Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code

Wayne Pfeiffer
San Diego Supercomputer Center
University of California, San Diego
La Jolla, California, USA
pfeiffer@sdsc.edu

Alexandros Stamatakis
Department of Computer Science
Technische Universität München
Munich, Germany
stamatak@cs.tum.edu

***Abstract*—A hybrid MPI/Pthreads parallelization was implemented in the RAxML phylogenetics code. New MPI code was added to the existing Pthreads production code to exploit parallelism at two algorithmic levels simultaneously: coarse-grained with MPI and fine-grained with Pthreads. This hybrid, multi-grained approach is well suited for current high-performance computers, which typically are clusters of multi-core, shared-memory nodes.**

The hybrid version of RAxML is especially useful for a comprehensive phylogenetic analysis, i.e., execution of many rapid bootstraps followed by a full maximum likelihood search. Multiple multi-core nodes can be used in a single run to speed up the computation and, hence, reduce the turnaround time. The hybrid code also allows more efficient utilization of a given number of processor cores. Moreover, it often returns a better solution than the stand-alone Pthreads code, because additional maximum likelihood searches are conducted in parallel using MPI.

The comprehensive analysis algorithm involves four stages, in which coarse-grained parallelism continually decreases from stage to stage. The first three stages speed up well with MPI, while the last stage speeds up only with Pthreads. This leads to a tradeoff in effectiveness between MPI and Pthreads parallelization.

The useful number of MPI processes increases with the number of bootstraps performed, but typically is limited to 10 or 20 by the parameters of the algorithm. The optimal number of Pthreads increases with the number of distinct patterns in the columns of the multiple sequence alignment, but is limited to the number of cores per node of the computer being used.

For a benchmark problem with 218 taxa, 1,846 patterns, and 100 bootstraps run on the Dash computer at SDSC, the speedup of the hybrid code on 10 nodes (80 cores) was 6.5 compared to the Pthreads-only code on one node (8 cores) and 35 compared to the serial code. This run used 10 MPI processes with 8 Pthreads each. For another problem with 125 taxa, 19,436 patterns, and 100 bootstraps, the speedup on the Triton PDAF computer at SDSC was 38 on two nodes (64 cores) compared to the serial code. This run used 2 MPI processes with 32 Pthreads each.

Keywords; hybrid parallelization; MPI/Pthreads; phylogenetics; RAxML

1. INTRODUCTION

The calculation of phylogenetic trees from multiple sequence alignments is extremely intensive computationally and will only become more so as the deluge of molecular sequence data continues. The use of parallel codes is thus essential for analyzing large data sets in a reasonable amount of time.

Fortunately, phylogenetic computations are amenable to parallelization at multiple algorithmic levels. Fine- or medium-grained parallelization can be used within a tree, while coarse-grained parallelization can be used across trees. Approaches that parallelize at multiple levels simultaneously are called *multi-grained*, while approaches that simultaneously use different programming models (such as MPI and OpenMP) are called *hybrid*.

Coarse-grained parallelization is straightforward with MPI and has been implemented in the MrBayes [1] and GARLI [2] phylogenetics codes. Various versions of the RAxML phylogenetics code [3-10] have included a similar implementation as well as fine- or medium-grained parallelizations using MPI, Pthreads, OpenMP, and Cell-specific code. To date, however, production versions of RAxML have allowed only one parallel approach to be used at a time. On the other hand, experimental versions of the code have included multi-grained and hybrid approaches, and a hybrid MPI/OpenMP version of the IQPNII phylogenetics code [11] was developed that parallelizes at two different levels, one medium-grained and the other fine-grained.

This paper describes a hybrid, multi-grained version of RAxML (Random Axelerated Maximum Likelikhood) that recently became publicly available for production use as open-source code [10]. New coarse-grained MPI code, which is simpler and often more efficient than in previous versions, was added to the production, fine-grained Pthreads code.

Whereas virtually all RAxML analyses can benefit from fine-grained Pthreads parallelization within a single tree search, the following three types of analyses involving multiple trees are also amenable to coarse-grained MPI parallelization.

1. Multiple maximum likelihood (ML) searches on the same data set, but starting from different initial trees. Typically 10 or more such searches might be made to find a near-optimal ML solution.

2. Multiple bootstrap searches, which are ML searches on data sets obtained by randomly re-sampling the columns of the multiple sequence alignment. Bootstrapping allows confidence values to be assigned to the interior branches of

the ML tree obtained in the first analysis. Typically 100 or more bootstraps are used.

3. A so-called *comprehensive analysis* that combines the two preceding analyses. Specifically, many rapid bootstraps are performed followed by a full ML search as described in [12]. This algorithm allows a complete, publishable, phylogenetic analysis in a single run.

The hybrid treatment of the first two analyses is straightforward, since they have essentially constant parallelism throughout, apart from minor load imbalances. By contrast, the comprehensive analysis proceeds in multiple stages with continually decreasing coarse-grained parallelism. This complicates its hybrid treatment, which is the primary topic of this paper, and leads to an interesting tradeoff between the preferred numbers of MPI processes and Pthreads.

