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.

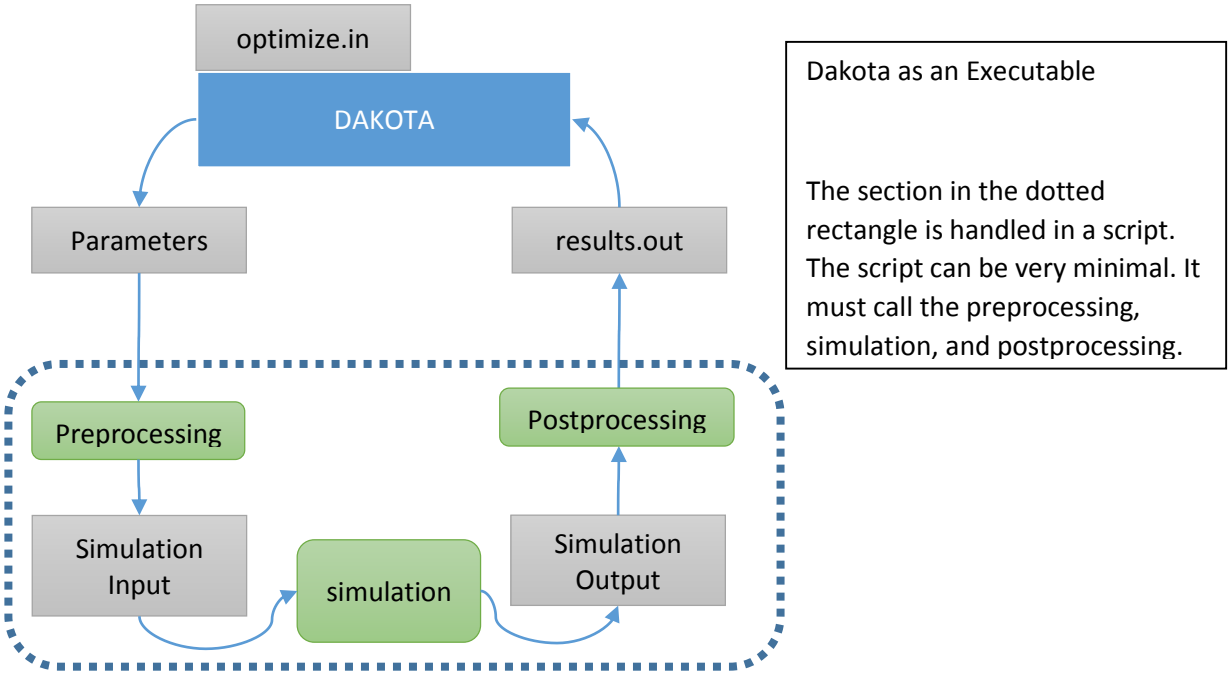
Here, I've set up what response DAKOTA expects from the simulation.

Sample Parameters File

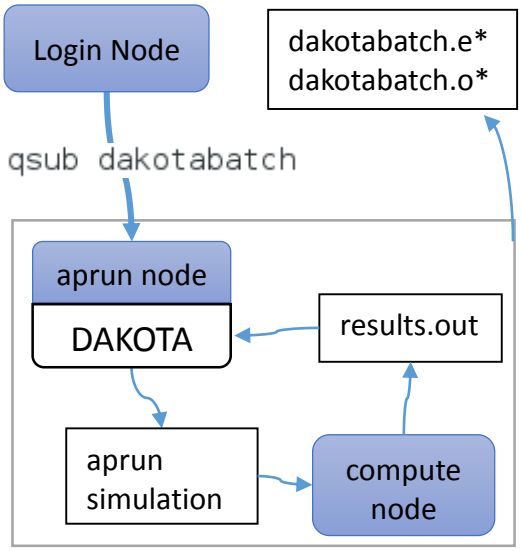
Dakota generates one of these for the simulation every loop.

```
                     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);
```

