anticipate.

4 Dakota

The dakota section will be a previously written document.

Logan Brown

Kwai Wong

DAKOTA as an Executable

DAKOTA is an optimization, sensitivity analysis, and uncertainty quantification package from Sandia National Labs. One of the best benefits of DAKOTA is the ability to couple it to a simulation. DAKOTA has a very large library of analysis functions that it can access in order to analyze data from a user generated simulation. DAKOTA works in a straightfoward loop.

1. Generate parameter values for a simulation
2. Run this input through the simulation
3. Read and analyze the output
4. Generate new parameter values for the simulation
    a. For optimization, condition these new parameters based off of the output from the simulation
    b. For sensitivity analysis or uncertainty quantification, base the new parameters after a given pattern or probability distribution
5. Repeat 2 – 4 for a desired number of iterations

When DAKOTA is run as an exectuable, it must perform all of this communication through writing and reading files in memory. The user must write a file for DAKOTA that details the number and types of parameters (variables) and the number of output values (objective_functions). The user must also specify information about the analysis they want to use, and any required information for that algorithm. Using this information, DAKOTA will write a parameter file with input values chosen by the algorithm. DAKOTA writes these parameters as a file which can be processed and read by the simulation. This file contains the values of the parameter variables, as well as information about the parameters themselves (such as the number of parameters), and the number of response functions. DAKOTA then calls upon the user simulation to generate a results file. This results file is processed for DAKOTA to read and analyze with one of its algorithms.

Sample Dakota Setup

```
strategy
  graphics
  tabular_graphics_data
    tabular_graphics_file = 'optimize.dat'
  single_method

method
  max_iterations = 5
  max_function_evaluations = 2000
  coliny_ea
    seed = 1
    population_size = 50
    fitness_type linear_rank
    mutation_type offset_normal
    mutation_rate 1.0
    crossover_type blend
    crossover_rate 1.0
    replacement_type chc = 10
    final_solutions = 50

variables
  continuous_design = 2
    lower_bounds      0.001        100.0
    upper_bounds      .1          1000.0
    descriptors       'radius'     "evalutations"

interface
  system
    analysis_driver = 'script'
    parameters_file = 'params.in'
    results_file    = 'results.out'

responses
  objective_functions = 1
  no_gradients
  no_hessians
```

The path to this input file is given to the DAKOTA executable. Here, I've decided to generate graphics and a data table.

Here, I've selected my method of analysis (coliny_ea), which is an evolutionary algorithm used for optimization. I also clarify some arguments for the algorithm.

Here I've chosen two parameter variables, named them, and given them bounds.

Here I set up the loop, shown in the chart on the next page.

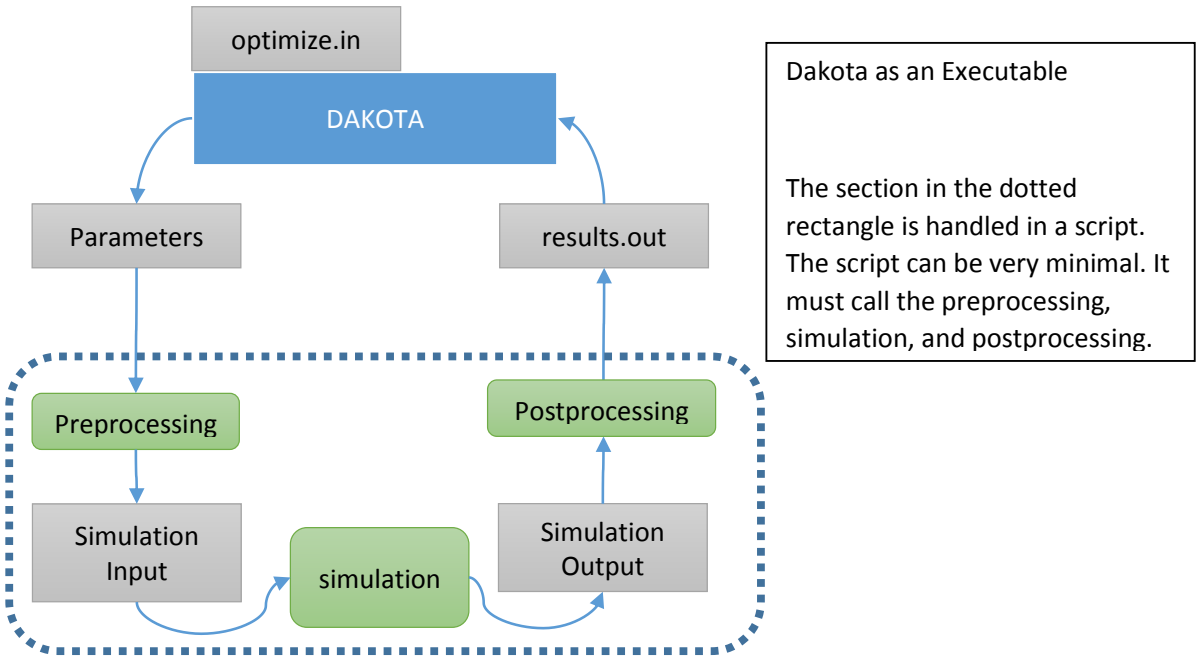
Here, I've set up what response DAKOTA expects from the simulation.

