BIOINFORMATICS ORIGINAL PAPER

Vol. 27 no. 10 2011, pages 1359–1367
doi:10.1093/bioinformatics/btr147

Phylogenetics

Advance Access publication March 27, 2011

Faster exact maximum parsimony search with XMP

W. Timothy J. White[1,*] and Barbara R. Holland[2,*]

[1]Institute of Fundamental Sciences, Massey University, Palmerston North, New Zealand and [2]School of Mathematics and Physics, University of Tasmania, Hobart, Tasmania, Australia

Associate Editor: David Posada

ABSTRACT

Motivation: Despite trends towards maximum likelihood and Bayesian criteria, maximum parsimony (MP) remains an important criterion for evaluating phylogenetic trees. Because exact MP search is NP-complete, the computational effort needed to find provably optimal trees skyrockets with increasing numbers of taxa, limiting analyses to around 25–30 taxa. This is, in part, because currently available programs fail to take advantage of parallelism.

Results: We present XMP, a new program for finding exact MP trees that comes in both serial and parallel versions. The serial version is faster in nearly all tests than existing software. The parallel version uses a work-stealing algorithm to scale to hundreds of CPUs on a distributed-memory multiprocessor with high efficiency. An optimized SSE2 inner loop provides additional speedup for Pentium 4 and later CPUs.

Availability: C source code and several binary versions are freely available from http://www.massey.ac.nz/~wtwhite/xmp. The parallel version requires an MPI implementation, such as the freely available MPICH2.

Contact: w.t.white@massey.ac.nz; barbara.holland@utas.edu.au

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on October 20, 2010; revised on March 16, 2011; accepted on March 17, 2011

1 INTRODUCTION

Of all the criterion-based approaches to evolutionary tree selection, the maximum parsimony (MP) criterion is the most intuitive: 'Select the tree or trees that require the fewest DNA substitutions'. The early popularity of MP was dampened by the discovery that it can be statistically inconsistent: in the 'Felsenstein Zone', increasing amounts of data will lead to increasingly certain recovery of the wrong tree (Felsenstein, 1978). Later, more general conditions leading to inconsistency were described (see Schulmeister 2004 for an extensive review). Elucidation of the assumptions made by MP led to the understanding that it is consistent when the expected number of changes per site, both across the tree and on any edge, are sufficiently small (Felsenstein, 2004; Steel, 2001, pp. 97–102). Steel and Penny (2000) chart the many variants of ML and their connections to MP. Of particular relevance is that MP is a most-parsimonious likelihood estimator for 'ample' datasets in which all taxa can be connected by a tree with maximum edge length 1, which is of practical significance when dealing with population (intra-species) data where taxa are highly similar (Holland *et al.*, 2005).

An often-overlooked fact is that the primary criticism levelled at MP—lack of statistical consistency in the general case—can be rectified through Hadamard conjugation (Penny *et al.*, 1996; Steel *et al.*, 1993). In particular, the Kimura three-parameter model (Kimura, 1981) and all its submodels can be corrected for directly using Hadamard conjugation, while any reversible model at all for which additive distances can be estimated can be dealt with via the Distance Hadamard transformation (Hendy and Penny, 1993). The resulting two-step approach, called corrected parsimony, is statistically consistent.

Although powerful heuristics for the MP problem have been developed (e.g. Goloboff 1999; Nixon 1999; Roshan *et al.* 2004; see also Felsenstein 2004, Chapter 4), they necessarily come without guarantees. We believe that a freely available, high-performance implementation of exact MP search is conspicuously absent, and would benefit the phylogenetics community. Our program, XMP, fills this gap. XMP offers:

- a parallelization strategy that is simultaneously highly portable across different computer architectures and highly efficient, scaling to hundreds of processors. This is the main new contribution;
- a new lower bounding approach for pruning unpromising regions of the search space, inspired by the MinMax Squeeze (Holland *et al.*, 2005);
- streamlined Fitch inner loop calculations using optimizations not published elsewhere; and
- a hand-optimized SSE2 assembly language implementation of the Fitch inner loop for Intel Pentium 4 and later CPUs, offering a potential 4-fold speed improvement.

The latter three improvements are also available in a single-processor version. We compare this version with a popular exact MP program and find that XMP is much faster in almost every case.

1.1 Existing implementations

Two branch and bound algorithms for exact maximum parsimony search were proposed by Hendy and Penny (1982). Their Algorithm I, which adds taxa to a partial tree one at a time, forms the basis of most current implementations of exact MP search (Bader *et al.*, 2006; Felsenstein, 1989). TurboTree (Penny and Hendy, 1987) took a different approach, inserting characters one at a time instead of taxa.

Hennig86 (Farris, 1989) was one of the earliest widely used parsimony programs offering an exact search feature.

*To whom correspondence should be addressed.

ExactMP (Bader *et al.*, 2006) is an exact MP search program designed to run in parallel on a shared-memory multiprocessor. For five 'hard' datasets, the authors achieve an average parallel speedup of 7.26 on eight processors using a queue-based locking and work distribution mechanism. In terms of absolute speed, Figure 4 of Bader *et al.* (2006), which compares the speed of ExactMP running on eight processors with PAUP* running on one processor, suggests that PAUP* is approximately four times faster than ExactMP on a single processor. We note that while ExactMP requires a shared-memory computer to run, XMP runs efficiently on both shared-memory computers and distributed-memory computers such as the BlueGene BG/L supercomputer or a networked cluster of commodity PCs.