Compared to the Pthreads-only code, the hybrid version of RAxML provides significant performance benefits for all three of the preceding analyses performed on clusters, currently the most common high-performance computers. Of particular importance is the ability to use multiple computer nodes to achieve greater speedup and shorter turnaround in a single run. This and other benefits are demonstrated here for comprehensive analyses of representative real-world data sets run on several modern clusters.

2. Implementation

RAxML has been parallelized in various ways and at various levels of granularity as the capabilities of the code have evolved. Table 1 summarizes this evolution.

RAxML-II [3] was the first version of the code to be parallelized. This implementation was medium-grained and done with MPI. In RAxML-OMP [4] the search algorithm was revised, and the parallelization was changed to fine-grained using OpenMP. This evolved into RAxML-VI-HPC [5], which also allowed multiple bootstraps or multiple searches on different starting trees to run in parallel using a coarse-grained MPI approach. However, such MPI runs could not use OpenMP simultaneously.

Subsequently, three experimental versions of RAxML were developed. The first [6] was specific to the Cell Broadband Engine. It was both multi-grained and hybrid, but involved Cell-specific code for the fine-grained parallelization as well as a Cell-specific system-level scheduler for efficiently exploiting the coarse-grain parallelism with MPI. The second experimental code [7] was targeted at the IBM Blue Gene/L. It was multi-grained, but used MPI at both levels of granularity.

The third experimental version [8] of RAxML was developed as part of a performance study comparing fine-grained parallelizations done with MPI, Pthreads, or OpenMP. Although each approach could be fastest depending upon the data set and computer, the Pthreads implementation was adopted in production Version 7.0.0 [9] and replaced the earlier OpenMP implementation. The Pthreads approach was chosen because it is more widely available, easier to compile for biologists, and simpler to use when prototyping. The MPI option was retained in Version 7.0.0, but disabled in 7.1.0 [10] when bootstopping [13] was added.

Version 7.2.4 (available at [10]) is the first version to include the hybrid parallelization described here. The fine-grained Pthreads parallelization is the same as in recent versions and is over the number of patterns (which are defined in Section 3). The coarse-grained MPI parallelization is over the number of separate tree searches, similar to that in Version 7.0.0. However, the new MPI approach is simpler than the former master/worker approach and has minimal MPI communication. Thus a fast and expensive interconnect is not required. The new approach is also more efficient when there is reasonable load balance, which is often the case.

The hybrid parallelization works well for all three analyses mentioned in the Introduction, but is expected to be particularly useful for the comprehensive analysis. However, the current implementation only handles a fixed number of bootstraps, not the case where that number can vary depending upon a bootstopping test [13]. Parallelization of that test, which operates on bipartitions of trees stored in a hash table, will require implementation of a framework for parallel operations on hash tables on multi-core nodes.

The comprehensive analysis consists of four main stages: 100 bootstrap searches, followed by 20 fast ML searches, 10 slow ML searches, and one final thorough ML search, where the numbers are those for the typical case described in [12]. The latter three stages comprise the full ML search.

TABLE 1. Evolution of parallel versions of RAxML

Year	Code version	Coarse-grained	Fine-grained	Multi-grained	Hybrid	Reference
2004	II		MPI[a]			[3]
2005	OMP		OpenMP			[4]
2006	VI-HPC	MPI	OpenMP	No	No	[5]
2007	Cell	MPI	Cell-specific	Yes	Yes	[6]
2007	Blue Gene/L	MPI	MPI	Yes	No	[7]
2008	Performance		MPI, Pthreads, or OpenMP	No	No	[8]
2008	7.0.0	MPI	Pthreads	No	No	[9]
2009	7.1.0		Pthreads			[10]
2009	7.2.4	MPI	Pthreads	Yes	Yes	This paper, [10]

a. This parallelization was medium-grained.

The first three stages can run completely in parallel via coarse-grained MPI parallelization, at least up to 10 MPI processes, and so can achieve significant speedup. By contrast, there is no speedup from MPI in the last stage, as discussed in the next subsection.

The new MPI code begins by having each MPI process parse its own input and then gives each process N/p bootstraps, where N is the number of bootstraps specified, and p is the number of processes. As the analysis proceeds, there are four noteworthy differences between the MPI and non-MPI code that affect the final solution. These differences are discussed in the next four subsections.

2.1. *One versus p thorough searches*

The most significant difference is in the final stage. After the slow searches are completed, the non-MPI code selects the tree with the best ML score from the slow searches to continue with a thorough tree search. By contrast, the MPI code lets each process continue with a thorough search starting from the best tree generated by the slow search computed locally by that process. Doing several thorough searches instead of just one as in the serial code increases the total work, but does not increase the run time very much. This additional, useful work is conducted in parallel, and the load from each process is typically well balanced.

Once all p thorough searches are done, the best solution among them is selected for output using a call to MPI_Bcast. That and a call to MPI_Barrier after the bootstrap stage are the only noteworthy MPI communications in the new code.