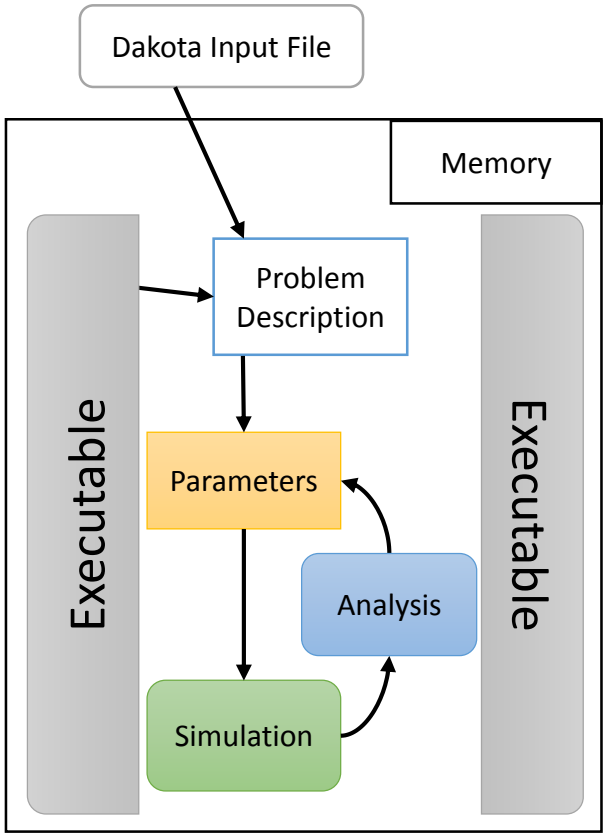
The parameters for the simulation can be generated using variables from the Dakota class and functions from the Problem_db class. For example, suppose we want to have 5 variables, with the first one initiated to 1, and the rest initated to 0. Once the input file has set up our DAKOTA implementation to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