Sample Parameters File

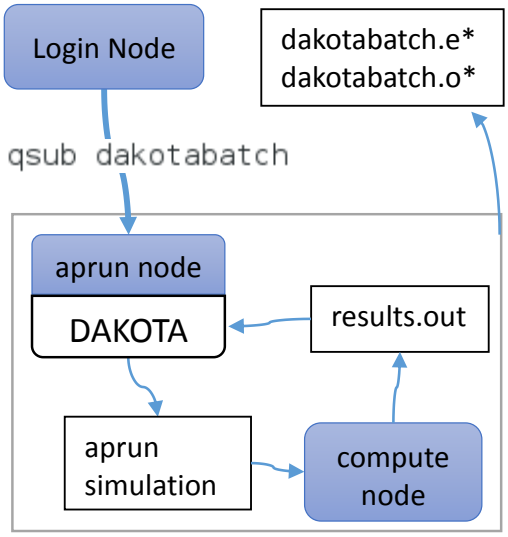
Dakota generates one of these for the simulation every loop.

```
                              2 variables
     2.000000000000000e-01 x1
     1.000000000000000e+03 x2
                              1 functions
                              1 ASV_1:response_fn_1
                              2 derivative_variables
                              1 DVV_1:x1
                              2 DVV_2:x2
                              0 analysis_components
                              1 eval_id
```

Running DAKOTA as an executable comes with some advantage. It will work on any executable, without needing the source code, so long as you can give the inputs generated by DAKOTA to the code, and give the output from the code to DAKOTA for analysis. In this way, it is versatile.



However, writing these input files and output files is slow, and bottlenecks the machine. Every analysis of DAKOTA must write four files, and most applications require several thousand analyses. Moreover, every run of DAKOTA must call 3 separate executables. This creates an additional bottleneck, especially on supercomputers such as Darter, a Cray XC30 system, because you cannot execute a program while running a program on the compute node. If we run DAKOTA's analysis on the compute node we cannot execute the simulation. Our only option is to submit the batch job, run DAKOTA on the aprun node, then use the script to run the simulation on the compute node.



Dakota as a Library

Fortunately, there is another way to run DAKOTA. DAKOTA can be compiled as a library, and the analytical functions of DAKOTA can be called as functions within the executable. This is a much more direct and efficient way to couple DAKOTA to the simulation. Database information is written into memory, rather than being written and read through files. Additionally, only one executable is called, and that executable can use the functionality of DAKOTA using specialized code. First, we must create the problem description. This can be entirely retrieved from an input file for DAKOTA, as above. However we can also choose to leave portions of it unwritten and use functionality within the executable to fill in the gaps. The input file must have the number of parameter variables and the number of response functions (objective_functions), but the algorithm, boundaries of the parameters, and other information can be defined at runtime. With a fully written input file, we can use

```
problem_db.manage_inputs(dakota_input_file);
```