2.2. *Sorting between fast and slow searches*

The non-MPI code typically selects only a fraction of the fast searches to continue with slow searches. The searches selected are those with the best ML values. This requires a sort by ML value of all of the fast searches.

In the MPI code each process sorts only its own local fast searches. This avoids communication, but is in general less optimal than sorting all of the searches at once. In practice, any loss of optimality seems to be more than offset by the additional thorough searching described previously.

2.3. *Numbers of bootstraps and subsequent searches*

In the non-MPI code the number of bootstraps done is that specified via the command line (excluding the case of bootstopping). By contrast, the number of bootstraps done in the MPI code can be slightly larger than the specified number depending upon the number of processes, since each process does an equal number of bootstraps. This in turn affects how many fast and slow searches are carried out based on hard-coded parameters.

Table 2 lists how many bootstraps and subsequent searches are done for various numbers of processes, given that the specified number of bootstraps is 100 for most rows or 500 for the last two rows. Since every process does a single thorough search, no speedup from MPI is expected for that stage. The first three stages, however, should speed up nearly perfectly for 2, 5, and 10 MPI processes, given reasonable load balance, while one or more of these stages will scale more slowly for other process counts.

Speedup beyond 10 processes becomes more limited because all processes are then doing a single slow search and a single thorough search. By 20 processes, speedup of the fast searches is also limited for the case of 100 bootstraps, though not for the case of 500 bootstraps. As shown in Section 5, using more than 10 or 20 processes is seldom justified.

2.4. *Treatment of random numbers*

Getting reproducible results is highly desirable in a parallel code, but requires special attention when the code uses random numbers. In contrast to the previous coarse-grained version of RAxML, the MPI code gives reproducible results for a given set of input parameters and a given number of MPI processes, provided random number seeds are specified via the –p parameter and either the –x parameter for rapid bootstrapping or the –b parameter for standard bootstrapping. Reproducibility is ensured by using the specified seeds on MPI Process 0 and seeds incremented by constant amounts (specifically, multiples of 10,000) on the other processes.

3. BENCHMARK DATA SETS

A data set is described by its multiple sequence alignment, which is a matrix of aligned molecular sequences. The rows of the matrix correspond to different taxa, and the columns correspond to character positions in the aligned sequences. Because some character positions may be redundant, the number of distinct columns, called *patterns*, is a more descriptive parameter than the number of characters.

TABLE 2. NUMBERS OF BOOTSTRAPS AND SEARCHES VERSUS NUMBER OF PROCESSES

Processes	**Bootstraps**	**Fast searches**	**Slow searches**	**Thorough searches**	**Bootstraps/ process**	**Fast searches/ process**	**Slow searches/ process**	**Thorough searches/ process**
1	100	20	10	1	100	20	10	1
2	100	20	10	2	50	10	5	1
4	100	20	12	4	25	5	3	1
5	100	20	10	5	20	4	2	1
8	104	24	16	8	13	3	2	1
10	100	20	10	10	10	2	1	1
16	112	32	16	16	7	2	1	1
20	100	20	20	20	5	1	1	1
10	500	100	10	10	50	10	1	1
20	500	100	20	20	25	5	1	1

Indeed, the amount of work to be done is roughly proportional to the number of patterns for a fixed number of taxa.

The benchmark results presented here are for five DNA and RNA data sets with the parameters listed in Table 3. These data sets (and more) are also considered in [12] and available at [14]. The data sets in the table are ordered by increasing number of patterns. Also listed in the table is the number of bootstraps recommended based upon the WC bootstopping test in [13].

4. BENCHMARK COMPUTERS

The benchmark runs used four computers, whose key characteristics are listed in Table 4. All of these computers are clusters with x64 processors that have similar clock speeds. However, the newer Nehalem and Shanghai processors are expected to perform better than the older Clovertown and Barcelona processors. Also, the bus-based memory subsystem of the Clovertown processor is generally slower than the memory subsystems of the other processors. The varying speeds of the interconnects (which are not shown) have a negligible effect on the performance of RAxML because the MPI communication is minimal.

All times reported here are for runs that had exclusive use of the assigned node(s). Indeed, for all of the computers except Triton PDAF, this is the normal mode of operation, and the user is charged for all cores in each assigned node, whether they are used or not. Thus it is desirable to run on core counts that are multiples of the cores per node. Nevertheless, some results are presented here on fewer cores than in a single node just to show complete scaling curves starting from the serial case.

Some runs were repeated to check on timing variability. It was found to be a few percent at most except on Triton PDAF, where variations of more than 20% were occasionally observed. The reason for such variations has not been determined. For cases when repeat runs were made, the fastest time is reported here.

On all of the computers the hybrid code was built using the Intel compiler and Open MPI. On Abe, Ranger, and Triton PDAF the compiler automatically invoked the SSE3 instructions. On Dash the compiler directive –xsse4.2 was used to invoke SSE4.2 instructions, which improved performance by about 10%. SSE3 directives available in the code since Version 7.2.1 were not invoked, because doing so had little effect when the Intel compiler was used as just described.

