International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Article

High-performance computing selection of models of DNA substitution for multicore clusters

The International Journal of High Performance Computing Applications
2014, Vol. 28(1) 112–125
© The Author(s) 2013
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342013495095
hpc.sagepub.com
SAGE

Diego Darriba[1,2], Guillermo L Taboada[1], Ramón Doallo[1] and David Posada[2]

Abstract
This paper presents the high-performance computing (HPC) support of jModelTest2, the most popular bioinformatic tool for the statistical selection of models of DNA substitution. As this can demand vast computational resources, especially in terms of processing power, jModelTest2 implements three parallel algorithms for model selection: (1) a multithreaded implementation for shared memory architectures; (2) a message-passing implementation for distributed memory architectures, such as clusters; and (3) a hybrid shared/distributed memory implementation for clusters of multicore nodes, combining the workload distribution across cluster nodes with a multithreaded model optimization within each node. The main limitation of the shared and distributed versions is the workload imbalance that generally appears when using more than 32 cores, a direct consequence of the heterogeneity in the computational cost of the evaluated models. The hybrid shared/distributed memory version overcomes this issue reducing the workload imbalance through a thread-based decomposition of the most costly model optimization tasks. The performance evaluation of this HPC application on a 40-core shared memory system and on a 528-core cluster has shown high scalability, with speedups of the multithreaded version of up to 32, and up to 257 for the hybrid shared/distributed memory implementation. This can represent a reduction in the execution time of some analyses from 4 days down to barely 20 minutes. The implementation of the three parallel execution strategies of jModelTest2 presented in this paper are available under a GPL license at http://code.google.com/jmodeltest2.

Keywords
High-performance computing (HPC), multicore cluster, Message-Passing in Java (MPJ), phylogeny, nucleotide substitution, performance evaluation

1. Introduction

In recent years, DNA sequence data has been accumulated in databases (e.g. GenBank) at an exponential rate. These DNA sequences can be used for example to study the history of the different species that inhabit our planet, for example estimating phylogenetic trees from multiple sequence alignments. All phylogenetic methods make assumptions, whether explicit or implicit, about the process of DNA substitution (Felsenstein, 1988). It is well known that the use of one or another probabilistic model of nucleotide substitution can change the outcome of the analysis (Buckley, 2002; Buckley and Cunningham, 2002; Lemmon and Moriarty, 2004), and model selection has become a routine step for the estimation of molecular phylogenies.

The most popular bioinformatic tool to select appropriate models of DNA substitution for a given DNA sequence alignment is jModelTest (Posada, 2008). This program calculates the likelihood score for each model and uses different model selection techniques to choose the "best" one according to the likelihood and number of parameters. The model selection strategies implemented in jModelTest are the Akaike information criterion (AIC) (Akaike, 1974), Bayesian information criterion (BIC) (Schwarz, 1978) and dynamic likelihood ratio tests (dLRTs) (Posada and Crandall, 2001).

[1] Computer Architecture Group, University of A Coruña, A Coruña, Spain
[2] Bioinformatics and Molecular Evolution Group, University of Vigo, Vigo, Spain

Corresponding author:
Diego Darriba, Computer Architecture Group, University of A Coruña, 15071 A Coruña, Spain.
Email: ddarriba@udc.es

Table 1. Substitution models available in jModelTest. Any of these can include a proportion of invariable sites (+*I*), rate variation among sites (+*G*), or both (+*I* + *G*).

Model	Free parameters	Base frequencies	Substitution rates	Substitution code
JC	k	Equal	AC = AG = AT = CG = CT = GT	000000
F81	k + 3	Unequal	AC = AG = AT = CG = CT = GT	000000
K80	k + 1	Equal	AC = AT = CG = GT, AG = CT	010010
HKY	k + 4	Unequal	AC = AT = CG = GT, AG = CT	010010
TrNef	k + 2	Equal	AC = AT = CG = GT, AG, CT	010020
TrN	k + 5	Unequal	AC = AT = CG = GT, AG, CT	010020
TPM1	k + 2	Equal	AC = GT, AT = CG, AG = CT	012210
TPM1uf	k + 5	Unequal	AC = GT, AT = CG, AG = CT	012210
TPM2	k + 2	Equal	AC = AT, CG = GT, AG = CT	010212
TPM2uf	k + 5	Unequal	AC = AT, CG = GT, AG = CT	010212
TPM3	k + 2	Equal	AC = CG, AT = GT, AG = CT	012012
TPM3uf	k + 5	Unequal	AC = CG, AT = GT, AG = CT	012012
TIM1ef	k + 3	Equal	AC = GT, AT = CG, AG, CT	012230
TIM1	k + 6	Unequal	AC = GT, AT = CG, AG, CT	012230
TIM2ef	k + 3	Equal	AC = AT, CG = GT, AG, CT	010232
TIM2	k + 6	Unequal	AC = AT, CG = GT, AG, CT	010232
TIM3ef	k + 3	Equal	AC = CG, AT = GT, AG, CT	012032
TIM3	k + 6	Unequal	AC = CG, AT = GT, AG, CT	012032
TVMef	k + 4	Equal	AC, AT, CG, GT, AG = CT	012314
TVM	k + 7	Unequal	AC, AT, CG, GT, AG = CT	012314
SYM	k + 5	Equal	AC, AG, AT, CG, CT, GT	012345
GTR	k + 8	Unequal	AC, AG, AT, CG, CT, GT	012345

jModelTest supports 88 submodels of the general time-reversible model (Table 1). In top of different substitution schemes and ACGT frequencies, each of these models can assume that some sites do not change between sequences (i.e. are invariant; "+I" parameter), or they do it at different rates (approximated with a discrete gamma distribution "+G"). The estimation of the α shape parameter of the gamma distribution can be complicated, and models that include this parameter ("+G" models) carry an extra computational burden.

We define the basic execution task as the optimization of a single model. jModelTest makes an extensive use of third-party bioinformatics libraries and software, aggregating multiple tasks in a pipeline and providing a high-level view of the analysis. Figure 1 shows the workflow of jModelTest, where the most time-consuming part of the process is the calculation of the likelihood scores (carried out by the PhyML program (Guindon and Gascuel, 2003)). Because this calculation represents more than 99% of the execution time in most cases, our parallel adaptation is focused in this part of the model selection process. The parallel strategies here exposed are implemented in a new version of jModelTest, available at http://code.google.com/jmodeltest2. A preliminary version of the parallelization of jModelTest, including only the shared and distributed memory versions, has been presented by Darriba et al. (2011a). This paper extends the previous work by implementing a hybrid shared/distributed memory version which overcomes the limitations of the previous work, namely the poor scalability and the workload imbalance, achieving 8 times higher performance (from speedups around 30 to speedups around 230).

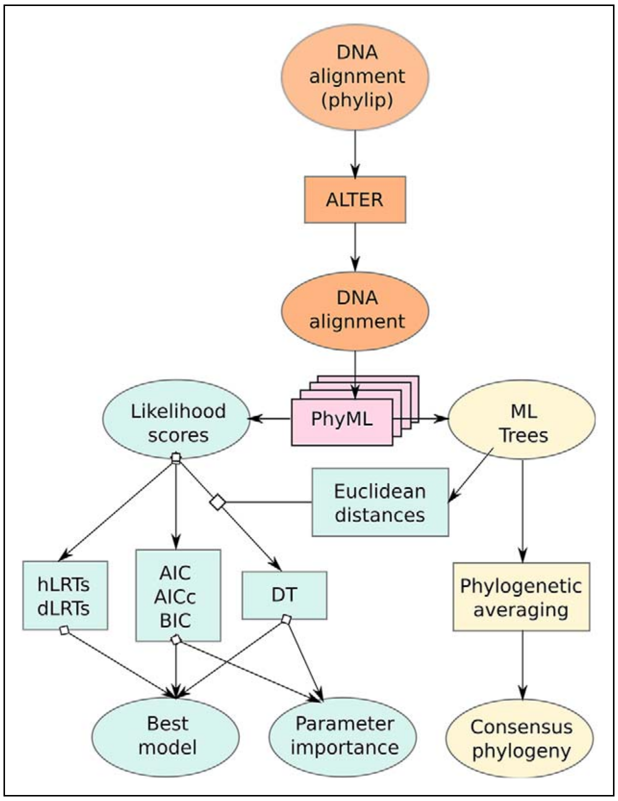

Figure 1. jModelTest algorithm workflow.