PAUP* is a popular program for performing many types of phylogenetic inference. PAUP*'s exact MP inference is very fast—it is one of two programs that we use as a benchmark for XMP. While some of the strategies used by PAUP* have been published (Swofford *et al.*, 1996), unfortunately many algorithmic details have not.

PHYLIP (Felsenstein, 1989) is a freely available implementation of many phylogenetic inference methods, and includes the program dnapenny for performing exact maximum parsimony analysis. This program is typically much slower than PAUP*—on the same hardware, dnapenny took slightly over 20 min to find the unique minimal tree for the mt-10 dataset, while PAUP* took 44 s and the plain (non-SSE2) version of XMP took 7.3 s. The SSE2-optimized version of XMP took just 2.2 s.

TNT (Goloboff *et al.*, 2008) is a freely available maximum parsimony program explicitly focusing on efficient heuristic methods for large datasets. TNT also provides a fast exact search facility, which we compare with XMP.

Althaus and Naujoks (2006) take quite a different approach to the usual implicit enumeration scheme. Instead of adding individual taxa to a partial tree, they build sets of candidate rooted monophyletic groups in a first pass, and in a second pass form unrooted trees from all legal combinations of three such groups. Because the inputs to the second pass are monophyletic groups, the authors are able to draw on a variety of rules developed for general Steiner tree construction to eliminate tree topologies that cannot possibly be optimal. They order groups cleverly so as to minimize the number of legality tests performed.

Sridhar *et al.* (2008) take a different approach again, providing two different integer programming formulations of the MP problem: one that contains only a polynomial number of variables and constraints, and one that in the worst case may require an exponential number of both, but which in practice is faster to solve. They demonstrate impressive solution times, in one case solving a 34-taxon problem in less than one min. However, their program is restricted to alignments containing binary characters, and produces only one MP tree, rather than all minimal trees. Although the program currently requires the commercial mixed integer solver CPLEX, they provide free access to a web-based front-end at http://www.cs.cmu.edu/~imperfect/index.html.

In the remainder, we assume as input an aligned DNA dataset having n taxa and k sites. Informally, a dataset with many taxa is *tall*; a dataset with many sites is *wide*. The *length* of an edge or tree is the minimum number of point substitutions it requires; the *parsimony score* or *MP score* of a set of taxa is the minimum length of any tree interconnecting those taxa. XMP accepts ambiguous nucleotides and gaps, the latter being interpreted as 'any nucleotide'—the same as in PAUP* under default settings (GAPMODE=MISSING).

2 METHODS

2.1 Branch and bound for maximum parsimony

Although the maximum parsimony (MP) score can be determined in $O(nk)$ time using the Fitch–Hartigan algorithm (Fitch, 1971; Hartigan, 1973) when a tree is given, the problem of finding a tree whose MP score is minimal is NP-complete (Graham and Foulds, 1982), so it is unlikely that any algorithm exists that is asymptotically faster than enumerating and scoring all possible trees. Nevertheless, branch and bound (B&B) algorithms can offer a substantial improvement in practice. B&B is a general strategy for solving optimization problems that operates by exploring a *search graph* in which each node corresponds to a subproblem, and each arc links a subproblem with a child subproblem formed by adding constraints to the parent. This search graph is usually not represented explicitly but rather is implicit in the recursion structure of the program. In the usual formulation of B&B for MP search, first introduced by Hendy and Penny (1982), a subproblem is a binary tree on a subset of the full taxon set, and there is a child subproblem for each edge in the tree, indicating where the next taxon will be added. (For now, we leave aside the question of which taxon will be added next.) Full trees correspond to feasible solutions. The algorithm begins by calculating an upper bound on the parsimony score, often by building a heuristic MP tree. Then evaluation of subproblems takes place, starting with the original unconstrained problem, represented by the unique binary tree on some chosen set of three taxa. For each subproblem visited, a lower bound on the MP score of any solution reachable via that node is computed. Clearly adding taxa to a tree can never decrease its length, so it is acceptable to use parsimony scores of partial trees as lower bounds; later we look at stronger bounds. The utility of B&B hinges on the following observation: when a node is visited whose lower bound exceeds the current upper bound, it follows that it cannot lead to an optimal solution, so evaluation of its descendants can be skipped. This event is called a *cutoff* and the descendant nodes are said to be *pruned*, *bounded out* or *fathomed*. As the algorithm proceeds and more-parsimonious full trees are discovered, the global upper bound is reduced, further accelerating performance. In practice, the improvement over exhaustive enumeration is highly dataset dependent, but significant for typical biological datasets.

2.2 Branch and bound in XMP

The above description of B&B leaves several questions unanswered:

- Which three taxa should be chosen for the initial tree?
- In what order should taxa be added to the tree?
- In what order should subproblems be evaluated?