TABLE 3. BENCHMARK DATA SETS

Taxa	Characters	Patterns	Recommended bootstraps [13]
354	460	348	1,200
150	1,269	1,130	650
218	2,294	1,846	550
404	13,158	7,429	700
125	29,149	19,436	50

TABLE 4. BENCHMARK COMPUTERS

Computer	Location	Processor	Cores/ node
Abe	NCSA	2.33-GHz Intel Clovertown	8
Dash	SDSC	2.4-GHz Intel Nehalem	8
Ranger	TACC	2.3-GHz AMD Barcelona	16
Triton PDAF	SDSC	2.5-GHz AMD Shanghai	32

5. PERFORMANCE

To investigate performance of the hybrid code, benchmark runs were made for all five data sets on one or more of the four computers. All runs were for comprehensive analyses. The first runs specified 100 bootstraps, with key input parameters on the RAxML command line being -m GTRCAT -N 100 -p 12345 -x 12345 -f a. Additional runs were made for the first four data sets using the larger numbers of bootstraps recommended by the WC bootstopping test in [13].

5.1. Results for 100 bootstraps specified

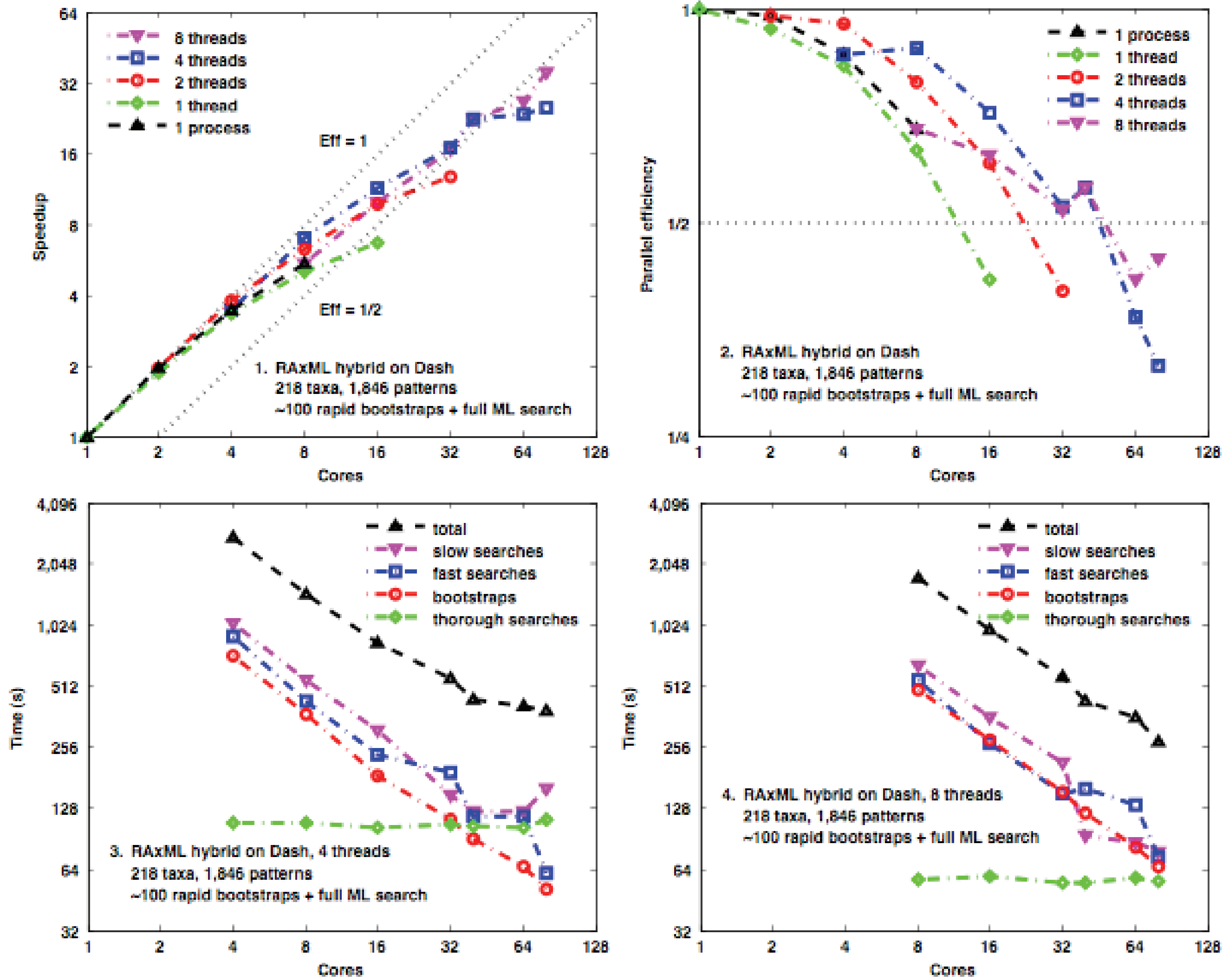
Fig. 1 shows a *speedup* plot for the medium-sized data set with 1,846 patterns run on Dash with 100 bootstraps specified. (Speedup is just the speed normalized to 1 on a single core.) Four curves are shown for constant numbers of threads, and one curve is shown for a single process. Also shown are curves for two values of the *parallel efficiency* (which is the speedup per core). A parallel efficiency of 1 corresponds to ideal, linear speedup.