2. Parallel algorithm for model selection

Most of the execution time of the model selection analysis is spent optimizing each substitution model from the candidate model set, maximizing the likelihood function (the

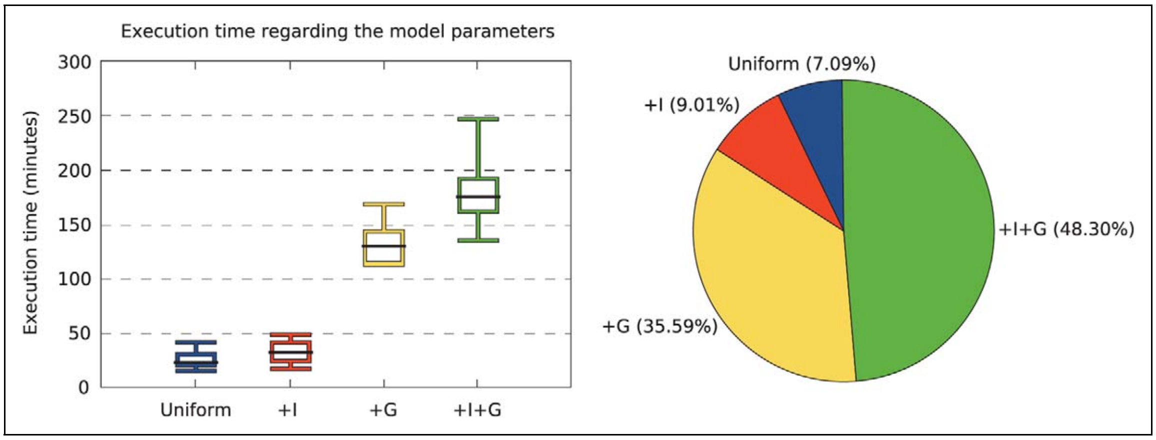

Figure 2. Computational load and execution times of the 88 model optimizations. There are 22 models of each rate variation parameter. The pie graph represents the proportion of the execution times of the models including invariant sites (+I), rate variation among sites (+G) or both (+I+G).

likelihood is the probability of the data, a multiple sequence alignment, given the model), which depends on the size and complexity of the data. For large-scale alignments this cannot be completed in a reasonable time with just a single core. Thus, we implemented a high-performance computing (HPC) version of jModelTest that supports its parallel execution on shared memory systems such as current multicore desktop processors and HPC clusters, distributing the workload among nodes and also taking advantage of multicore processors within nodes.

Maximum likelihood model optimization was proved NP-complete (Chor and Tuller, 2006). Thus, it is really difficult to estimate the runtime of each single task. However, we can estimate the relative workload depending on the model parameters. Figure 2 shows the high variance between task runtimes regarding invariant sites (+I) and discrete rate categories (+G) parameters. This variance slightly depends on the input data characteristics (e.g.number of taxa, sequences length or divergence between sequences). A representative real dataset (91 taxa and 33,148 base pairs). In order to homogeneously distribute the workload, it is better to run the most complex (i.e. +I+G and +G models) tasks at first (reverse complexity estimate). The lightest tasks would take up the remaining computational resources as long as the candidate models are optimized.

This paper presents three parallel algorithms for model selection using asynchronous communication and dynamic load-balancing: (1) a threaded shared-memory approach using Java built-in thread pool; (2) a distributed memory approach using a Message-Passing in Java (MPJ) (Shafi et al., 2009); and (3) a hybrid shared-distributed memory approach using message-passing for inter-node synchronization, a custom thread pool for intra-node synchronization, and OpenMP (Dagum and Menon, 1998) for parallelizing the basic task. The first two approaches are based on the parallel execution of model optimization tasks, presenting a coarser-grained parallelism than the last one, where the model optimization is executed in parallel as well (multilevel parallelism, with message-passing combined with shared memory thread pools and OpenMP base executions).

2.1. Design overview

The original implementation was partially redesigned to grant model extensibility, traceability and encapsulation, taking advantage of the code included in the ProtTest3 API (Darriba et al., 2011b), a similar program for protein sequences already adapted for HPC environments.

Figure 3 shows the high-level design of the HPC version of jModelTest. There is not coupling between the front-end and the back-end layers, delegating communications through a façade design pattern. Some features were organized into a class hierarchy, decoupling the related classes from the controller (i.e. ModelTestService), therefore making the model easier to extend through several interfaces:

1. The execution modes use the *RunPhyml* hierarchy. A common interface hides the model optimization behavior, and internally is able to run several PhyML instances in a shared memory architecture using a thread pool (*RunPhymlThreaded*), synchronize several processes in a distributed memory architecture (*RunPhymlMPJ*) or synchronize multiple thread pools in different nodes (*RunPhymlHybrid*).
2. The model selection task can be performed applying different information criteria that in general terms behaves in the same way. For this reason a common specification (*InformationCriterion*) hides each single criterion. As before, this decouples these classes from the controller, and also brings extensibility to the architecture.
3. The view classes do not directly depend on the inner model. The use cases are implemented in the main application service. Using an observer design pattern the execution information is displayed in real time. This works this way not only for the graphical user interface (GUI), but also for the command

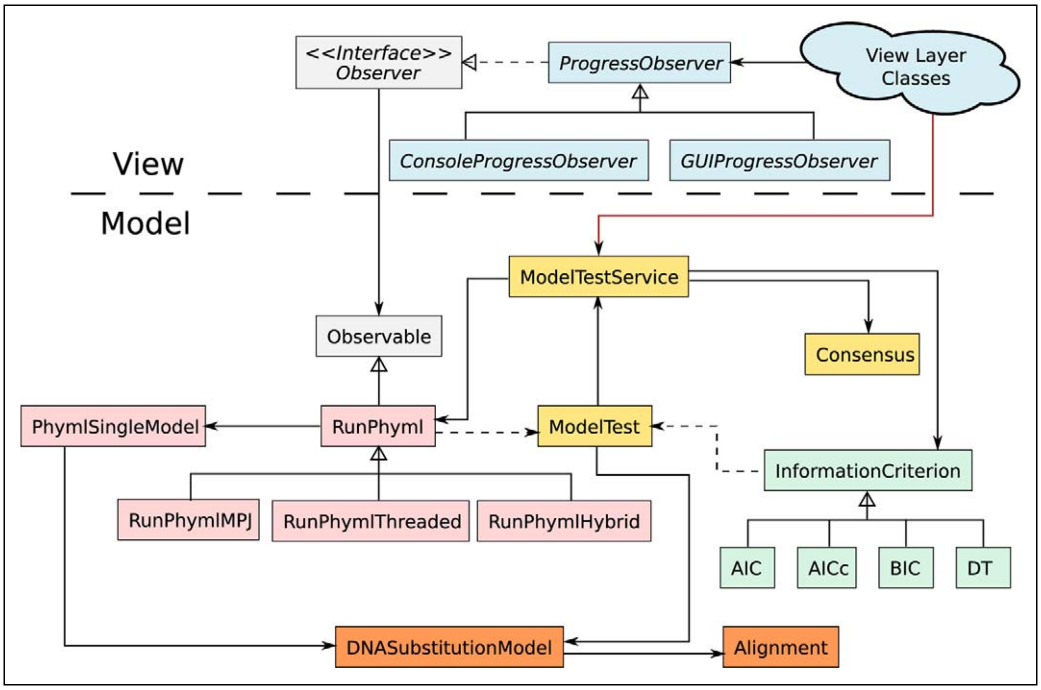

Figure 3. High level design of jModelTest2.

console executions, both threaded and distributed. In the distributed approach, only the root model is in charge of the I/O operations, unless they mean read/write data from/into scratch.