B&B involves adding taxa to a current partial tree in some order. At any point, given a partial tree containing $m < n$ taxa, we must choose (i) the next taxon to add and (ii) the order in which we should add that taxon to each of the $2m-3$ edges in the tree. As a general rule, the 'worst' taxon should be added next (so as to produce bound cutoffs as early in the search tree as possible), and that taxon should be added in its 'best' position first (so that new, tighter upper bounds are discovered sooner, possibly leading to earlier pruning of search trees stemming from other placements of the taxon). One important decision is whether to use a static or dynamic taxon addition order. A static order always adds taxa to the tree in the same order, while a dynamic order decides the taxon to add next just before the addition is to take place, so it can depend on the topology of the current tree. Although Purdom *et al.* (2000) indicates that using a dynamic taxon addition order can improve running times for tall datasets, we note that performance is highly variable across datasets, and it complicates the calculation of lower bounds on the length added by the remaining taxa. Also it is difficult to efficiently decide which taxon should be added next. We chose to use a fixed taxon addition order for XMP, which has the advantage of making lower bound calculations extremely fast (see Section 2.6). Where different implementation strategies are in conflict, we have generally favoured those that leave the B&B inner loop as streamlined as possible.

The order in which taxa are added to the tree is critical to performance (Hendy and Penny, 1982; Purdom *et al.*, 2000), with different orders easily leading to running time disparities of many orders of magnitude. XMP uses the standard max–mini approach (Nei and Kumar, 2000): first, an exhaustive search is used to find the three-taxon tree whose length is greatest; then $n-3$ rounds take place, in each of which the taxon t_i whose minimum length increase across all edges is maximum across all remaining taxa is added to the tree by inserting it at its minimum-length edge. This heuristic attempts to order taxa so that every tree built on any initial sequence of taxa has length as great as possible, in order to force early cutoffs.

Regarding the order in which the chosen taxon is added to the edges of the current partial tree, XMP takes the simple approach of using a preorder depth-first search (DFS) tree traversal, which is convenient for recursion. In the typical case where the initial upper bound is optimal (which was the case for all the datasets used for performance testing), the order in which placements are tried has no bearing on the set of search nodes evaluated, so this simple order suffices.

2.3 Parallelization strategy

A simple way to subdivide the original B&B problem is to have a set of worker processes that request work from a boss process whenever they are idle, while the boss enumerates complete trees down to some small depth (number of taxa) and sends each subproblem to a requesting worker to solve, until all starting trees have been exhausted. However, the subproblems generated this way may vary greatly in difficulty, leading to enormous imbalances in solution time: it can happen that all but one process finish work in milliseconds and sit idle, while one process continues working for months. One way around this is to arrange for the boss never to hand off the last subproblem, but rather subdivide it by inserting the next taxon at each edge, and hand off one of the generated subproblems instead. We found that although this sometimes improves behaviour, a different problem can arise: after the boss begins additional subdivision, subproblems issued to workers become smaller and smaller to the point where running time is dominated by communication between boss and workers. In effect, the situation approaches a serial B&B enumeration in which every node evaluation requires round-trip communication between a boss and a worker. Since communication latency typically dwarfs the time needed for a single node evaluation, performance drops dramatically.

Bader *et al.* (2006) confine themselves to a shared-memory multiprocessor, which affords them the ability to investigate advanced locking and lock-bypassing priority queues for managing subproblems. Approaches based on priority queues guarantee a minimum number of node evaluations, but increase the time needed for each node to be evaluated and complicate load balancing. Provided that the initial upper bound is tight, a simple depth-first search will evaluate exactly the same nodes (most likely in a different order), but is simpler, enables fast incremental tree modifications and has better locality properties. So XMP forgoes priority queues entirely, trusting that the initial TBR search phase (see Section 2.6) will bring us a near-optimal upper bound, and opts for a simple load-balancing approach that works on both shared-memory and distributed-memory multiprocessors: *work stealing* (Blumofe and Leiserson, 1999). A single boss process starts with the original tree on three taxa, and hands the entire problem to the first worker process that requests work. Every subsequent work request made to the boss causes it to choose a non-idle *victim* worker at random and steal a job (subproblem) from it to pass back to the requesting worker, or *thief*. Intuitively, work stealing never performs unnecessary communication, and provided steal victims are chosen randomly, it has excellent performance characteristics in terms of expected total execution time (Blumofe and Leiserson, 1999). All requests in XMP go via the boss, which simplifies termination detection and makes it easier to guarantee starvation-free servicing of workers. Although boss-free, truly distributed work stealing can be achieved on a distributed-memory computer by using a one-sided communications protocol that does not require synchronization with the destination, the necessary hardware support (Remote Direct Memory Access or RDMA) is often inadequate and must be emulated at additional cost: tellingly, in their study on the performance of distributed work stealing, Dinan *et al.* (2009) set aside one of every eight CPUs (apparently without adjusting their efficiency measurements) for a 'data server' process just to emulate efficient one-sided communication, despite the fact that their network technology, InfiniBand, nominally supports RDMA. XMP's centralized strategy does place an upper limit on scalability, though as our results show, surpassing this limit requires hundreds of processors.

The parallel version of XMP uses the industry standard MPI message-passing interface for handling all communication between processes. Using MPI rather than a thread-based approach enables XMP to compile and run on a wide variety of systems. For maximum efficiency, we use only non-blocking sends and receives, allowing computation and communication to proceed simultaneously whenever possible, and the boss uses `MPI_Waitsome()` to prevent starvation of workers. For portability, the program makes no assumptions about shared accessibility of files across processes.

In addition to requesting work when they are idle, workers announce improved upper bounds they discover to the boss, which broadcasts them. Workers poll `MPI_Test()` to detect incoming UB changes and steal requests. We use the adaptive polling interval technique advocated by Dinan *et al.* (2008) to balance computational throughput with responsiveness, using parameters $i_{max}=1024, i_{min}=1, i_{inc}=1, i_{dec}=2$. Briefly, the polling interval increases linearly up to a maximum while no message is received, halving upon receipt of a message.