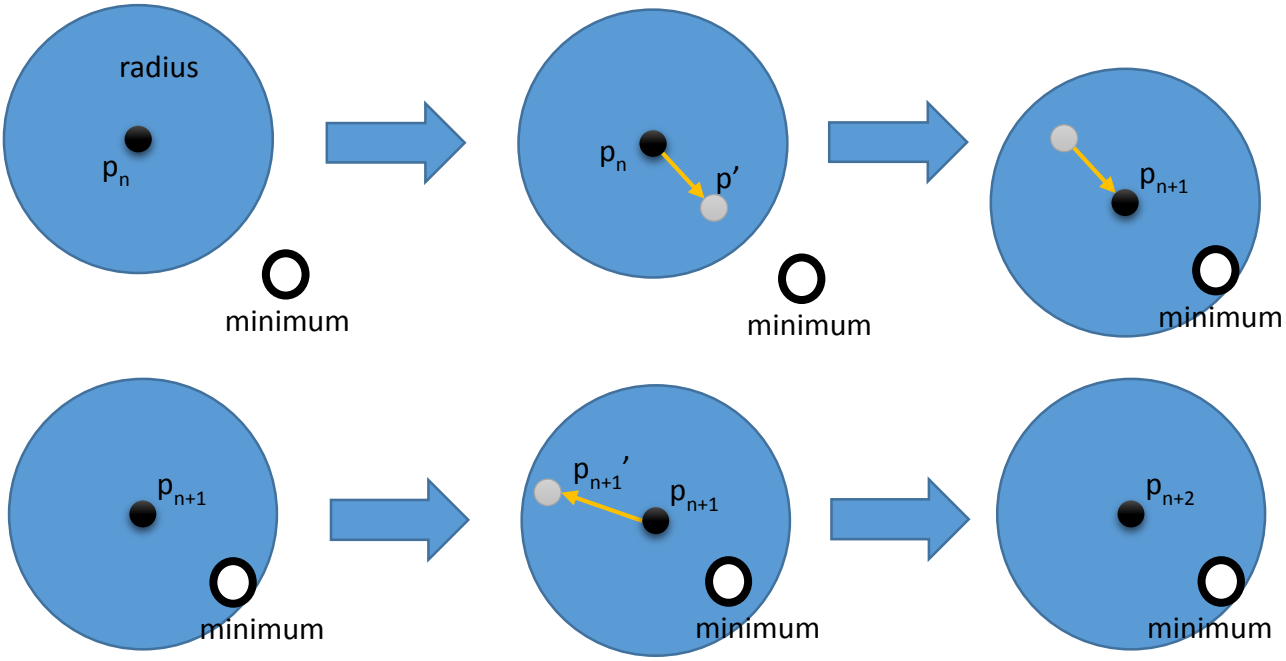


Tuple Space

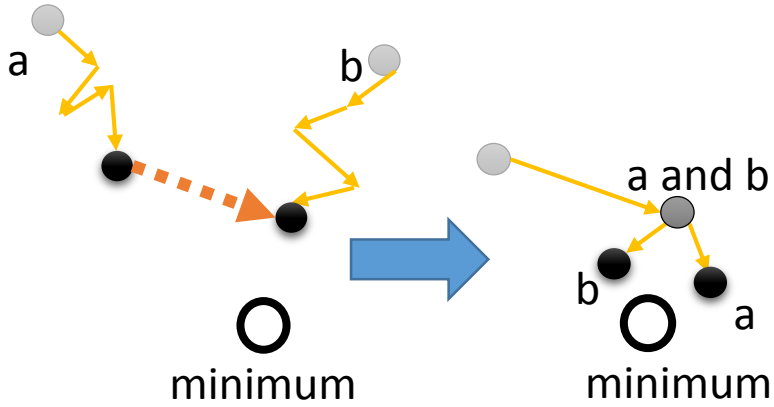
When running these large simulations, the processors often may want to communicate amongst each other. For many applications, convergence to a result can be improved by allowing communication between processors. The IEL uses Tuple Comms, instead of direct communication. This allows for asynchronous exchanges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



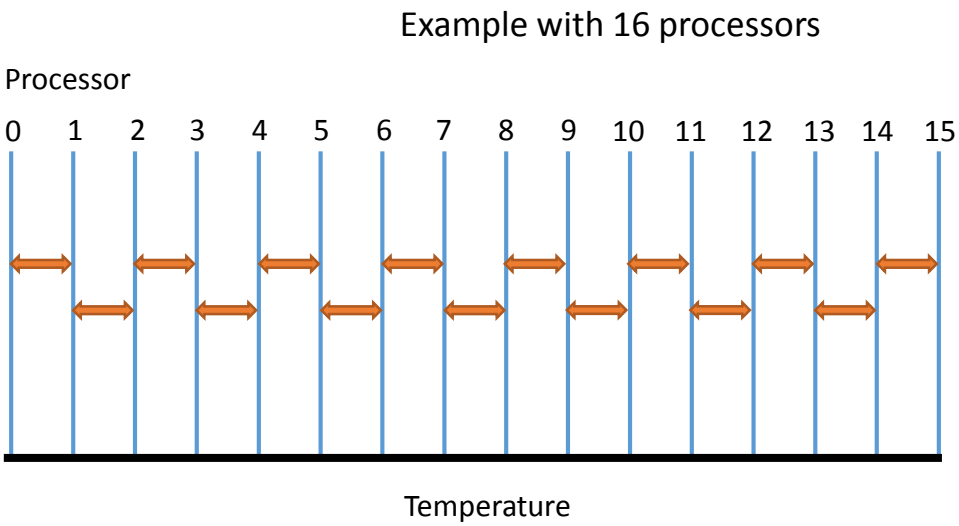
To improve convergence, I allow the processors to communicate after a given number of steps, and check other processors for a potentially better point.



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.