Or with a partially written input file, with gaps to be filled, we can use

```
problem_db.parse(dakota_input_file);
```

The missing parts of the input file can be defined inside the executable. The code below is an example from path_to_dakota_source/src/library_mode.cpp

```
static const char default_input[] =
  " method,"
  "     optpp_q_newton"
  "       max_iterations = 50"
  "       convergence_tolerance = 1e-4"
  " variables,"
  "     continuous_design = 2"
  "       descriptors 'x1' 'x2'"
  " interface,"
  "     direct"
  "       analysis_driver = 'plugin_text_book'"
  " responses,"
  "     num_objective_functions = 1"
  "     num_nonlinear_inequality_constraints = 2"
  "     analytic_gradients"
  "     no_hessians";
 nidr_set_input_string(default_input);
```

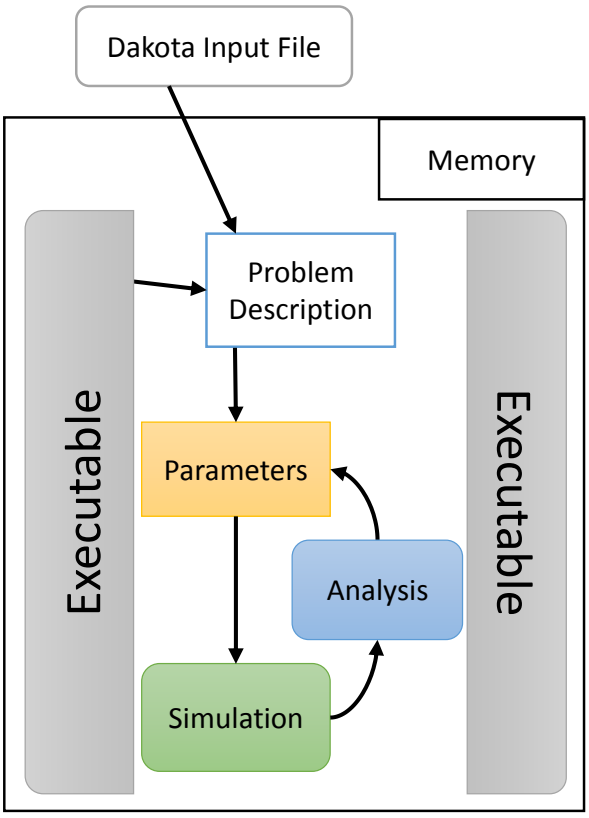
The parameters for the simulation can be generated using variables from the Dakota class and functions from the Problem_db class. For example, suppose we want to have 5 variables, with the first one initiated to 1, and the rest initated to 0. Once the input file has set up our DAKOTA implementation to have 5 variables, run

```
Dakota::RealVector vec(5, 0);
vec[0] = 1;
problem_db.set("variables.continuous_design.initial_point", vec);
```

This combination declares a DAKOTA vector of 5 real numbers, which is used to populate the database. This database will contain all of the information about the inputs for the simulation. Once the variables are initialized and the problem description is set, the strategy can be executed by

```
Dakota::Strategy selected_strategy(problem_db);
problem_db.lock();
selected_strategy.run_strategy();
```

The chart on the right summarizes this information. Everything happens inside the executable, and all information is written inside memory. The problem description (choice of analysis type such as optimization or sensitivity analysis, number of variables and their boundaries, number of output functions, etc.) is defined from a combination of the executable and the input file. The parameters are written, inside the executable, and given to the simulation. DAKOTA can then directly analyze the results, without reading or writing anything, completing the loop.