Despite its simplicity, our boss worker formulation maps neatly to both 'big' and 'small' multiprocessors:

- on computers with only a few processors, such as modern desktop workstations, the boss process spends almost all its time waiting, and consequently takes up very little CPU time. This effectively leaves an 'extra' CPU free to allocate to a worker—i.e. in order to use the entire capacity of such a machine, XMP should be run with the number of processes set to one more than the number of CPUs. As our results show, the task switching that occurs when the boss needs to service a request requires very little overhead; and
- computers with many processors, such as the IBM BlueGene series of supercomputers, commonly mandate a fixed allocation of processes to CPUs. In this case, time that the boss process spends waiting is necessarily wasted. However, due to the scale of these systems, the boss has many more requests to service and thus spends little time idle, so that typically only a fraction of one CPU is wasted.

2.4 Subdivision into jobs

It seems natural to define subproblems as partial trees on $m<n$ taxa: expanding a subproblem is then done by inserting the $(m+1)$-th taxon at each of the $2m-3$ edges in turn. Although ideal for serial computation, this scheme causes problems for parallel implementations because it is difficult for steal victims to 'break off' large *jobs* (sets of subproblems) to send back to thieves, which is necessary for minimizing communication overhead. This problem can be overcome by adopting a finer notion of subproblem that constrains the set of edges at which the next taxon can be inserted, along with a novel representation for sets of subproblems.

XMP uses a *remaining edge pair list* (REPL) to compactly encode a family of subproblems formed by stripping taxa in reverse order from a base partial tree. A job whose base tree contains m taxa is represented by a list of $m-2$ integer pairs; the elements of the i-th pair identify a range of edges in a partial tree of size $i+2$. Edges are numbered in DFS preorder. To understand the REPL, first note that given (a) an initial tree on three taxa, (b) a list of remaining taxa to be inserted in order, and (c) a fixed policy for numbering edges, then a list of m edge indices uniquely identifies a tree on the first $m+3$ taxa by interpreting each edge index as identifying the edge to insert the next taxon at. The set of subproblems represented by an REPL containing m pairs $(L_i, R_i), 1 \le i \le m$ may be described recursively as follows:

(1) If $L_m > R_m$, then no subproblems are included; otherwise, construct the $(m+2)$-taxon tree from the edge index list $L_{1...m-1}$ as described above. The subproblem corresponding to this partial tree, with taxon $m+3$ constrained to be inserted at an edge having index in the range $L_m \ldots R_m$, is included; and

(2) If $m > 1$, remove the final pair from the list and increase L_{m-1} by 1. All subproblems that would be included by this new REPL are also included.

The initial problem—a tree on three taxa in which the fourth can be inserted at any edge—is given by $(0,2)$. Each worker maintains an REPL as it enumerates its B&B tree: every time a taxon, say the m-th, is added to a partial tree containing $m-1$ taxa, a pair $(0, 2m-2)$ is appended to the list, representing the $2m-3$ edges at which the next taxon can now be inserted; whenever the m-th taxon is removed, the last pair in the list is removed, and the first element of the new final pair is incremented.

REPLs can be quickly updated during local B&B search, and their compactness reduces the size of job messages. But their primary advantage, and the key to dealing with steal requests effectively, is that an REPL can be easily partitioned into two REPLs representing disjoint sets of subproblems having the following properties:

- The B&B recursion exploring the original REPL can continue exploring one of the new REPLs; and
- The other REPL contains a largest possible subproblem (smallest possible partial tree).

When a steal request arrives, the victim can quickly discover a subproblem corresponding to the smallest partial tree that it has available by scanning its REPL for the first edge range (i,j) with $i<j$. Suppose this is the m-th pair. The REPL to send back consists of the first $m-1$ edge ranges (which necessarily have both edge indices equal) plus the edge range (j,j). The victim then sheds this job from its own workload by decrementing R_m in its own REPL. This can be done in $O(m)$ time.

Upon receipt of a new job REPL, a thief assembles the base tree from the list by 'fast-forwarding' the usual B&B enumeration process—at each tree size, taxon insertion is simply skipped for any edge having index less than the corresponding L_i. Normal B&B then resumes.

2.5 Coping with complexity

A number of factors contribute to complexity in the parallel code: tracking the states of workers, the necessity for both worker-initiated and boss-initiated communications, reliably detecting termination and our desire to use non-blocking I/O for efficiency. That complexity led to bugs. Bugs in parallel software can be nightmarish due to the difficulty in reproducing them, so we decided to model the communicating system of processes using the MPI-Spin extension to the model checker Spin (Holzmann, 1997; Siegel, 2007). This excellent tool caught one obvious, and two extremely subtle bugs which we subsequently fixed. After heavy optimization of the Spin model, the final run examined 1.4 billion state transitions and required 52 min and 44 GB of RAM to confirm that every possible interleaving of execution sequences involving one boss and two workers is free of deadlocks and other assertion violations—a very strong indication that the program works correctly with any number of workers.

The remaining subsections concern topics that apply to both serial and parallel versions of XMP.

2.6 Upper and lower bounds

By default, XMP initially attempts to find a tight upper bound on the MP score by performing greedy (hill-climbing) TBR branch swapping on 100 trees produced by random addition order. A minimum spanning tree is also calculated, although this usually produces a loose bound.

