Supplementary material for "ProtTest 3: fast selection of best-fit models of protein evolution"

Diego Darriba, Guillermo L. Taboada, Ramón Doallo, David Posada

February 9, 2011

Abstract

Summary:

This appendix deals with some technical issues that were not specified in the application note for ProtTest 3: how the hybrid version of ProtTest 3 works and how the workload balancing is performed.

Availability:

ProtTest 3 source code and binaries are freely available under GNU license for download from `http://darwin.uvigo.es/software/prottest3`, linked to a Mercurial repository at Bitbucket (https://bitbucket.org/).

1 Hybrid Computation

The scalability of ProtTest 3 using either shared or distributed memory is limited by the replacement models with the highest computational load, usually the "+I+G" models, which could take up to 90% of the overall runtime. In these cases, the runtime was determined by the longest optimization, resulting in poor speedups. Moreover, the higher the number of cores, the higher the workload imbalance due to runtime differences. In fact, ProtTest 3 usually could take advantage of up to 50 cores, approximately. This important limitation prompted us to develop a hybrid (shared/distributed memory) approach, in which we reduced the overhead of the model optimizations using a thread-based executor within the distributed memory implementation.

We parallelized the basic task –ML optimization– to get rid of the limitation of using a single core per model. Thus, we modified PhyML (Guindon and Gascuel, 2003) to produce a thread-based version, using OpenMP (Dagum and Menon, 1998). In this way, the models with the highest computational load could run in parallel, significantly reducing the total runtime. However, this strategy is only possible when memory is shared, like in a cluster node, being limited by the number of available cores per system. Our solution (see Fig. 1) was to implement a message-passing based distribution of the tasks across a distributed memory machine, using MPJ Express (Shafi *et al.*, 2009). This way, it is possible to take advantage of multi-core clusters through the execution of a thread-based model optimization process together with the message-passing implementation of ProtTest-HPC. This two-level parallelism resulted in a much more efficient exploitation of the available computational resources.

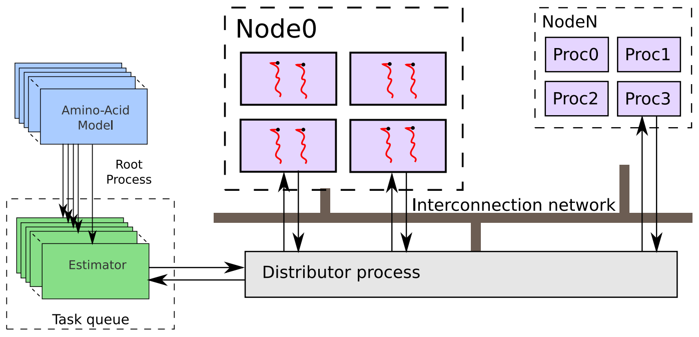

Figure 1: ProtTest 3 hybrid strategy, where two threads are run per external ML optimizer (PhyML) process.

2 Scheduling and Load Balancing

2.1 High Level Scheduling

The load balancing of the model selection task is not trivial, due to the uncertain and highly variable execution times of the model optimizations. Even the use of a dynamic task queue could not be the optimal solution in many cases.

ProtTest 3 attempts to balance the workload in two steps. At first, the workload of each single model optimization is estimated. Since the distribution is performed at a task level (i.e, the 120 models to optimize are distributed among processors), the accuracy of this estimate is essential in order to achieve the optimal performance. ProtTest 3 includes an extensible hierarchy of heuristics to perform this task. For example, the default heuristic uses the model parameters to estimate its relative weight compared to every other model.

The next step is to distribute the set of models among the computational resources, once they are sorted by their computational workload (the relative workload estimation is a good metric for this task). The best strategy is to use a dynamic scheduling of the set of tasks. ProtTest 3 implements this using a Java thread pool in shared memory architectures, and a similar approach in distributed memory. In this case, a distributor thread works as the model distributor and dynamic scheduler. The process starts sending a single model for optimization to each process, and every time a process returns the optimized model to the scheduler, it sends the next model from the sorted queue.