Runs for the limiting cases of one process and one thread did not use the hybrid code. Instead, runs for one process and multiple threads used the Pthreads-only code to avoid the overhead associated with using a single MPI process. That overhead was found to be more than 10% for the smallest data sets. Runs for one thread and multiple processes used an MPI-only code, since the Pthreads code with its master/worker parallelization does not work for a single thread. Runs for one process and one thread used the serial code.

Fig. 1 shows good scaling up to 80 cores. There the speedup is 35 using 10 processes and 8 threads. Also, shown are differences in performance at lower core counts depending upon the number of threads. However, these differences are difficult to see in the speedup plot.

Thus, Fig. 2 displays the same data in a parallel efficiency plot, which rotates the curves in Fig. 1 until the lines of constant parallel efficiency are horizontal. This increases the separation between the curves and makes the differences in performance much clearer. In particular, using 4 threads is fastest on 8 and 16 cores, while using 8 threads is best on 64 and 80 cores. In between, 4 and 8 threads perform similarly.

Fig. 2 also shows that the parallel efficiency on 40 and 80 cores is better than on 32 and 64 cores, respectively. This is because the more efficient runs use 5 and 10 processes, which are expected to scale better than 4 or 8 processes based on the numbers listed in Table 2.



Figures 1 to 4. Scaling plots showing speedup (1), parallel efficiency (2), and run-time components (3 and 4) for the problem with 1,846 patterns on Dash

Additional insight into the scaling can be obtained from Figs. 3 and 4, which plot the run-time components versus the number of cores for the same problem using either 4 or 8 threads on Dash. The time for the first three stages (bootstraps, fast searches, and slow searches) decreases up to 40 cores using 4 threads and up to 80 cores using 8 threads. This reflects the effectiveness of MPI for these stages. However, the time for the last stage (thorough searches) is roughly constant, since the only parallelism exploited for its speedup is that via Pthreads.

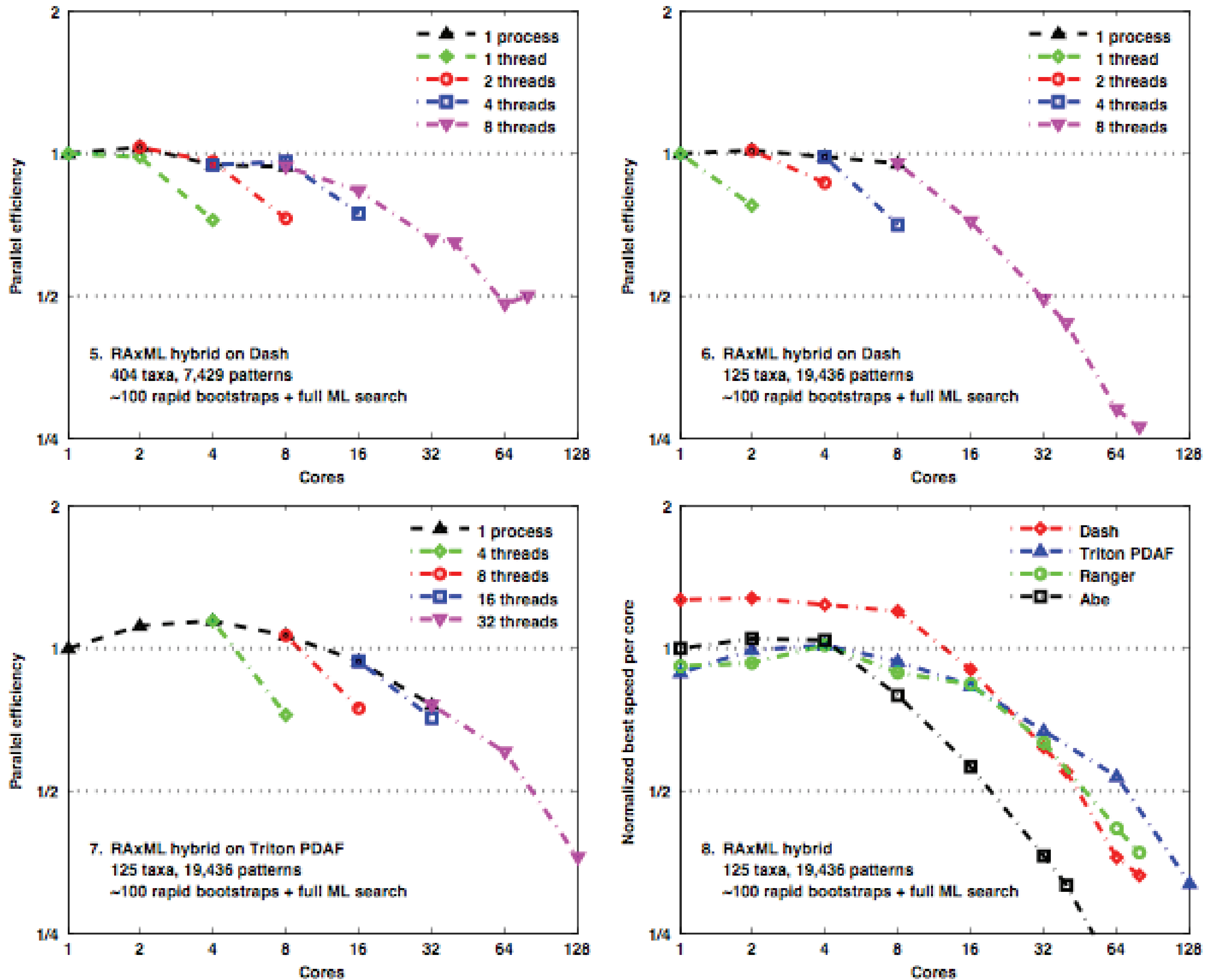
Comparing Figs. 3 and 4, the time for the thorough searches is almost twice as long using 4 threads as with 8. By contrast, the times for the other stages are slightly shorter using 4 threads as compared to 8. This leads to a total time that is shorter using 4 threads at low core counts and shorter using 8 threads at high core counts, consistent with the parallel efficiency curves in Fig. 2.