During the B&B phase, lower bounds on MP scores are needed for each partial tree examined. The MP score of the partial tree is an admissible but usually suboptimal bound; XMP employs several strategies for improving on this that are described in the following subsections. In each case, we are given a partial tree on $m<n$ taxa and tasked with finding a lower bound on the length that must be added by inserting the remaining $n-m$ taxa in some fashion; this can be added to the MP score of the partial tree to get a lower bound on the MP score of any full tree that is reachable from it. XMP takes the standard approach of calculating bounds that depend only on the taxa present in the tree, and not on the tree topology. Because XMP uses a fixed taxon addition order, only $n-2$ different subsets of taxa will ever be encountered, meaning that all lower bounds can be precomputed and stored in a lookup table for speed.

The parallel version of XMP does not parallelize the initial computation of upper and lower bounds. Instead each worker permutes sites randomly before evaluating bound heuristics, and the overall best bounds of each kind are retained. This occasionally produces superlinear speedups.

2.7 Single column discrepancy lower bound

Consider a partial tree containing some subset of the taxa, and a site containing d distinct nucleotides. If only $e<d$ distinct nucleotides appear at that site among the taxa in the tree so far, then the remaining $d-e$ nucleotides must be added by the remaining taxa, with each distinct nucleotide incurring a cost of at least 1 substitution. These lower bounds can be summed over all sites to produce an overall *single column discrepancy* (SCD) lower bound that is cheap to compute and leads to speedups of typically 1.3 to 2.4 for static taxon addition order (Purdom *et al.*, 2000).

In practice, the algorithm is complicated slightly by ambiguous nucleotides. To handle these, we split the problem into computing an upper bound on a given subset of taxa, and a lower bound on the entire taxon set: the difference is a lower bound on the length that must be added by the remaining taxa. Upper bounds for a single site are found by representing ambiguous nucleotides as *state sets* of unambiguous nucleotides, and subtracting the frequency of the most commonly occurring nucleotide from the number of taxa. This is equivalent to determining the most frequently occurring nucleotide x, then constructing a new, unambiguous site in which every state set containing x is replaced with x itself and every other state set S is replaced with any nucleotide $y \in S$, and finally applying equation 2 of Steel and Penny (2005) to this new site. This formula produces optimal upper bounds in the absence of ambiguous nucleotides, and good quality bounds in other cases. Tight single-site lower bounds can be found by solving a maximum set packing problem. We performed this in an offline step for each of the 2^{2^4-1} possible sets of distinct ambiguous nucleotides that could be present at a site, and stored the results in a lookup table. The SCD lower bound can be requested with the `-Bd` option to XMP.

2.8 Incompatibility lower bound

The I-bound of Holland *et al.* (2005) exploits the fact that every non-overlapping pair of incompatible sites must increase the length of an MP tree by at least 1, and sometimes more. This bound has the advantage that it can be added to the SCD bound to produce a stronger lower bound. XMP provides a similar bound via a greedy approximate maximum matching algorithm for finding incompatible site pairs, which is available using the `-Bi` option.

2.9 PARTBOUND lower bound

The MinMax Squeeze (Holland *et al.*, 2005) attempts to produce provably optimal MP trees by pushing a lower bound on MP scores up until it meets the length of trees found heuristically. That work extends the lower bounding technique first established as the Partition Theorem by Hendy *et al.* (1980), which essentially states that the MP score of a dataset must be at least the sum of the MP scores of each part in a sitewise partition of the dataset. XMP contains a new lower bounding technique based on this approach, PARTBOUND, available via the `-Bp` option. Rather than maximize the overall lower bound on the first m taxa, we seek partitions that maximize the sum of the *final scores* for each part. The *raw score* of a part π is $LB(\pi,n)-UB(\pi,m)$, where $LB(\pi,i)$ is a lower bound on the MP score of any tree on the first i taxa, restricted to the sites in π, and $UB(\pi,i)$ is an upper bound defined similarly. Raw scores may be negative. The final score of a part is the greater of the raw score and the SCD bound for the first m taxa summed over all sites in the part. Final scores are always non-negative, and their sum can be safely added to the MP score of any tree built on the first m taxa. Starting from the trivial partition (one site per part), XMP searches partition space with a greedy site-swapping heuristic that attempts to increase the total final score—or, when impossible, the total raw score—until no final score improvement has been made for two iterations.

Part upper bounds are calculated as for the SCD bound. The challenge is to calculate good lower bounds: XMP uses several strategies, choosing the best for a given part. After identical sequences are collapsed, the Kruskal algorithm (Kruskal, 1956) is used to find the largest 1-connected components. If one or two components result, the length of the minimum spanning tree (MST) is used; this bound dominates the D-bound of Holland *et al.* (2005) when one of the edges has length >2. If three or more components result, then the lengths of Steiner trees (which may or may not have a single Steiner vertex) on all possible sets of three components are calculated, and the longest chosen. (Again, ambiguous nucleotides prove an annoyance since distances between sequences containing them may violate the triangle inequality; nevertheless, it can be shown that these constructions yield valid lower bounds—see the Supplementary Material.) Finally, the length of an MST is at most twice the length of a Steiner tree in any metric space, so finding an MST and dividing the length by 2 yields another lower bound on the MP score (Proposition 5.4.1 in Semple and Steel, 2003). For pairs of sites containing no ambiguous nucleotides, there is no need to divide by 2 (Bruen and Bryant, 2008).

