Leslie Matrix Parallelization Tutorial
by Matt Beasley

Model from Module 13.3, "Time after Time—Age- and Stage-Structured Models"

Introduction to Computational Science:
Modeling and Simulation for the Sciences, 2nd Edition

Angela B. Shiflet and George W. Shiflet
Wofford College

This tutorial details how to add MPI functionality to a C program that uses Leslie matrices. Each step has a corresponding spot or spots in the program *leslieMat-serial.c* where the code should be edited. These spots will be indicated by the following line(s) for each step *x*.

```
/************************ x ************************/
```

1. Include the MPI library header file.

2. Variables *iRank* and *iProcSize* have already been initialized, so initialize MPI itself and the rank and size using the appropriate MPI functions.

3. Only the root process needs to check the command line arguments, so use a conditional for the section of code that parses the command line arguments.

4. Use the MPI broadcast function to tell all the processes what the value of *statusContinue* is so that they can exit in the case of an error.

5. In the case that *statusContinue* is 0, there was an error, so use a conditional such that only the root process calls the *print_usage* function and then include the MPI function to end MPI for all processes before exiting.

6. Use a conditional so that only the root process calls the *load_values* function and then have the root process store *iSize*, *iWidth*, and *iGenerations* into the array *aiTemp*.

7. Use the MPI broadcast function to send the array that was just made to the other processes. Then, for the processes that are not the root, set the variables *iSize*, *iWidth*, and *iGenerations* to the appropriate values of *aiTemp* and use the function *malloc_double_list* to make an array *ptrValueArray* with size *iSize*. Then, use the MPI broadcast function once again so that every process has the values of *ptrValueArray*.

8. Use a conditional such that the root process is the only process to call the *query_distribution* function and then have every other process call *malloc_double_list* to create the array *ptrDistArray* with size *iWidth*. Then, use the MPI broadcast function so that every process has the values of *ptrDistArray*.

9. In the function *mat_product_mat*, which is called by *array_to_power*, the second *for* loop needs to have new parameters that are adjusted for parallelization. Since every process is calculating a different section, the values of *i* in this outermost loop need to reflect the rank of the current process. When *i* is initialized, *iWidth* needs to be multiplied by *iRank* and divided by *iSize* to obtain the correct starting value for the current process. Similarly, *i* will be incremented until it is equal to the value obtained when *iWidth* is multiplied by *iRank* + 1 and then divided by *iSize*.

10. In the function *mat_product_mat*, use the *MPI_Allreduce* function with *MPI_IN_PLACE* as the send buffer to put together the work each process did and obtain *ptrResults*.

11. After *array_to_power* is called, the root process is the only process that needs to call the *mat_product_vector* function. Use a conditional to do this as well as have all of the other processes call *malloc_double_list* to create the array *ptrResultValues* with size *iWidth*. Then, use the MPI broadcast function to send the values of *ptrResultValues* to every process.

12. Use the MPI barrier function to make sure every process has reached this point of the code before continuing.

13. Use a conditional to make sure only the root process calls the *print_results* function.

14. Call the appropriate MPI function to end MPI in the *main* function.