One point of note is that the MPI implementation has no barriers between the last three stages, so the times for those stages vary depending upon the MPI process, since the load is not perfectly balanced. The times shown are those for the last process to finish.

The availability of the hybrid code not only allows greater speedup than is possible with the earlier Pthreads-only code, but also allows more efficient use of the available cores. For example, on a single 8-core node of Dash, using 2 processes and 4 threads is 1.3x faster than using 8 threads in the Pthreads-only code and nearly 1.4x faster than using 8 processes in the MPI-only code. This leads to a speedup of 6.5 on 10 nodes using the hybrid code as compared to a single node using the Pthreads-only code.

In general, the optimal number of threads increases with the number of patterns in the data set. This can be seen in Figs. 5 and 6, which show parallel efficiency plots for runs on Dash for the two data sets with the largest number of patterns. For these data sets, runs on 16 or more cores of Dash should use 8 threads, the maximum possible, for optimal performance.

Runs for the data set with the largest number of patterns were also made on Triton PDAF, which has 32 cores per node. The resulting parallel efficiency plot is shown in



Figures 5 to 8. Scaling plots showing parallel efficiency (5 to 7) and best speed per core (8) for various problems and computers

Fig. 7. As can be seen there, optimal performance is achieved using all 32 threads available, and the scaling at high core counts is better than on Dash.

For the same data set, Fig. 8 compares the best speeds per core on all four computers. (Here best means the fastest at each core count for the optimal number of threads, and the plotted speed per core is just the parallel efficiency normalized to that for Abe.) The scaling differences are striking.

From 1 to 4 cores, all of the computers except Dash show superlinear speedup (increasing parallel efficiency), because their cache utilization is improving. By contrast, Dash exhibits ideal, linear speedup up to 8 cores, suggesting that its newer cache design is more effective.

As the core count increases further, efficiency drops off fastest for Abe and then Dash. Both have 8 cores per node, but the memory bandwidth is much higher on Dash. The drop in efficiency is more gradual on Ranger and Triton PDAF, which have 16 and 32 cores per node, respectively, and can therefore use more threads. Thus, even though Dash is fastest up to 16 cores, Triton PDAF becomes faster at higher cores counts. Having fast processors (as Dash does) is always beneficial, but having more cores per node (as Triton PDAF does and many future computers will) allows more threads, which is advantageous for data sets with a large number of patterns.

The upper part of Table 5 summarizes the fastest times for each data set at various core counts for runs with 100 bootstraps specified. For the first four data sets, Dash is fastest in all cases. For the largest data set, Triton PDAF is faster at high core counts, as noted previously.

The table also lists the number of threads at which the best time on the indicated computer is achieved. On 80 cores of Dash, for example, the optimal number of threads is 4 for the first data set and 8 for the last four data sets. Those values correspond to using 20 and 10 processes, respectively. Similarly, the optimal number of threads on 64 cores of Triton PDAF is 32 for the last data set, which corresponds to using 2 processes.