2.2. Shared memory implementation

The shared memory implementation of jModelTest relies on a thread pool to handle the execution of tasks on shared memory architectures. This approach is fully portable as it relies on thread pools from the Java Concurrence API, present in Java 1.5 and higher. Figure 4 and Algorithm 1 present the shared memory parallel operation. The task queue contains the whole set of tasks which will be processed by the thread pool in a particular order (reverse complexity estimate) (Figure 5(a)).

This multithreaded shared memory approach is especially suitable for the parallel execution of jModelTest2 on multicore desktop computers, benefiting also from the availability of a GUI. However, it is limited by the number of available cores in the system, and especially from the memory consumption, directly proportional to the number of cores being used.

2.3. Distributed memory implementation

In order to handle the computation of tasks on distributed memory architectures (e.g. clusters), this implementation manages processes, which rely on message-passing for communication, and uses a dedicated distributor thread to allocate the workload according to a dynamic distribution strategy (Figure 5(b)).

The process synchronization is explicitly achieved through message-passing blocking communication. The models are sequentially distributed and gathered among the processes through non-blocking communications. Only the root process is in charge of I/O, centralizing the displaying of runtime information and results. This central management has a negligible impact on the whole performance as long as there is no output during the model optimization task, which is by far the most time-consuming part of the model selection process. Every process works with its own copy of the input alignment from scratch, avoiding this way read or write conflicts. Figure 6 and Algorithm 2 show the operation in a distributed memory environment.

This approach saves the previous bottleneck with memory requirements. However, because each single task is executed sequentially, the workload imbalance in the model optimization tasks (see Figure 2) represents a new bottleneck. In fact, in most cases half of the candidate models requires more than 80% of the total execution time. The more computational resources are used, the more probably is that the total execution time depends on the optimization of a single model, the one which takes longer to optimize. This way, looking at Figure 2 it can be seen that the maximum runtime of a single model optimization is 248 minutes. The total runtime of the dataset is 135 hours. Thus, dividing the total runtime by the maximum single task runtime we can estimate that the highest speedup achievable is 32.83, no matter how many processes are used. Although this is a particular example, empirical tests show that the workload imbalance usually leads to a maximum speedup below 40 for these task-level parallel strategies.

```
Data: Execution parameters, input data, resource information
Result: Best-fit model
begin Application initialization                                    // tasks initialization
    build(model set);
    Estimate workload per model;
    Sort models in reverse complexity estimate;
    begin Initialize parallel environment
        Initialize thread pool;
        Synchronize thread pool;
    end
end
begin Tasks Computation                                             // model optimization
    build(task queue);
    foreach model optimization task do
        Wait for an idle thread;
        Assign the next task to the thread;
        Optimize model;
    end
end
```

Algorithm 1. jModelTest algorithm for shared memory parallel model optimization.

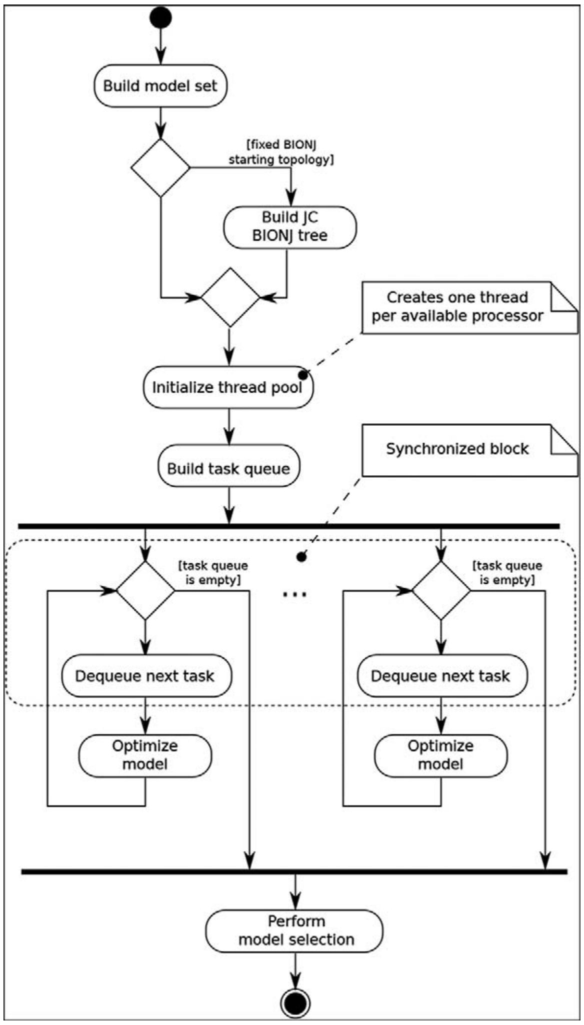

Figure 4. Activity diagram for the shared memory parallel operation algorithm.

$$Speedup_{max} = \frac{Runtime}{Runtime_{max}}$$

2.4. Parallel model optimization implementation

A detailed knowledge of the performance of the PhyML parallel implementation is key for an optimal assignment of resources (processor cores) for each model optimization task. As the +G and +I+G models used to take four times longer to optimize than the rest of the models, a proportion of four-to-one in relative speedups would balance the workload for each task. However, since the parallel efficiency of the PhyML parallel implementation is not optimal a trade-off between the expected speedup and the available computational resources has to be considered.

PhyML uses the Maximum Likelihood algorithm (ML) (Felsenstein, 1981) for finding the model parameters that maximize the likelihood function. The likelihood evaluation algorithm is the most time-consuming part of the ML process, because it is executed for each new model state proposal (i.e. after changing the parameters configuration, the tree topology or the branch lengths). This likelihood evaluation algorithm is highly parallelizable, since it is a site-independent operation performed all along the column patterns in the alignment. We have slightly changed the source code fixing unnecessary carried dependencies in order to parallelize this loop using OpenMP. Further than the source code analysis, the results of the parallel PhyML have been thoroughly validated. The sources of this parallel patch are available from the authors.

