Distributing GARP 2.1 with Best subsets using MPI  
(the version packaged with Open Modeller 0.2.1)

What is GARP ?
Genetic Algorithm for Rule Set Production
GARP is a genetic algorithm that creates ecological niche models for species. The models describe environmental conditions under which the species should be able to maintain populations. For input, GARP uses a set of point localities where the species is known to occur and a set of geographic layers representing the environmental parameters that might limit the species' capabilities to survive.

Motivation for developing a distributed version

In the GARP with Best Subsets implementation, the user can specify the number of runs and the maximum generation per run as input parameters. Each run is then executed by taking a random sample of the environment, presence and absence occurrences of the species. The individual runs are independent of each other and therefore can be parallelized. The following are some execution times for 100 runs and varying number of generations per run. As can be seen, the time taken for just 100 runs can be very high. Hence by parallelizing this computation, users will be able to take advantage of calculating more number of runs in reasonably less time.

<table>
<thead>
<tr>
<th>Number of Runs</th>
<th>Maximum Generations(Number of iterations run by Genetic Algorithm)</th>
<th>Time taken to compute on single processor machine (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>400</td>
<td>22</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>50</td>
</tr>
<tr>
<td>100</td>
<td>10000</td>
<td>289 (~5 hours)</td>
</tr>
</tbody>
</table>

MPI (Message Passing Interface)
The message passing model is particularly advantageous due to its portable nature and the high performance when distributing tasks across a cluster of machines. Also, the master-child (root/child) configuration which is easy to code using MPI is particularly suited for this project. In this project, the C++ MPICH implementation of the MPI standard has been used.
The MPI APIs used in this project are:
1. `void MPI::Init()`
2. `int MPI::Comm::Get_rank()`
3. `int MPI::Comm::Get_size()`
4. `void MPI::Intracomm::Bcast(void *buffer, int count, const Datatype& datatype, int root)`
5. `void MPI::Intracomm::Reduce(const void *sendbuf, void *recvbuf, int count, const Datatype& datatype, const Op& op, int root)`
6. `MPI::Finalize()`
7. `MPI::Comm::Recv(void *buf, int count, const Datatype& datatype, int source, int tag, Status& status)`
8. `MPI::Comm::Send(const void *buf, int count, const Datatype& datatype, int dest, int tag)`

**Parallelization Implementation**
The configuration is of 1 master and n slave nodes.

**Algorithm for Master:**
1. Read the input configuration file.
2. Get the number of slave nodes available for computation.
3. While there are still runs to be computed, send a command to the available slave node to start a run computation.
4. Once all the slaves are busy, wait to receive results (commission error and omission error) from any slave node.
5. When results are received from a slave, store the results and if any more runs are to be computed, send a command to this slave (since it is now available) to start a new run.
6. Repeat steps 4 and 5 as long as there are runs still left to be computed.
7. When computation of runs is complete, send a message to each slave node indicating the end of computation phase.
8. Now, calculate the best subsets from all the computed runs.
9. Broadcast the best subsets ids to all the slave nodes.
10. Broadcast the sample data whose occurrence information has to be gathered from all the slave nodes.
11. Receive the sample occurrence information data from all the nodes.
12. Sum it and compute the average.
Algorithm for Slave
1. Read the input configuration file
2. Wait for a command to compute a run from the master.
3. While master sends compute run command, execute the run and send back the results (commission and omission error) to the master.
4. When master sends end of computation message, exit the loop in 3.
5. Wait for total number of best runs from the master.
6. Once the total number of best runs is received from master, wait for that number of best runs ids.
7. Find the local runs that are present in the best runs id and move them into a separate array of best runs.
8. Wait for a query from the master (sample data).
9. On receiving the sample data, compute the occurrence value for the sample data and send the result back to master.
10. Wait for the done message from the master, quit on receiving it.

Files modified for implementation:
1. src/console/om_console.cpp – master and slave code.
2. Src/console/Makefile
4. Src/inc/om_algorithm.hh
5. algorithms/garp/garp_best_subsets/best_subsets.cpp – master and slave code.
6. Algorithms/garp/garp_best_subsets/best_subsets.hh
7. algorithms/garp/garp_best_subsets/garp_run.cpp
8. algorithms/garp/garp_best_subsets/Makefile

Since there is a clash between the MPI Map and List datatype names and the ones defined in OpenModeller, we rename the ones in OpenModeller as follows:

Make changes to rename Map to OMMap in following files:
1. src/lib/control.cpp
2. src/lib/environment.cpp
3. src/lib/env_io/map.cpp
4. src/inc/env_io/map.hh
5. src/inc/om_control.hh
6. src/inc/environment.hh
7. src/console/om_viewer.cpp
8. src/swig/java/om_wrap.cxx

Make changes to rename List to OMLList in following files:
1. src/inc/file_parser.hh
2. src/inc/list.hh
3. src/inc/algorithm_factory.hh
4. src/inc/list.cpp
5. src/console/graph/graphic_x11.h
How to make the sources

1. Unzip the tarball.
2. Run the `./configure` command with the following options:
   ```
   ```
   This will generate the makefiles.
   In ROOT/src/lib/Makefile --- add the $(GDAL_CFLAGS) ... since this is required for compilation at 3 locations – line nos 466, 477, 488.
3. Run make
4. Run make install
5. Download the http://openmodeller.cria.org.br/download/rain_coolest.tgz which is an example map. Run “om_console request.txt”

Results

<table>
<thead>
<tr>
<th>Number of Runs</th>
<th>Maximum Generations (Number of iterations run by Genetic Algorithm)</th>
<th>Time taken to compute on single machine (in minutes)</th>
<th>Time taken to compute on 10 slave nodes (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>400</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>50</td>
<td>8</td>
</tr>
<tr>
<td>100</td>
<td>10000</td>
<td>289 (~5 hours)</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
<td>(job did not complete – probably took too long)</td>
<td>60</td>
</tr>
</tbody>
</table>

As can be seen from the figures, there is a considerable improvement in performance.