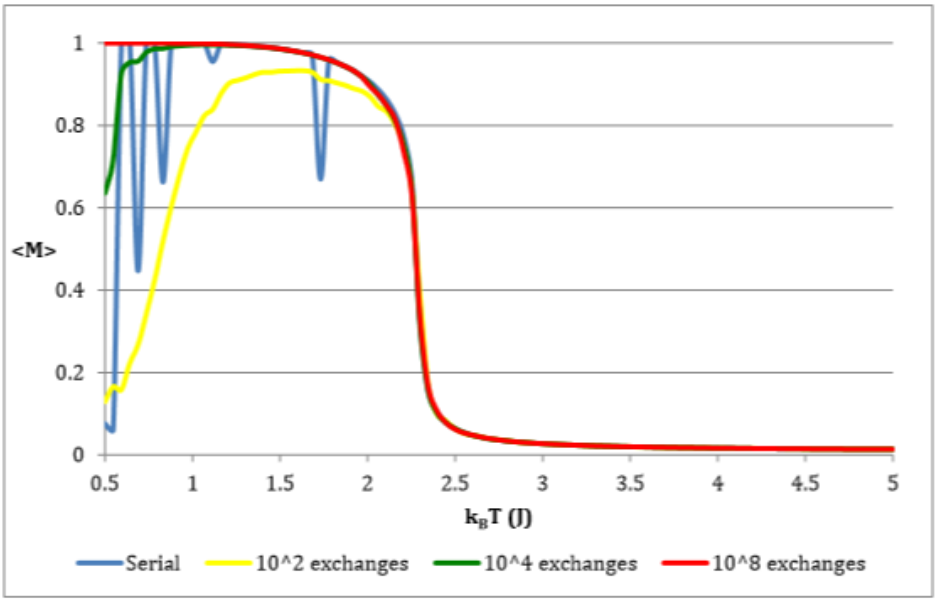
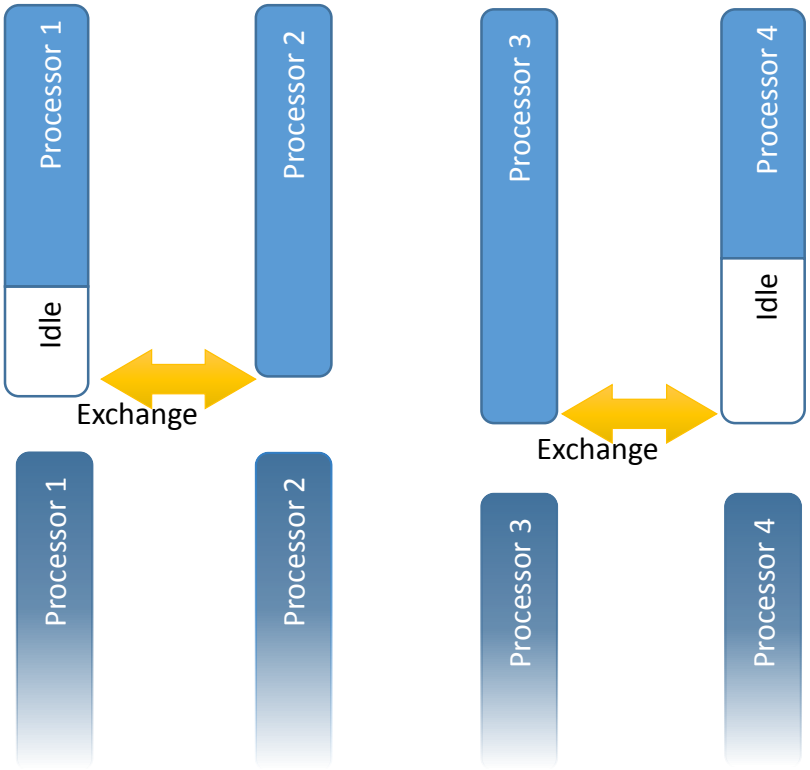