Figure 8 shows the performance of this OpenMP-based parallel PhyML version, where the scalability is notably higher for models with rate variation among sites (+G). In these models, the likelihood evaluation for each site is

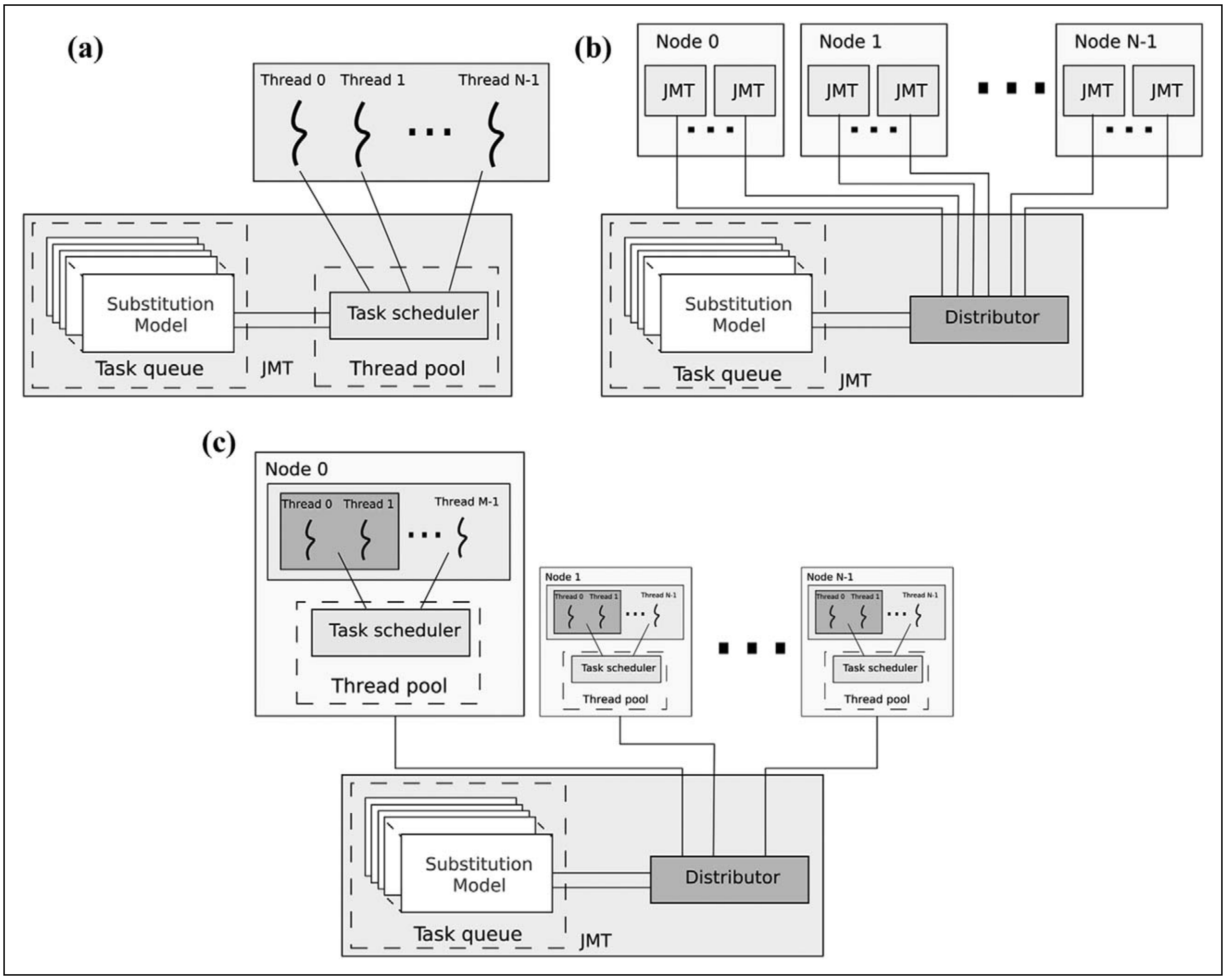

Figure 5. Parallel execution strategies in (a) shared memory, (b) distributed memory and (c) hybrid shared/distributed memory.

repeated for each different rate category (typically four categories are used).

The parallel section here represents around 75% of runtime for single category models and 98% of the total execution time for models with rate variation among sites. Amdahl's law states an expected speedup of 2.91 and 7.02, respectively, using 8 threads, and 3.36 and 12.31, respectively, using 16 threads. However, there is a high parallel overhead as long as the consumed execution time is caused by a large number of sequential repetitions of the call to this function and not so by the computational load of this loop itself.

Looking at these results, it is important to select the best ratio of resources allocated to +G and +I+G models regarding non-gamma models. For example, a four-to-one ratio is expected to balance the workload of the tasks for four- and eight-core nodes. In addition, using a number of threads that is a divisor of the number of available cores per node will maximize the number of +G and +I+G models that can be optimized in parallel. For example, an efficient allocation rule for a cluster of 12-core nodes would be the use of 4 or 6 threads for each gamma model (+G and +I+G) and a single thread for the rest (uniform rates and +I).

2.5. Hybrid shared/distributed memory implementation

The performance limitations of the previous implementations can be solved using the previous parallel implementation of the basic task for avoiding workload imbalance, and relying on a distributed memory approach to cope with memory limitations, thus implementing a three-level hybrid shared/distributed memory implementation (Figure 5(c)). Figure 9 and Algorithm 3 show its operation in a hybrid shared/distributed memory environment, such as a cluster of multicore nodes. A single jModelTest process is executed for each node, containing a custom thread pool implementation that manages the tasks that are executed within the node, and the number of cores allocated for each task (i.e.the number of OpenMP threads used for the model optimization task). This distribution, known as "*thread scheduling*", allows different amounts of computational resources to be assigned depending on the estimated

