Parallelizing Rosetta

Almost all Rosetta applications are single threaded, and by defualt will use only a single CPU core. However, it is frequently desireable to run Rosetta on multiple CPUs, given the computational requirements of most modeling tasks. There are several ways of doing this.

Muliple processes writing to multiple files/directories

The most simplistic way of running Rosetta on multiple CPUs is to manually split your job up. In other words, if you need to generate 10000 models, you could start 100 processes which each generate 100 models. The down side of this method is that each process will generate identically named files, necessitating the use of the -out:file:prefix or -out:file:suffix options to avoid processes overwriting each others files. Additionally, if silent files are being used for output, each process must output to a different silent file.

Multiple Processes writing to one directory

If one is using pdb file output, it is possible to use the option -multiple_processes_writing_to_one_directory. If this option is specified, Rosetta will create a file called input_0001.pdb.inprogress for each structure it is generating. This makes it possible for other Rosetta processes to write into the same directory without attempting to process the same structure. The advantage of this method is that one can generate large numbers of pdb structures using multiple processes without relying on each process having a different output prefix or suffix. This option is only useful when outputting PDB files.

MPI job distribution

If you want to run Rosetta on a cluster, the most efficient way of doing so is with the MPI job distributor. When Rosetta is run on a set of CPUs using the MPI job distributor, 1 CPU is reserved as a "master" node, and the remaining CPUs are "slave nodes". The master node assigns jobs to each slave node, and manages the output of those nodes. There are several advantages to this method. The primary advantage is that the master node serves to cooridinate the output of the slaves. This means that all the slaves can write to a single silent file without corruption, which greatly decreases the load Rosetta places on the filesystem during large compute jobs. Second, because all the MPI processes are under the control of a single system, you can allocate multiple processes in a single command. This is useful on large clusters where it is important to minimize the total number of jobs. Because there is no communication between Rosetta jobs, Rosetta's performance scales linearly with the number of CPUs, and fast network connections between CPUs are not necessary for MPI to be used effectively.

Compiling Rosetta for use on an MPI cluster.

In order to use the MPI job distributor, Rosetta be compiled with the the MPI code enabled. The scons command to do this is as follows:

```
./scons.py mode=release extras=mpi bin
```

However, in order for this command two work properly, you need to do the following:

- openMPI or MPICH must be installed
- the header and include files for openMPI or MPICH (specifically, mpi.h must be in your include PATH)
- you must know the name and location of your mpicxx and mpicc compiler. These are typically in your PATH and called "mpicxx" and "mpicc" respectively, though they sometimes have other names.

The details of these prerequisites vary wildly from cluster to cluster. You should contact your cluster administrator and work with them to find the locations of these paths.

Once you have done this, you will modify the Rosetta build system. Specifically, you will modify the file main/source/tools/build/user.settings The types of modifications you will need to make are dependent on the details of your system. Reproduced below are a few examples exhibiting some of the changes you might have to make.

This is an example of the user.settings configuration necessary to compile Rosetta for use on the NICS Kraken cluster:

```
settings = {
    "user" : {
        "prepends" : {
        },
        "appends" : {
        },
        "overrides" : {
            "cxx" : "CC",
            "cc" : "cc",
            "ENV" : os.environ,

    },
        "removes" : {
        },
    }
}
```

This file is one of the most straightforward cases. The only modifications that have been made are in the "overrides" section, which redefines the c++ compiler command to "CC", and the c compiler command to "cc", which are the names of the mpicxx and mpicc compilers. Additionally, the line "ENV" :os.environ sets the internal environment of the scons build system to be equal to that of the shell, which is necessary.

A more complex case is the configuration necessary to compile rosetta on the vanderbilt ACCRE cluster:

```
settings = {
    "user" : {
        "prepends" : {
           "include_path" : [
                "/blue/csb/apps/mpich2/Linux2/1.4.1p1/x86_64/include",
                ],
           "library_path" : [
                "/blue/csb/apps/mpich2/Linux2/1.4.1p1/x86_64/lib",],

        },
        "appends" : {
            "defines"    : [ "MPICH_IGNORE_CXX_SEEK","__USE_XOPEN2K8" ],
            "flags" : {
              "link" : ["lm", "lglib-2.0",  "lmpich","fPIC"],
              },
        },
        "overrides" : {
        "cxx": "/blue/csb/apps/mpich2/Linux2/1.4.1p1/x86_64/bin/mpicxx",
        "cc": "/blue/csb/apps/mpich2/Linux2/1.4.1p1/x86_64/bin/mpicc",
        },
```

```
        "removes" : {
          "flags" : {
            "compile" : ["fno-exceptions",],
            },
        },
    }
}
```

In this case, the paths of the include and header files for MPI are specified in the "prepends" section of the configuration. Additionally, several compile flags are added. Determining what these compile flags should be will depend on your specific cluster and version of the MPI libraries. Your cluster administrators will likely be able to advise you with the specific details.