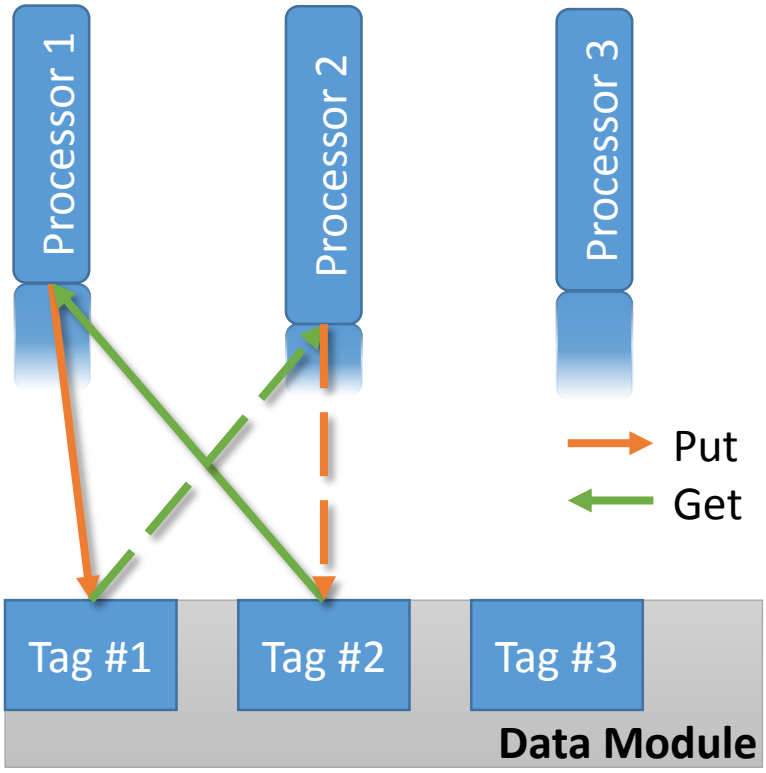
Figure 2.2: Mean magnetization per spin versus temperature under different exchange frequencies

Direct Communication Vs Tuple Comm



Direct Communication

When two processors want to exchange, as soon as one of them finishes, it must wait idly for the other to finish. This hurts performance.