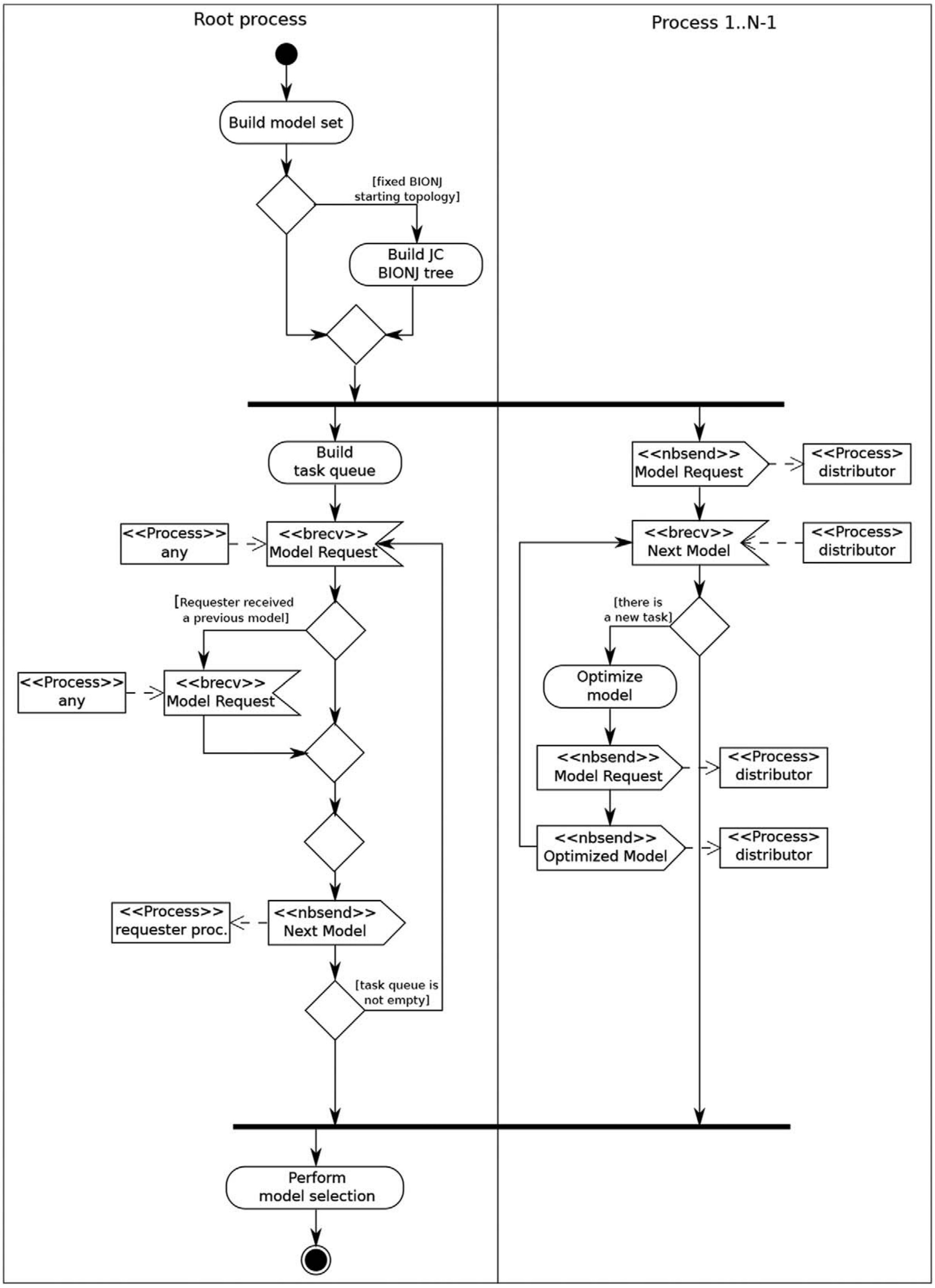

Figure 6. Activity diagram for the distributed memory parallel operation algorithm.

```
Data: Execution parameters, input data, resource information
Result: Best-fit model
begin Application initialization                                // tasks initialization
    build(model set);
    if Master process then
        Estimate workload per model;
        Sort models in reverse complexity estimate;
        build(task queue);
    else
        Wait for task;
    end
end
begin Tasks Computation                                         // model optimization
    if Master process then
        while There are active workers do
            Wait for task request;
            if Worker had a previous model then
                Receive results from previous model;
                Update execution progress;
            end
            if Task queue is not empty then
                Send the next task to the requester worker;
            else
                Send termination message to the worker;
            end
        end
    else
        Send task request;
        while There are tasks to execute do
            Receive task;
            Execute task;
            Send task request;
            Send previous task results;
        end
        Finalize;
    end
end
```

Algorithm 2. jModelTest algorithm for distributed memory parallel model optimization.

$$\boxed{Speedup_{max} = \frac{Runtime}{Runtime_{max}}}$$

Figure 7. Maximum reachable speedup using task-level parallelization.

workload of each task. Therefore, the parallel efficiency is maximized in the simplest models by using a reduced number of cores (one or two), while the heaviest tasks are split, thus balancing the global workload.

Parallel synchronization between nodes is performed using MPJ, as in the previous distributed memory version. Each MPJ process uses a custom implementation of the thread pool, where both tasks and cores per task are managed. Thus, the model optimization tasks can be heterogeneously distributed among the total number of cores within the node. The workload is decomposed relying on our custom parallel PhyML version implemented with OpenMP.

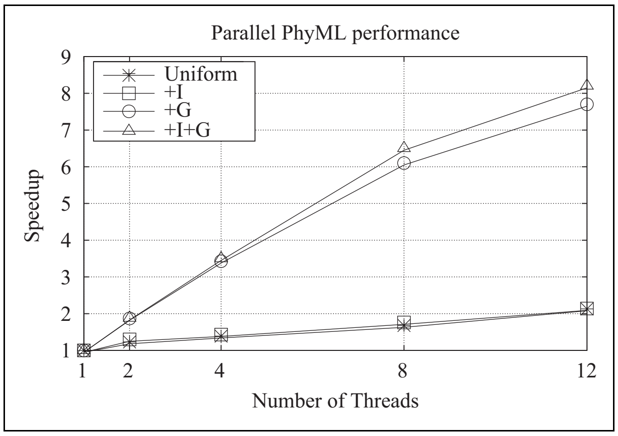

Figure 8. Parallel PhyML performance.

3. Performance evaluation

The performance of the three parallel algorithms for model selection of jModelTest2 has been evaluated on two

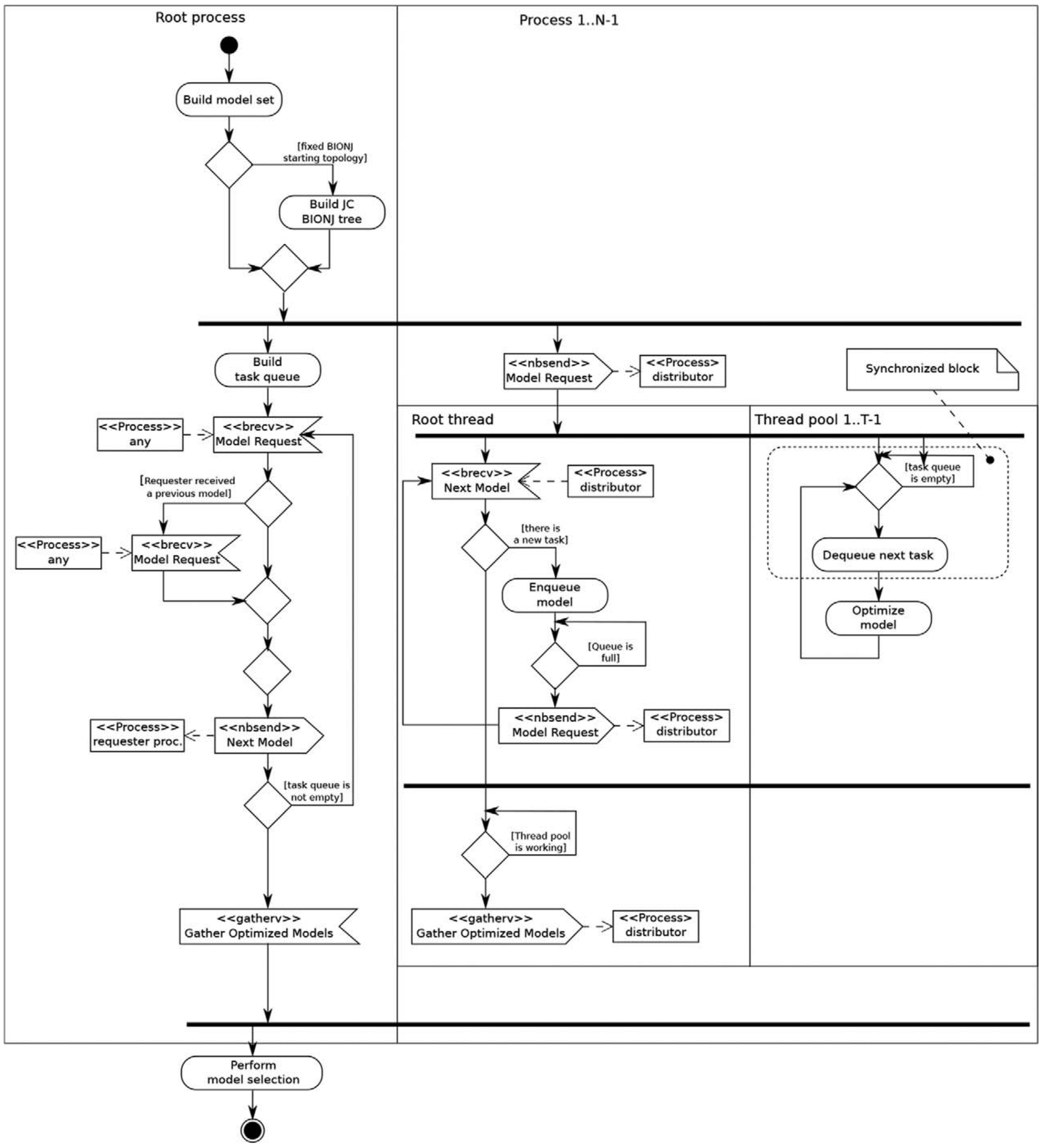

Figure 9. Activity diagram for the hybrid shared/distributed memory parallel operation algorithm.

representative HPC systems, a 40-core shared memory system and a cluster of multicore nodes (44 nodes with 12 cores per node, hence 528 cores). The distributed memory and the hybrid shared/distributed memory algorithm have been evaluated in this latter system.