2.10 Fast fitch parsimony

Ronquist (1998) describes how Fitch parsimony operations can be accelerated by encoding state sets as bit vectors and storing multiple sites in a machine word. In the terminology of that paper, XMP horizontally packs four-bit state sets into machine words of 32, 64 or 128 bits in width. He also observes that modern superscalar, deeply pipelined CPUs penalize unpredictable conditional branches in program code—a trend that has become more severe in the years since. The Fitch algorithm (Fitch, 1971) tests whether two state sets have a non-empty intersection and thus appears to require such a branch; however, he offers several algorithms that cleverly sidestep the problem by using bit masking techniques.

Although Algorithm 8 of Ronquist (1998) increases performance by using only predictable conditional branches, it does not achieve the full potential of this approach because it still loops over each possible state a site may take. The algorithm on p. 271 of Goloboff (2002) improves matters slightly by ‘unrolling’ the loop, but XMP boosts speed further by eliminating all per-state calculations. As the following C code shows, the trick involves exploiting the carry produced by binary addition:

```
u = ((((x & y & 0x77777777) + 0x77777777) |
      (x & y)) & 0x88888888) >> 3;
z = (x & y) | ((x | y) &
    ((u + 0x77777777) ^ 0x88888888));
```

Here `x` and `y` are 32-bit words each containing blocks of 8 input state sets from two child sequences, and `z` is assigned the resulting block of 8 output state sets for the parent sequence. `u` will have each 4-bit nibble set to 0001 if the corresponding site necessitated a mutation, and 0000 otherwise; it is also used for computing length increases. `&`, `|`, `^` and » are the operations AND, OR, XOR and right-shift, respectively, and numbers beginning with `0x` are in hexadecimal. We deliberately leave in common subexpressions like `(x & y)`, trusting the compiler to do a better job of deciding when and how to evaluate them than we could.

To understand the process, consider a single site (nibble). We compute the intersection of the state sets `(x & y)` and mask out the leftmost bit; adding the binary value 0111 to this value will produce a 4-bit sum whose leftmost bit is 1 if and only if any of the remaining 3 bits are 1, thereby detecting whether `x` and `y` share any states in the set {`A`,`C`,`G`} with a single machine instruction. ORing this value with the original intersection nibble produces a 4-bit value whose leftmost bit is 1 if and only if any of the original 4 intersection bits were 1, i.e. if `x` and `y` share any states at this site. The reason for masking out the leftmost bit of the intersection nibble is to guarantee that adding 0111 cannot cause a carry into the nibble to the left: this permits a single 32-bit addition to perform shared-state detection for all 8 input state sets in parallel. This general technique was apparently first discovered by Lamport (1975), who credits D.E. Knuth.

With knowledge of whether the two taxa share any states at a given site now stored in the leftmost bit within the corresponding nibble, further masks and shifts can be used to compute `u` and the resulting Fitch state sets `z`. To turn the `u` values into the required all-1 or all-0 masks, XMP uses the O(1) calculation `((u + 0x77777777) ^ 0x88888888)`, which is faster than the repeated shifting and ORing described by Ronquist (1998) and Goloboff (2002). Because all operations respect nibble boundaries, the entire Fitch calculation can be performed in parallel across all eight sites as with earlier algorithms. We speculate that some of these techniques are already used in some existing MP programs—for example, in a personal communication note Goloboff (2002, p. 272) attributes to Farris unspecified optimizations that produce ‘similar results …with about half the operations’—but they do not seem to be explicitly described in the phylogenetics literature.

In an initial step, XMP discards parsimony-uninformative sites and condenses the remaining groups of equivalent site patterns into individual weighted columns. By sorting sites in decreasing order of weight, we enable two additional shortcuts: (i) we can exit the innermost loop as soon as the length added to the tree exceeds the current bound, which is more likely to happen early on; (ii) we can swap over to a faster bit-counting algorithm for computing the remaining cost as soon as all weights in a block have dropped to 1, since from that point on all sites must have this weight. By default, the number of 1-bits in a machine word is counted using a multiply-mask-shift sequence, although an alternative implementation can be selected that sums adjacent nibbles, then adjacent pairs of nibbles, and so on. Both approaches take a small, fixed time and are likely to be faster than either of the iterative schemes detailed in Ronquist (1998). The lookup table approach suggested by Moilanen (1999) and used by Bader *et al.* (2006) may be faster, but we concluded that using a large chunk of memory to hold a lookup table was likely to degrade cache performance unnecessarily.

Goloboff (1993) describes a way to speed up parsimony searches by avoiding a complete first-pass Fitch optimization for each taxon insertion, enabling amortized $O(k)$ Fitch scoring of taxon insertions. For correct handling of ambiguous nucleotides at leaf nodes, a workaround is required (Goloboff, 1996, pp. 204–205), though this does not impact the time complexity. XMP uses a similar scheme by Yan and Bader (2003) that handles this situation without additional bookkeeping. We enhance this by applying

Shortcut C of Goloboff (1996, pp. 209–211) to eliminate unnecessary second-pass recursion. Finally, we noticed that as Fitch performance increased, proportionally more execution time was spent on bookkeeping overhead. Exploiting the fact that all memory allocations occur in LIFO order during the main B&B phase, we allocate a block of memory beforehand and thereafter use single pointer additions and subtractions for quickly allocating memory from this block when needed.