2.2 Hybrid Scheduling

In the hybrid-memory version of ProtTest, the number of shared memory threads is also taken into account to make the distribution among nodes. At execution time, the number of available processors cores per ProtTest 3 process is given. This implies a dependency between the logical distribution of tasks and the physical mapping of the processes (i.e., thread affinity). However, it results into the best parallel efficiency of ProtTest 3.

The scheduler calculates the number of threads that each model optimization should use to get the best overall performance, taking the number and complexity of candidate models and the number of threads per process as parameters.

Figure 2 shows the parallel performance of our OpenMP parallel version of PhyML. PhyML gets an almost linear speedup using up to 4 threads, slightly depending on the input data for a higher number of threads. With this information, ProtTest 3 can aims for the best combination between process-level and thread-level parallelism to attain the best possible performance.

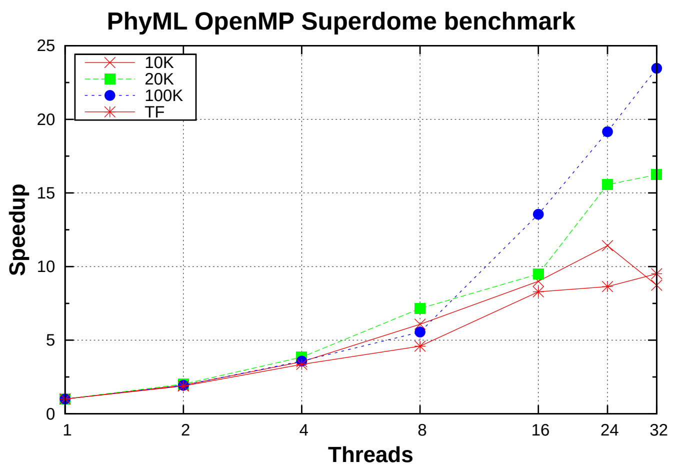

Figure 2: PhyML parallel performance on an HP Superdome system

3 Fault Tolerance

ProtTest 3 can require long execution times and involve a significant number of computing resources. For this reason we also implemented fault tolerance support. Every intermediate running status of the application is held in a serializable Java object, subject to its storage (checkpoint) in a snapshot file by the centralized checkpoint manager (*CPManager*) once it has been notified (through a custom *Observer* pattern) that a task has been completed and the ProtTest 3 status has been validated (Fig. 3). *CPManager* is also in charge of restoring

ProtTest 3 up to the last consistent saved status after a failed execution. This checkpointing system works at a high level, so the application status is verified every time a model optimization is complete.

The overhead of the fault tolerance support is almost negligible compared to the global run times ($< 0.1\%$), so it is enabled by default. Furthermore, it is fully transparent to the user. Every time the application starts, *CPManager* automatically looks up for consistent snapshot files in the snapshot directory and relaunches any previously failed execution.

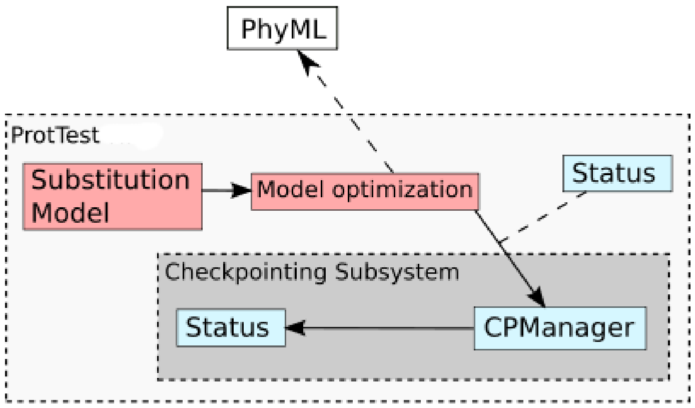

Figure 3: ProtTest 3 fault tolerance subsystem.

References

Dagum, L. and Menon, R. (1998). OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, **5**(1), 46–55.

Guindon, S. and Gascuel, O. (2003). A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Syst Biol*, **52**(5), 696–704.

Shafi, A., Carpenter, B., and Baker, M. (2009). Nested parallelism for multi-core HPC systems using Java. *J Parallel Distr Com*, **69**(6), 532–545.