3.1. Data set configuration

The data sets used in the performance evaluation consist of 4 simulated multiple sequence alignments, covering a wide range of number of sequences, from 12 to 91, and a wide range of sites for each sequence, from 3000 to 33,148 (see Table 2). The largest alignment, ALIGN1 (91 sequences of 33,148 sites), has the largest sequential runtime (5.65 days) whereas the sequential execution time of the smallest one, ALIGN4 (12 sequences of 5831 sites), is around 5 hours.

The calculation of the model likelihood scores requires an initial phylogenetic tree (“base” tree), generated using likelihood estimation (ML) (Felsenstein, 1981). This algorithm provides much more accurate results in the model selection process in exchange of significantly higher runtimes than other algorithms such as BIONJ (Gascuel,

```
Data: Execution parameters, input data, resource information
Result: Best-fit model
begin Application initialization                                    // tasks initialization
    build(model set);
    if Master process then
        Estimate workload per model;
        Set the required threads per task;
        Sort models in reverse complexity estimate;
        build(task queue);
    else
        begin Initialize parallel environment
            Initialize custom thread pool;
        end
        Wait for tasks;
    end
end
begin Tasks Computation                                              // model optimization
    if Master process then
        while There are active workers do
            Receive task request and number of idle threads on the requester pool;
            if Task queue is not empty then
                Select the most complex task that requires at most the available threads;
                Send the next task to the requester worker;
            else
                Send termination message to the worker;
            end
        end
    else
        while There are tasks to execute do
            while There are idle threads do
                Send task request and number of idle threads;
                Receive task;
                Execute task in parallel asynchronously;
            end
            Wait for idle threads;
        end
        Finalize;
    end
    Gather selection results at the Master process;
end
```

Algorithm 3. jModelTest algorithm for hybrid shared/distributed memory model optimization.

Table 2. Data sets used in the performance evaluation.

Name	Number of sequences	Length	Base Tree	Sequential runtime (hh:mm:ss)
ALIGN1	91	33,148	ML	135:42:01
ALIGN2	40	4,203	ML	15:23:56
ALIGN3	40	3,200	ML	14:33:48
ALIGN4	12	5,831	ML	4:51:13

1997). ML estimation is NP-complete, while BIONJ has a computational complexity of $O(n^3)$, where n is the number of sequences.

The distribution of the tasks is limited by the maximum number of substitution models to be computed, 88 in this case, so it is not possible to take advantage from the use of more than 88 processes. Moreover, the variability of the runtimes across models has a significant impact on performance as workload imbalance reduces the speedup and the parallel efficiency obtained. Thus, Figure 2 presents the overhead of the model likelihood calculation of ALIGN1 for each model type. In this case, when a small number of models per processor is distributed the differences in execution times can delay significantly the completion of the parallel execution.

Furthermore, even the execution times of the optimization of the models with the same parameters (e.g. "+I" and "+I+G" models) present significant variance. This characteristic, together with the fact that their execution time cannot be estimated a priori, contribute to the presence of a performance bottleneck as the number of cores increases (i.e. the fewer models per processor, the less probability the work is balanced). In order to reduce this overhead a heuristic, which consists of starting the optimization with the most complex models, has been proposed. This approach has reported more balanced executions as the main source of imbalance, starting the computation of a complex model at the end of the optimization process, is avoided.

However, the scalability using the shared and distributed memory implementations is limited by the replacement models with the largest execution time, which can account for more than 80% of the overall runtime. In these cases, the runtime is determined by the longest optimization, even if the execution is prioritized efficiently using the proposed heuristic for selecting the models to be optimized first.

3.2. Testbed configuration

The shared memory testbed is a system with 4 Westmere-EX (Westmere-based EXpandable/multiprocessor) Intel Xeon E7-4850@2.0 GHz 10-core processors (hence, 40 cores) and 512 GB memory. The OS is Linux Ubuntu 11.10 64 bits, the C compiler is gcc 4.6 and the JVM is OpenJDK 1.6.0_23 64-bit Server VM.

The second testbed is a cluster of multicore nodes, used for the evaluation of the distributed and hybrid shared/distributed memory implementation. This cluster is also a Westmere-based system, Westmere-EP (Efficient Performance), which consists of 44 nodes, each of them with 2 Intel Xeon X5675@3.06 GHz hexa-core processors (hence 12 cores per node, 528 cores in the cluster) and 24 GB of RAM (1104 GB of RAM in the cluster). The interconnection networks are InfiniBand (DDR 4X: 16 Gbps of maximum theoretical bandwidth), with OFED driver 1.5.3, and Gigabit Ethernet (1 Gbps). The OS is Linux CentOS 5.3, the C compiler is gcc 4.6, the JVM is Oracle 1.6.0_23, and the Java message-passing library is FastMPJ 1.0b.

The performance metrics considered in this performance evaluation are the execution time and its associated speedup, defined as $Speedup(n) = T_{seq}/T_n$, where T_{seq} is the runtime of the sequential execution of jModelTest2, and T_n the time measured when using n cores. Another metric considered is the parallel efficiency, defined as $Efficiency(n) = Speedup(n)/n$, which is 100% in the case of a linear speedup (speedup of n when using n cores) and is close to 0% in case of highly inefficient parallel executions.

3.3. Evaluation of the shared memory algorithm

Figure 10 and Table 3 present, respectively, the speedups and execution times obtained using the shared memory implementation in the 40-core Westmere-EX shared memory system. In this scenario the speedup is close to the ideal case (i.e. obtaining speedups around n with n cores), especially using up to 24 threads. The use of a higher number of threads (32 and 40) results in workload imbalance due to the reduced number of models optimized per thread, which makes it more difficult to balance the workload even with this dynamic distribution.

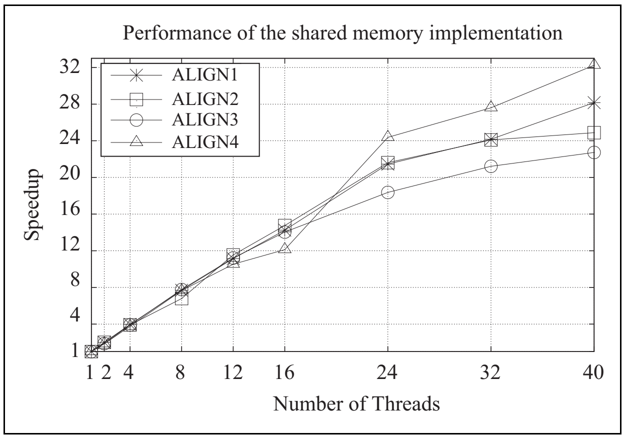

Figure 10. Scalability of the shared memory version (40-core Westmere-EX testbed).

Table 3. Execution times (hh:mm:ss) in a shared memory system (40-core Westmere-EX testbed).