2.11 SSE2 optimized version

The SSE2 instruction set, available on Pentium 4 and later CPUs, includes instructions for operating on 128-bit quantities. Particularly on Core2 and later CPUs where most of these instructions execute in 1 clock cycle, this offers a potential 4-fold performance improvement over the usual 32-bit operations. We developed an optimized version of the Fitch inner loop using hand-coded SSE2 assembly language, which, like the regular C-code version, avoids conditional branches for maximum performance. Due to syntax differences, the SSE2-optimized version is currently only available for Windows compilers.

3 RESULTS

XMP was run on the real and synthetic datasets from Bader *et al.* (2006) as well as three other real datasets ranging in height, width and difficulty level. The leftmost five columns in Table 1 summarize the datasets. Performance was measured in the following environments: a quad-processor 2.66 GHz Core2 Windows XP PC using MPICH2, an 8-processor 3.2 GHz Linux PC using MPICH2 and the BlueFern BlueGene/L (BG/L) supercomputer at the University of Canterbury, using its proprietary implementation of MPI. BlueFern is a distributed-memory supercomputer, affording the chance to test XMP's ability to scale up across hundreds of CPUs without the benefit of fast access to a central memory store. For brevity, we report only a subset of results; see the Supplementary Material for a fuller picture.

Table 1. Datasets and performance of XMP `-Bdi` on 8-CPU SMP

Dataset	Taxa	Sites	MP length	Trees	T1	T8W	S/Up
h1	12	64	364	1	42.73	5.82	7.34
h2	12	64	367	2	34.43	4.72	7.29
h3	12	64	359	1	5.31	0.77	6.90
h4	13	64	397	2	215.24	29.16	7.38
h5	13	64	396	3	264.88	35.91	7.38
mh1	20	64	124	1134	14.86	2.16	6.88
mh2	20	64	192	3	7.03	1.09	6.45
mh3	20	64	118	*	873.94	123.97	7.05
mh4	20	64	303	5	9.97	1.47	6.78
mh6	20	64	128	612	24.41	3.45	7.08
e1	24	500	593	6	0.13	0.16	0.81
e3	24	500	589	36	0.11	0.18	0.61
e4	24	500	584	3	0.07	0.13	0.54
e5	24	500	577	3	0.12	0.14	0.86
e6	24	500	579	2	0.09	0.15	0.60
Eukar	27	2461	3512	60	1.55	1.35	1.15
rbc14	14	759	963	2	22.44	3.06	7.33
Metaz	20	1008	825	3	23.81	3.33	7.15
its36	36	607	233	62370	1552.64	192.92	8.05
mt-10	10	10539	16179	1	4.84	3.29	1.47
32hum	32	202	95	*	2269.51	301.66	7.52

Leftmost five columns describe datasets used; rightmost three columns give performance of XMP `-Bdi` on 8-CPU SMP. T1: Total elapsed time in seconds for serial version. T8W: total elapsed time in seconds for 8-worker parallel version (9 MPI processes). S/Up: speedup (T1/T8W).
* indicates more than 100 000 trees; only the first 100 000 were saved.

3.1 Serial performance

Figure 1 compares the times of the single-processor version of XMP with PAUP* and TNT on Windows XP. Four variants of XMP are considered: '-Bd' indicates that only the SCD bound was used; '-Bdi' indicates the sum of this bound and the incompatibility bound; '-Bp' indicates the new PARTBOUND bound; and '-Bp SSE2' indicates the SSE2-optimized implementation of this bound (all others use the portable C version). For parity with XMP's upper bounding strategy, PAUP* runs first used a corresponding HSEARCH command to find upper bounds; TNT does not allow the initial upper bound to be specified for an exact search, so no corresponding attempts could be made to improve initial upper bounds for this program. We note that both XMP and PAUP* found a tight upper bound in their initial TBR phase each time. An upper limit of 100 000 trees was set.

As Figure 1 shows, on all but three datasets XMP `-Bdi` is faster than PAUP*, often by a considerable margin. For mh6, 32hum and its36, XMP `-Bdi` is 50, 19 and 14% slower, respectively. For its36, XMP `-Bp` is faster, beating PAUP* by 18%, while for the other two datasets, `-Bp` is around 1% slower than `-Bdi`.

On all but two datasets, XMP `-Bdi` is faster than TNT, again often by a significant margin. For mh2, XMP `-Bdi` is 8% slower, while for mh3, TNT is dramatically faster than both XMP `-Bdi` and PAUP* at 260.61 s versus 1299.92 s and 8673.11 s, respectively. On the other hand, while XMP `-Bdi` and PAUP* take just 35.12 s and 23.42 s for mh6, respectively, TNT takes 2922.51 s. TNT regularly runs in under half the time of PAUP* for mid-range datasets, but performs badly on the largest two datasets.

We find that the performance on many datasets is actually affected very little by the choice of lower bound strategy. This seems surprising at first, since in the tables of lower bounds computed for each strategy, in all cases the entries near the start (being the bounds for small partial trees) show the `-Bdi` bound to greatly exceed the corresponding `-Bd` bound, and the `-Bp` bound to be greater still (data not shown). However as the partial tree size increases, the bounds for each strategy necessarily decrease, eventually becoming equal at some tree size. If a partial tree grows to that size under the `-Bdi` or `-Bp` bound without being eliminated, then it and its subproblems will be evaluated as with the simple `-Bd` bound; it is in this region of the search space that B&B presumably spends most of its time.