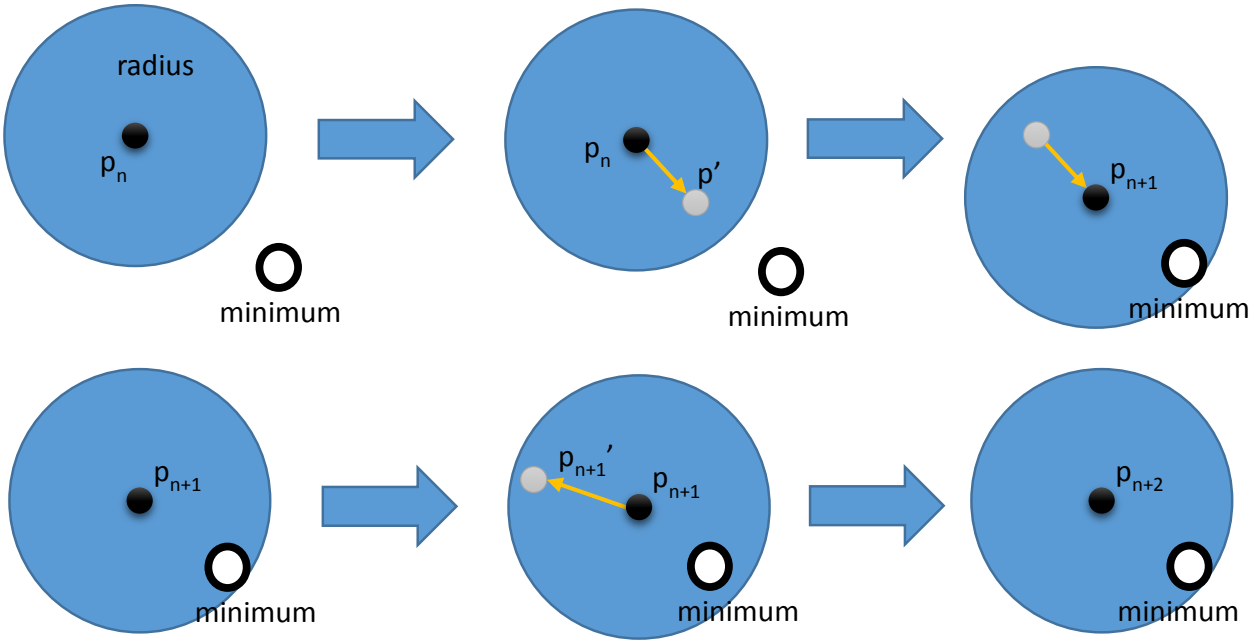


Tuple Space

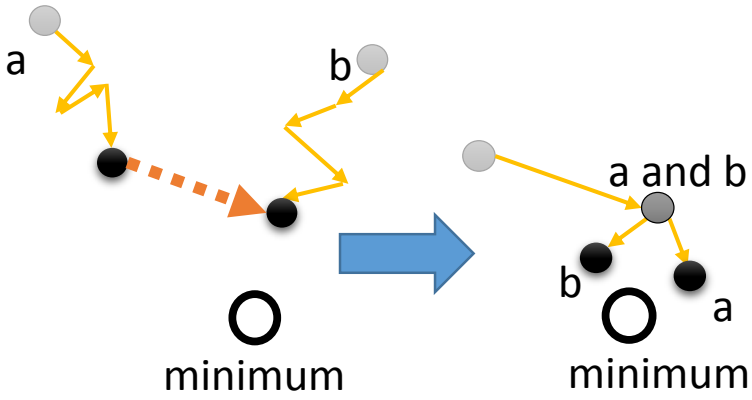
When running these large simulations, the processors often may want to communicate amongst each other. For many applications, convergence to a result can be improved by allowing communication between processors. The IEL uses Tuple Comms, instead of direct communication. This allows for asynchronous exchanges in the processors, and avoid another bottleneck.

Rosenbrock Code

To test this, I wrote a code to perform a random walk towards the minimum of the Rosenbrock function.