Threads	ALIGN1	ALIGN2	ALIGN3	ALIGN4
1	135:42:01	15:23:56	14:33:48	04:51:13
2	66:11:01	07:39:52	07:45:53	02:37:45
4	34:14:08	03:56:18	03:43:59	01:17:28
8	17:39:16	02:16:03	01:52:30	00:38:01
12	12:10:04	01:20:03	01:17:56	00:27:40
16	09:31:29	01:02:39	01:02:13	00:24:02
24	06:19:31	00:42:45	00:47:34	00:11:57
32	05:36:53	00:38:21	00:41:11	00:10:33
40	04:49:08	00:37:08	00:38:27	00:09:02

The processors of this testbed have simultaneous multithreading (SMT) or hyperthreading, whose activation allows to run 80 threads simultaneously on 40 physical cores (two threads per physical core). However, the use of more than 40 threads in this evaluation has not reported any benefit as the workload imbalance limits the scalability. In fact, the speedups obtained from running 64 and 80 threads (not shown for clarity purposes) are slightly lower than running 40 threads due to the higher overhead of executing twice the number of model optimization tasks, which burdens memory access performance and OS thread scheduling. This result would seem questionable as Intel has reported that hyperthreading can provide up to 30% higher performance and indeed we reported such a performance benefit in our previous work on an 8-core system (Darriba et al., 2011a). Nevertheless, there is no contradiction as hyperthreading increases jModelTest2 shared memory scalability when an increment in the number of threads improves the workload balance, such as running on an 8-core system, whereas it does not yield any benefit when using 40 or more cores.

3.4. Evaluation of the distributed memory algorithm

The distributed memory implementation of jModelTest2 has been evaluated on the Westmere-EP cluster, showing

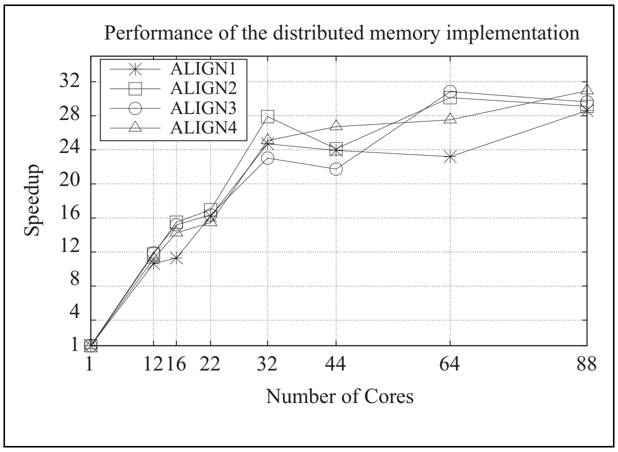

Figure 11. Scalability of the distributed memory version (Westmere-EP testbed).

Table 4. Execution times (hh:mm:ss) of the distributed memory implementation (Westmere-EP testbed).

Threads	ALIGN1	ALIGN2	ALIGN3	ALIGN4
1	100:11:08	12:53:56	12:36:04	04:01:50
12	09:25:32	01:05:50	01:03:29	00:21:34
16	08:51:50	00:49:56	00:49:47	00:16:58
22	06:11:03	00:45:38	00:46:15	00:15:39
32	04:03:22	00:27:44	00:32:50	00:09:38
44	04:11:06	00:32:05	00:34:47	00:09:03
64	04:19:04	00:25:41	00:24:31	00:08:47
88	03:30:02	00:26:36	00:25:31	00:07:49

the measured execution times and the associated speedups in Figure 11 and Table 4, respectively. In this testbed the sequential execution time is around 15–25% faster than on the Westmere-EX system, an expected result according to their respective computational power, measured in terms of the SPEC CPU floating point CFP2006 benchmark (the optimization of models is intensive in floating point operations). Thus, a single core from the Westmere-EX system obtains a result of 49.6 in the CFP2006 whereas a single core from the Westmere-EP system achieves a result of 60.3, a 22% higher.

This implementation distributes the workload among the available message-passing processes, using up to 88 processes, the maximum number of models to be optimized and running each process in a single core. In this testbed the particular allocation of processes among the cluster nodes has a negligible impact on performance ($< 0.1\%$ runtime overhead) due to the computationally intensive nature of the application with respect to the communications required (ML optimization accounts for nearly all of the execution time). In this performance evaluation the processes have been distributed among the cluster nodes using a fill-up allocation rule, minimizing the number of nodes by using the 12 cores available per node (e.g. the execution using 88 processes has distributed 12 processes per node across 7 nodes, and 4 processes in the eighth node).

The workload imbalance presented in the evaluated data sets (22 out of 88 tasks optimizing the most complex models accounts for approximately 50% of the total runtime and 44 out of 88 tasks require more than 80% of the total runtime, as shown in Figure 2) imposes an upper bound in the scalability, limiting the measured speedups to around 30, as for the shared memory implementation. Thus, distributing a small number of models per process severely limits the load balancing benefits, as it is not possible to take advantage of the spare computational power available once a process finishes its task processing and the task queue is empty. In fact, little performance benefits are obtained when using more than 32 processes, the speedups using 32 processes are around 25 (78% parallel efficiency), whereas the speedups using 88 processes are around 30 (34%), the use of 56 additional processes (a 175% resources increase, from 32 up to 88 cores) hardly improves speedups (20% higher, from 25 to 30). When using more than 32 cores most of the processes only compute a single model and finish working earlier than the longest running model, which is the one that determines the overall runtime (and, hence, the speedup) in these scenarios.

3.5. Evaluation of the hybrid shared/distributed algorithm

The limitations of the shared and distributed memory implementations of jModelTest2 have motivated the development of a more scalable implementation based on the decomposition of the model optimization among multiple threads (see Section 2.4). This decomposition depends on the allocated resources, the number of available cores per node and the computational cost of each model optimization. Thus, the longest running model computations, such as the gamma models (+G and +I+G), are split among multiple threads whereas the lightest ones are executed by one or two threads. The final objective is to achieve the workload balance among all of the involved processes. Thus, this new implementation is able to take advantage of hybrid shared/distributed memory architectures, such as clusters of multicore nodes, without compromising significantly its efficiency.

This new implementation of jModelTest2 has been evaluated on the Westmere-EP cluster, where each node has 12 physical cores and can run up to 24 simultaneous threads thanks to hyperthreading. The measured execution times and the associated speedups are shown in Figure 12 and Table 5, respectively. The experimental results have been obtained using 12 threads per node for executions from 192 up to 480 threads, thus 16, 24, 33 and 40 nodes have been used for executions on 192, 288, 396 and 480 threads, respectively. The performance results using 40 threads (4 nodes) and 88 threads (8 nodes) are also shown for comparative purposes against the results of the shared and distributed memory versions, evaluated using up to 40 and 88 cores, respectively. Here the resources allocated for each type of model (i.e. uniform, +I, +G or +I+G) are selected

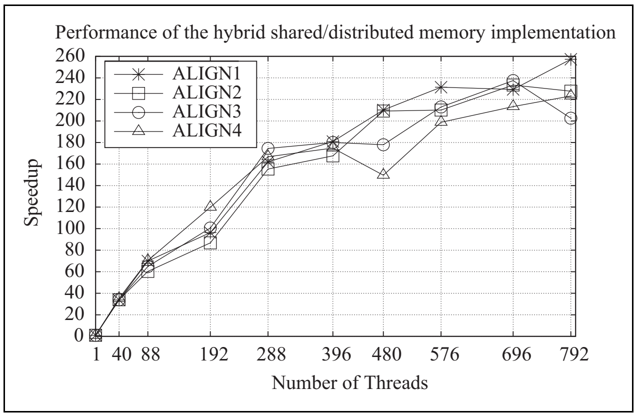

Figure 12. Scalability of the hybrid shared/distributed version (Westmere-EP testbed).

Table 5. Execution times (hh:mm:ss) of the hybrid shared/distributed memory implementation (Westmere-EP testbed).