TABLE 5. FASTEST TIMES FOR EACH DATA SET

Taxa	Patterns	Best time (s) / threads on					Speedup on				Computer/ bootstraps
		1c	*8c*	*16c*	*40c*	*80c*	*8c*	*16c*	*40c*	*80c*	
Results for 100 bootstraps specified											
354	348	1,980	432 /2	307 /2	168 /4	130 /4	4.58	6.45	11.79	15.23	Dash
150	1,130	2,325	456 /4	283 /4	139 /4	95 /8	5.10	8.22	16.73	24.47	Dash
218	1,846	9,630	1,370 /4	846 /4	430 /8	271 /8	7.03	11.38	22.40	35.54	Dash
404	7,429	72,866	9,494 /4	5,497 /8	2,830 /8	1,828 /8	7.67	13.26	25.75	39.86	Dash
125	19,436	22,970	3,018 /8	2,006 /8	1,314 /8	1,092 /8	7.61	11.45	17.48	21.03	Dash
125	19,436	32,627	3,844 /8	2,179 /16	1,351 /32 *	847 /32 +	8.49	14.97	24.15 *	38.52 +	Triton PDAF
Results for >100 bootstraps specified											
354	348	15,703	2,286 /1	1,287 /1	702 /2	443 /2	6.87	12.20	22.37	35.45	Dash/1,200
150	1,130	10,566	1,714 /2	980 /2	473 /2	290 /4	6.16	10.78	22.34	36.43	Dash/ 650
218	1,846	33,738	5,184 /2	2,778 /2	1,290 /4	845 /4	6.51	12.14	26.15	39.93	Dash/ 550
404	7,429	355,724	45,851 /4	25,454 /4	11,229 /4	6,270 /8	7.76	13.98	31.68	56.73	Dash/ 700

1c, 8c, 16c, 40c, and 80c refer to the number of cores used in a run. The run with a * was made on 32c, while that with a + was made on 64c.

The scaling on Dash improves as the number of patterns increases in the first four data sets, consistent with the scalability of fine-grained parallelism discussed in Subsection 4.2 of Ott, et al. [7]. The scaling on Dash drops for the last data set because the fraction of time spent doing thorough searches is much larger, and those searches are not sped up by MPI. As noted previously and as is apparent from the table, much better scaling is achieved for that data set on Triton PDAF, which allows more threads.

5.2. Results for more than 100 bootstraps specified

The preceding results are all for comprehensive analyses using a nominal value of 100 for the number of bootstraps. However, Pattengale, et al. [13] developed a so-called *WC bootstopping* test to determine how many bootstraps are needed for adequate statistical support and concluded that more than 100 bootstraps are typically required. In particular, the recommended values for the number of bootstraps for the first four data sets vary from 550 to 1,200, as listed in Table 3, and only the last data set has a smaller recommended value of 50.

Thus, additional runs were made on Dash for the first four data sets using the larger, recommended values for the number of bootstraps. The results for the fastest times and corresponding numbers of threads at various core counts are listed in the bottom part of Table 5.

Two changes from the results in the upper part of the table are apparent. First, the scaling is significantly improved. Second, the optimal number of threads is reduced. These changes occur because the fraction of time spent doing bootstraps and fast searches increases, and both of these stages are especially amenable to coarse-grained MPI parallelization.

The improvement in scaling is greatest for the first data set, where the increase in the number of bootstraps from 100 to 1,200 is largest. For this case the speedup on 80 cores increases from about 15 to 35, and the optimal number of threads drops from 4 to 2 (so the corresponding number of processes increases from 20 to 40). The highest absolute speedup is nearly 57 for the fourth data set, corresponding to a drop in run time from more than 4 days to less than 1.8 hours.

These results give optimistic estimates for the speedups achievable when a hybrid version of RAxML that includes bootstopping becomes available. The actual speedups will be reduced somewhat, because some time will be required to perform the bootstopping tests.

6. QUALITY OF SOLUTION

As mentioned previously, the additional thorough searches in the hybrid algorithm for the comprehensive analysis often give a better solution. This can be seen from the data in Table 6. In all cases shown, the multi-process solutions are as good as or better than the serial solutions. The largest improvement is for the third data set. Comparing the last two columns also suggests some benefit from doing more fast searches when the number of bootstraps is greater than 100.

7. DISCUSSION

As a rule of thumb, runs are not cost effective when the parallel efficiency becomes less than 1/2. However, the reference for calculating the parallel efficiency is relevant. In Section 5, the reference is a single core. Since users are often charged for all cores in a node, the reference could instead be a single node. In this case, larger core counts can be justified.

As an example, consider the first row of data in Table 5, which corresponds to the comprehensive analysis with 100 bootstraps of the data set with the fewest patterns. For

TABLE 6. FINAL MAXIMUM LIKELIHOODS FOR EACH DATA SET

Taxa	Patterns	Final ML for 1 process & 100 bootstraps	Final ML for 10 processes & 100 bootstraps	Final ML for 10 processes & >100 bootstraps
354 *	348	-6,560.07	-6,560.09	-6,559.65
150	1,130	-39,604.89	-39,601.30	-39,601.30
218	1,846	-134,170.79	-134,160.23	-134,154.49
404 *+	7,429	-156,117.94	-156,116.77	-156,117.94
125	19,436	-825,204.82	-825,204.82	

Results for data sets with * vary with number of threads, while those with + vary with processor type. Results listed are for 8 threads on Dash.

this case the parallel efficiency on 40 cores of Dash is only 0.29 using a single core as reference, whereas it is 0.51 using a single node as reference. Thus using 40 cores for this case seems justified. Likewise, using 80 cores seems justified for most of the other cases.

Looking to the future, data sets with many more patterns than considered here will become commonplace. This will have two important consequences for comprehensive analyses, both of which favor using a hybrid code with many threads per process and, hence, a computer with many cores per node.