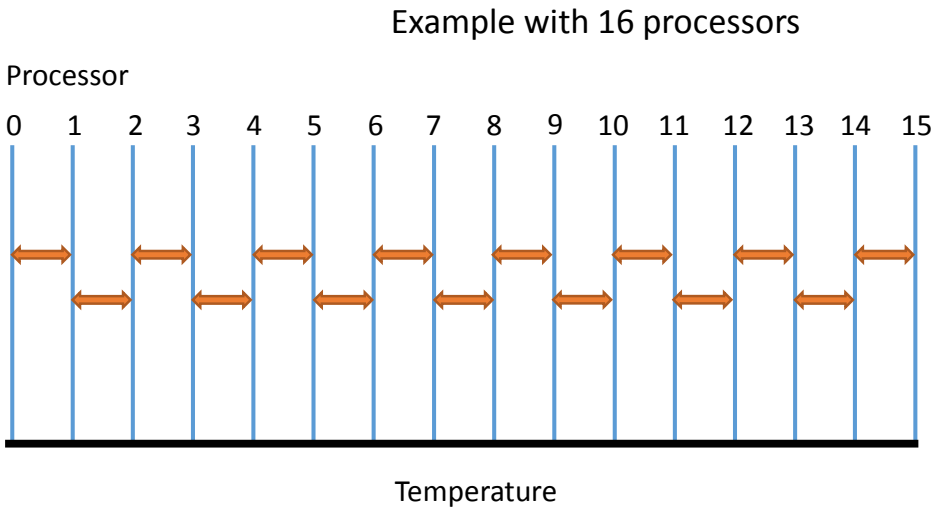
To improve convergence, I allow the processors to communicate after a given number of steps, and check other processors for a potentially better point.



Ising Model

Another example that we can improve using Tuple Comm exchanges is the Ising Model code by Siu Wun Cheung and Yiwei Zhao. The Ising Model is a phyics model using in ferromagnetism that numerically approximates difficult quantities such as Magnetism and Energy. These properties vary with respect to the temperature of the material.

The interval of the temperature is divided evenly, and each processor is given a single point. These points can perform direct exchanges to improve convergence. Above, 16 processors are used.



Every n steps, the processors will attempt an exchange with a neighboring processor. This process is repeated a number of times. On the right, the result from 0, 100, 1000, and 100000000 exchanges are shown. The total number of steps is kept constant at 10^9. 96 processors were used.

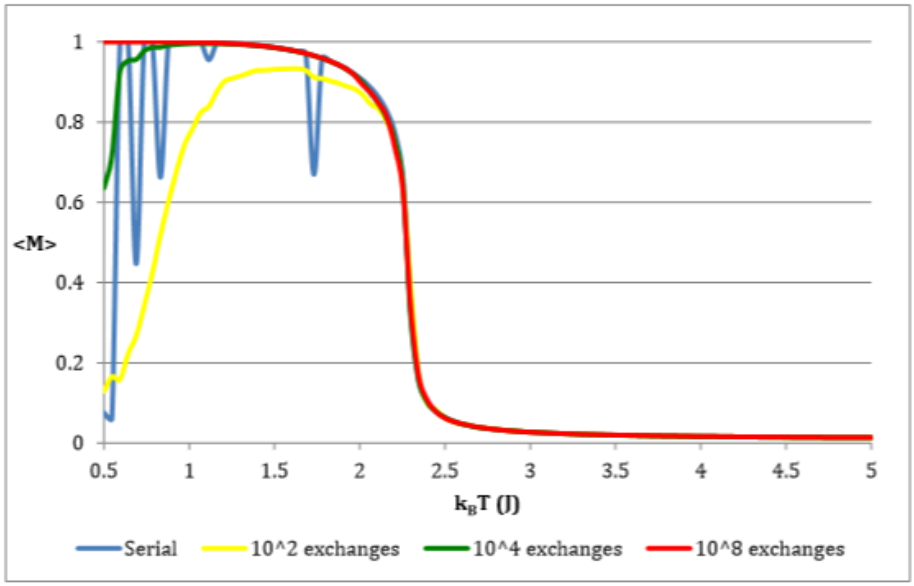


Figure 2.2: Mean magnetization per spin versus temperature under different exchange frequencies