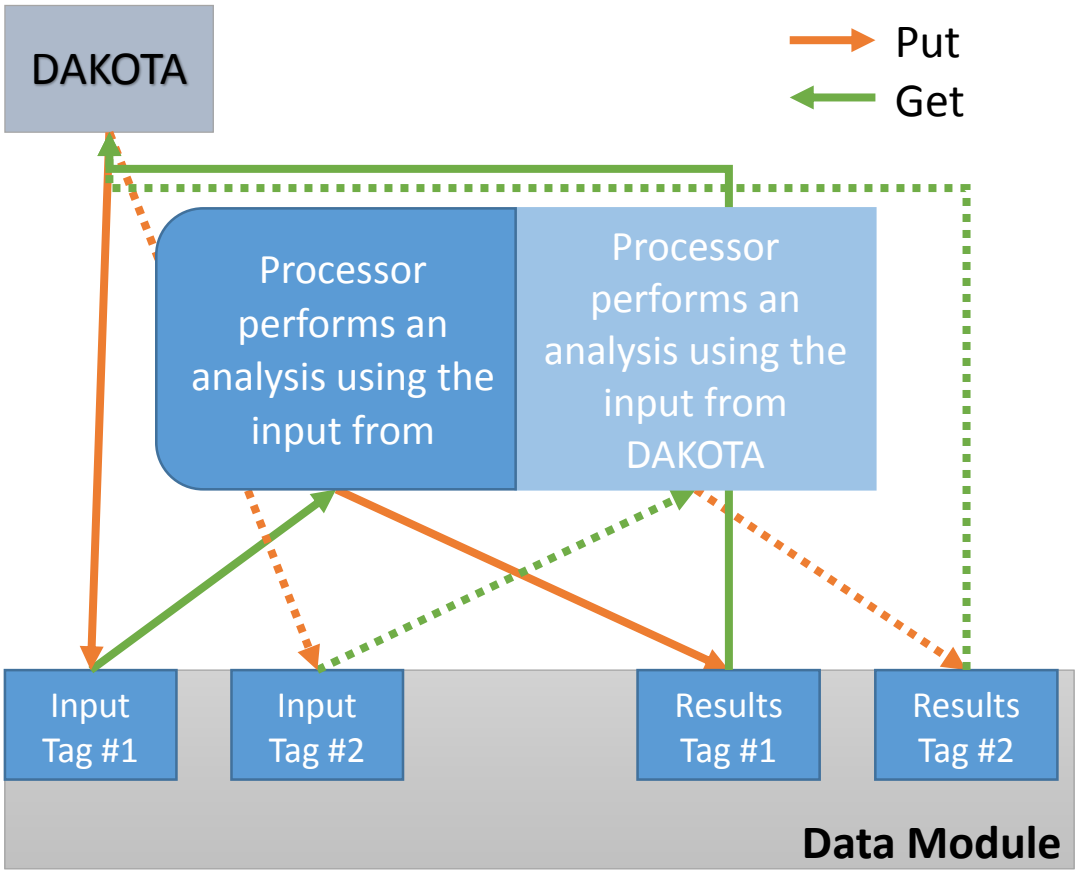


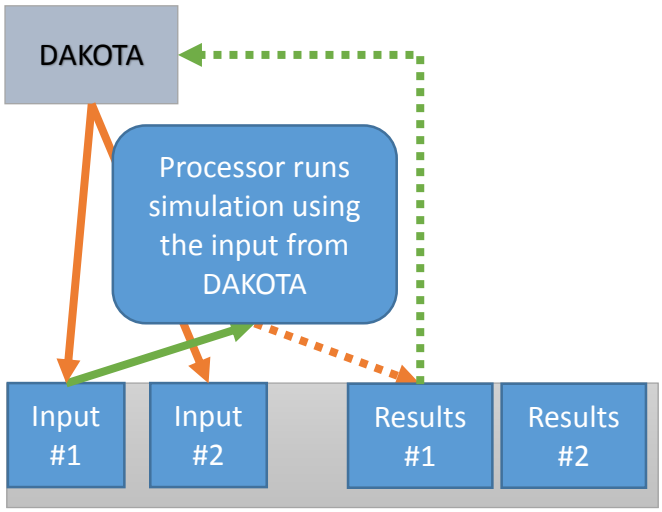
Tuple Comms

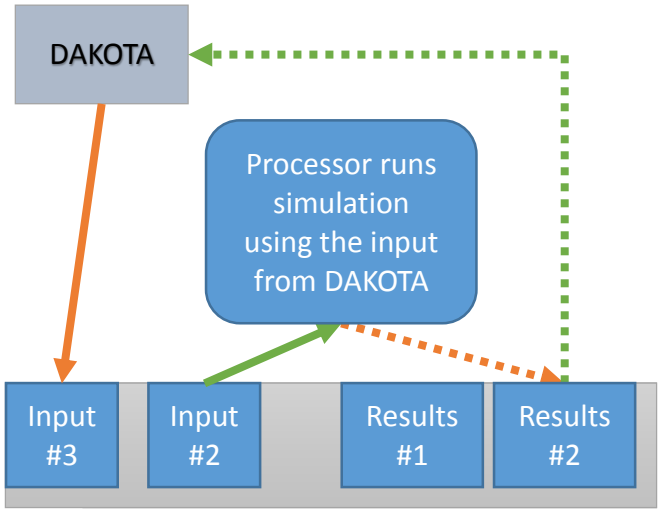
Using tuple communication, each processor puts its output information into a tagged memory spot, then can continue working. The processor does not have to wait for any other processors to finish, thus eliminating idle time.

Dakota and the Tuple Space

The Tuple Comms is also useful for DAKOTA. Dakota can put its parameter information into the tuple space, and get the information from the tuple space. We will place the parameter information into the Data Module. Then, when a processor finishes its run, it can deposit information for DAKOTA.




DAKOTA
Processor runs simulation using the input from DAKOTA
Input #1
Input #2
Results #1
Results #2


DAKOTA
Processor runs simulation using the input from DAKOTA
Input #3
Input #2
Results #1
Results #2