First, fewer bootstraps will be needed than were recommended by the bootstopping test for most of the data sets considered here. In fact, the recommended number of bootstraps could turn out to be close to the nominal value of 100 considered in Subsection 5.1.

Second, not enough memory per core will be available to analyze a single tree using one MPI process per core. Instead the memory of multiple cores, perhaps even the entire node, will be needed for each MPI process.

8. SUMMARY

This paper has described a hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code that simultaneously exploits coarse-grained and fine-grained parallelism. The hybrid code works for all analyses with multiple tree searches, but is especially targeted at the so-called comprehensive analysis, in which many rapid bootstraps are followed by a three-stage maximum likelihood search.

The bootstrapping and first two stages of the subsequent search speed up well with MPI. However, the last stage, a thorough ML search, speeds up only with Pthreads. This leads to a tradeoff in effectiveness between MPI and Pthreads parallelization.

The useful number of MPI processes increases with the number of bootstraps performed, but typically is limited to 10 or 20 by the parameters in the comprehensive analysis algorithm. The optimal number of Pthreads increases with the number of patterns in the data set and the total number of cores being used, but is limited to the number of cores in a node.

The hybrid code is available for production use and has three significant benefits over the most recent Pthreads-only code.

1. Multiple computer nodes can be used in a single run to achieve greater speedup. For a comprehensive analysis of an example problem with 218 taxa, 1,846 patterns, and 100 bootstraps, the speedup on 10 nodes (80 cores) of the Dash computer using 10 MPI processes and 8 Pthreads was 6.5 compared to that on one node (8 cores) using 8 Pthreads alone. For the same problem, the speedup on 80 cores was 35 compared to the serial run on a single core.

2. The number of Pthreads per node can be adjusted to achieve more efficient use of cores. For the example just described, the speed using 2 MPI processes and 4 Pthreads on a single node of Dash was 1.3x faster than using 8 threads with the Pthreads-only code.

3. Additional thorough ML searches in the comprehensive analysis algorithm often lead to a better solution.

In general, the hybrid code provides a versatile tool for analyzing the data sets of today as well as those of tomorrow.

ACKNOWLEDGMENTS

Funding was provided by the National Science Foundation (NSF) and the German Science Foundation (DFG). Code development and benchmarking were done on Abe at NCSA, Dash at SDSC, and Ranger at TACC, all of which are supported by NSF. Additional benchmarking was done on the Triton PDAF computer at SDSC, which is supported by the University of California, San Diego.

REFERENCES

[1] F. Ronquist, J.P. Huelsenbeck, and P. van der Mark, MrBayes 3.1 Manual, 2005, mrbayes.csit.fsu.edu/mb3.1_manual.pdf.

[2] D.J. Zwickl, MPI version of GARLI, www.nescent.org/wg_garli/MPI_version,

[3] A. Stamatakis, T. Ludwig, and H. Meier, "Parallel Inference of a 10.000-Taxon Phylogeny with Maximum Likelihood," Proc. Euro-Par 2004, LNCS 3149, Springer-Verlag, 2004, pp. 997-1004.

[4] A. Stamatakis, M. Ott, and T. Ludwig, "RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs," Proc. PaCT 2005, LNCS 3606, Springer-Verlag, 2005, pp. 288-310.

[5] A. Stamatakis, "RaxML-VI-HPC: Maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," Bioinformatics, vol. 22(21), 2006, pp. 2688-2690.

[6] F. Blagojevic, D.S. Nikolopoulos, A. Stamatakis, and C.D. Antonopoulos, "Dynamic Multigrain Parallelization on the Cell Broadband Engine," Proc. PPoPP '07, 2007, pp. 90-100.

[7] M. Ott, J. Zola, S. Aluru, and A. Stamatakis, "Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L," Proc. SC07, 2007.

[8] A. Stamatakis and M. Ott, "Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study," in M. Chetty, A. Ngom, and S. Ahmad (Eds), Proc. PRIB 2008, LNBI 5265, Springer-Verlag, 2008, pp. 424-435.

[9] A. Stamatakis, The RaxML 7.0.0 User Manual, icwww.epfl.ch/~stamatak/index-Dateien/software/RAxML-Manual.7.0.0.pdf

[10] A. Stamatakis, Exelixis Lab Web site, wwwkramer.in.tum.de/exelixis/software.html.

[11] B.Q. Minh, L.S. Vinh, H.A. Schmidt, and A. von Haeseler, "Large Maximum Likelihood Trees," Proc. NIC Symposium 2006, 2006, pp. 357-366.

[12] A. Stamatakis, P. Hoover, and J. Rougemont, "A Rapid Bootstrap Algorithm for the RaxML Web Servers," Syst. Biol., vol. 57(5), 2008, pp. 758-771.

[13] N.D. Pattengale, M. Alipour, O.R.P. Bininda-Emonds, B.M.E. Moret, and A. Stamatakis, "How Many Bootstrap Replicates are Necessary?" Proc. RECOMB 2009, LNCS 5541, Springer-Verlag, 2009, pp. 184-200.

[14] icwww.epfl.ch/~stamatak/RAPID-RESULTS.tar.bz2.