Threads	ALIGN1	ALIGN2	ALIGN3	ALIGN4
1	100:11:08	12:53:56	12:36:04	04:01:50
40	02:58:04	03:13:29	03:09:01	00:06:52
88	01:25:55	00:12:50	00:11:39	00:03:26
192	01:02:24	00:08:55	00:07:31	00:02:01
288	00:37:07	00:04:59	00:04:20	00:01:27
396	00:33:14	00:04:37	00:04:12	00:01:23
480	00:28:36	00:03:42	00:04:15	00:01:37
576	00:25:59	00:03:41	00:03:33	00:01:13
696	00:26:13	00:03:19	00:03:11	00:01:08
792	00:23:22	00:03:24	00:03:44	00:01:05

depending on the number of total resources. For example, an execution with 4 nodes and 12 threads per node would use 4 threads for each gamma model (+G and +I+G) and a single thread for the rest.

This benchmarking has also taken into account the evaluation of the impact of hyperthreading in the performance of this new implementation. Thus, the executions with 576, 696 and 792 threads have used 32 nodes × 18 threads per node, 29 nodes × 24 threads per node and 44 nodes × 18 threads per node, respectively. The main conclusion derived from the analysis of these performance results is that jModelTest2 takes advantage of hyperthreading, both running the maximum number of simultaneous threads per node (24) or sharing half of the physical cores (18 threads running on 12 physical cores).

The analysis of the results shows that this implementation achieves significantly higher scalability, speedups around 230, in the range 203–257, which is the result of multiplying the scalability of the distributed memory processing (speedups around 30) by the scalability obtained by the parallel execution of the optimization of the models (speedups up to 8). In fact, this multilevel parallel implementation increases 8 times jModelTest2 performance, that is to say, its performance benefits are equivalent to the scalability obtained from the parallelization of the model optimization.

4. Conclusions

A popular tool for the statistical selection of models of DNA substitution is jModelTest, a sequential Java application that requires vast computational resources, especially CPU hours, which has motivated the development of its parallel implementation (jModelTest2, distributed under a GPL license at http://code.google.com/jmodeltest2). This paper presents its three parallel execution strategies: (1) a shared memory multithread GUI-based desktop version; (2) a distributed memory cluster-based version with workload distribution through message-passing; and (3) a multilevel hybrid shared/distributed memory version that distributes the computation of the likelihood estimation task across cluster nodes while taking advantage, through a thread-based parallelization, of the multiple cores available within each node.

The performance evaluation of these three strategies has shown that the hybrid shared/distributed memory implementation of jModelTest2 presents significantly higher performance and scalability than the shared and distributed memory versions, overcoming their limitations that force their execution using only up to 32–40 cores. Thus, the new implementation can take advantage efficiently of the use of up to several hundreds of cores. The observed parallel efficiencies are around 38–49%, with speedups in the range 203–257 on 528 physical cores. This performance has been obtained thanks to the workload balance provided by the thread-based decomposition of the most costly model optimization tasks.

The shared memory implementation of jModelTest2 provides scalable performance although generally up to 40 threads. Nowadays this limitation is becoming more and more important as the number of available cores per system continues increasing, especially with the advent of many-core processors which definitely demand further workload decomposition in jModelTest2.

The distributed memory implementation shows similar performance results as the shared memory version despite supporting distributed memory architectures with hundreds of cores. In fact, using more than 32 cores results in highly inefficient scalability gains due to the significant workload imbalance present in jModelTest2. However, this solution avoid memory limitations with large input data.

The hybrid shared/distributed memory implementation provides a three-level parallelism avoiding memory limitation as the previous strategy while using a heterogeneous computational resource distribution in order to achieve a better load balancing. Although there is a high overhead in the parallel execution of each task, this strategy homogenizes the task execution times, thus balancing the workload.

Acknowledgement

Special thanks are due to Stephane Guindon for his help.

Funding

This work was financially supported by the European Research Council (grant number ERC-2007-Stg 203161-

PHYGENOM to D.P.) and the Spanish Ministry of Science and Education (grant numbers BFU2009-08611 to D.P. and TIN2010-16735 to R.D.).

References

Akaike H (1974) A new look at the statistical model identification. *IEEE Transactions on Automatic Control* 19(6): 716–723.

Buckley TR (2002) Model misspecification and probabilistic tests of topology: evidence from empirical data sets. *Systematic Biology* 51: 509–523.

Buckley TR and Cunningham CW (2002) The effects of nucleotide substitution model assumptions on estimates of nonparametric bootstrap support. *Molecular Biology and Evolution* 19: 394–405.

Chor B and Tuller T (2006) Finding a maximum likelihood tree is hard. *Journal of the ACM* 53(5): 722–744.

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

Darriba D, Taboada GL, Doallo R and Posada D (2011a) HPC selection of models of DNA substitution. In: *Proceedings of the Workshop on High Performance Computational Systems Biology (HiBi'11), 9th International Conference on Computational Methods in Systems Biology (CMSB'11)*, Paris, France, pp. 65–72.

Darriba D, Taboada GL, Doallo R and Posada D (2011b) Prottest 3: fast selection of best-fit models of protein evolution. *Bioinformatics* 27: 1164–1165.

Felsenstein J (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution* 17: 368–376.

Felsenstein J (1988) Phylogenies from molecular sequences: inference and reliability. *Annual Review of Genetics* 22: 521–565.

Gascuel O (1997) BIONJ: an improved version of the NJ algorithm based on a simple model of sequence data. *Molecular Biology and Evolution* 14(7): 685–95.

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

Lemmon AR and Moriarty EC (2004) The importance of proper model assumption in Bayesian phylogenetic. *Systematic Biology* 53: 265–277.

Posada D (2008) jModelTest: phylogenetic model averaging. *Molecular Biology and Evolution* 25(7): 1253–1256. DOI: 10.1093/molbev/msn083.

Posada D and Crandall KA (2001) Selecting the best-fit model of nucleotide substitution. *Systematic Biology* 50(4): 580–601.

Schwarz G (1978) Estimating the dimension of a model. *Annals of Statistics* 6: 461–464.

Shafi A, Carpenter B and Baker M (2009) Nested parallelism for multi-core HPC systems using Java. *Journal of Parallel and Distributed Computing* 69(6): 532–545.

Author biographies

Diego Darriba received the B.S. (2010) and M.S. (2011) degrees in Computer Science from the University of A Coruña, Spain. Currently he is a Ph.D. student at the Universities of A Coruña and Vigo. His research interests are in the area of bioinformatics, focused on applying HPC to phylogenetics and phylogenomics.

Guillermo L Taboada received the B.S. (2002), M.S. (2004) and Ph.D. (2009) degrees in Computer Science from the University of A Coruña, Spain, where he is currently an Associate Professor in the Department of Electronics and Systems. His main research interest is in the area of HPC, focused on high-speed networks, programming languages for HPC, cluster/Grid/cloud computing and, in general, middleware for HPC.

Ramón Doallo received his B.S. (1987), M.S. (1987) and Ph.D. (1992) degrees in Physics from the University of Santiago de Compostela, Spain. In 1990 he joined the Department of Electronics and Systems at the University of A Coruña, where he became a Full Professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is coauthor of more than 140 technical papers on these topics.

David Posada received his B.S. (1994) and M.S. (1997) degrees in Biology from the University of Santiago de Compostela, Spain, and his Ph.D. (2001) in Zoology from the Brigham Young University. In 2003 he joined the University of Vigo, where he became a Full Professor in 2010. He has extensively published in the areas of phylogenetics and phylogenomics. He is coauthor of more than 120 technical papers on these topics.