PARTBOUND (`-Bp`) is clearly a loss for very wide datasets such as mt-10 and Eukar. By comparison, `-Bdi` never shows excessive overhead, and is beaten by `-Bp` only for the its36 and rbc14 datasets, where it does not do much worse.

Unsurprisingly, the SSE2-optimized version of XMP is everywhere faster than the portable C version. Generally, the speedup increases with the width of the dataset, since this translates to a greater proportion of time spent in the innermost loop. The clearest example is the mt-10 dataset, although the total running time as shown on Figure 1 is dominated by the PARTBOUND calculation. Considering just the B&B time components for the mt-10 `-Bp` runs, the portable C version requires 6.05 s while the SSE2 version requires just 1.38 s—an improvement of roughly 4.4 times,

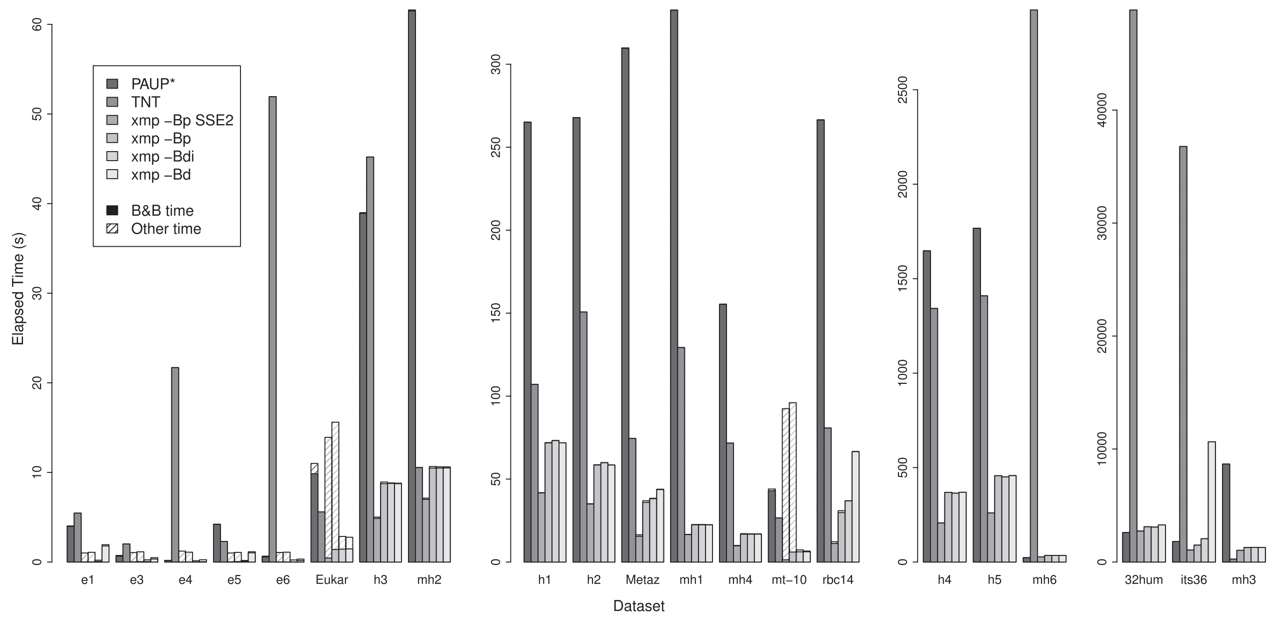

Fig. 1. Execution time of XMP versus PAUP* and TNT on 1 CPU. Each named dataset is analysed using exact search with PAUP*, TNT and four variants of XMP. The time taken for each run is shown as a sum of B&B time (solid) and other time (lined) components; for most runs, other time is insignificant. Separate time axes show detail for short, medium, long and very long runs. The underlying data are available in the Supplementary Material.

exceeding the 'theoretical' limit of 4-fold improvement. This is presumably due to inefficiencies in the compiler-generated code for the portable C version.

3.2 Parallel performance on eight CPUs

The rightmost three columns of Table 1 show the performance of XMP when run on the 8-CPU Linux machine using the `-Bdi` option. The T1 column shows the total time in seconds for the serial XMP version, and the T8W column shows the total time for the parallel version using 9 MPI processes (1 boss and 8 workers). The S/Up column is the speedup or ratio. This setup is similar to that used by Bader *et al.* (2006). For the five datasets labelled 'hard' in that paper, we find an average speedup of 7.258, almost exactly equalling the result obtained there. However, the 'moderate' and 'real' groups fare much better with XMP, achieving speedups of 6.847 and 5.210, respectively, while Figure 3 of Bader *et al.* (2006) shows that ExactMP obtains speedups of less than 6 and less than 5, respectively. The slowdown observed for the 5 'easy' datasets is simply a consequence of the fact that the B&B phase for these datasets takes much less than 1 s, meaning that overall runtime is dominated by one-time overheads such as communicating input data to all workers. In our article, all speedups and efficiencies are calculated with respect to the serial version.

Interestingly, the its36 dataset experiences a superlinear speedup due to the discovery of better lower bounds. We found this phenomenon occurred only rarely in our experiments.

3.3 Parallel performance on hundreds of CPUs

We ran each dataset using various numbers of CPUs on the BG/L in Virtual Node mode, which gives XMP control of both CPUs on each BG/L compute node. Figure 2 shows the efficiency of each

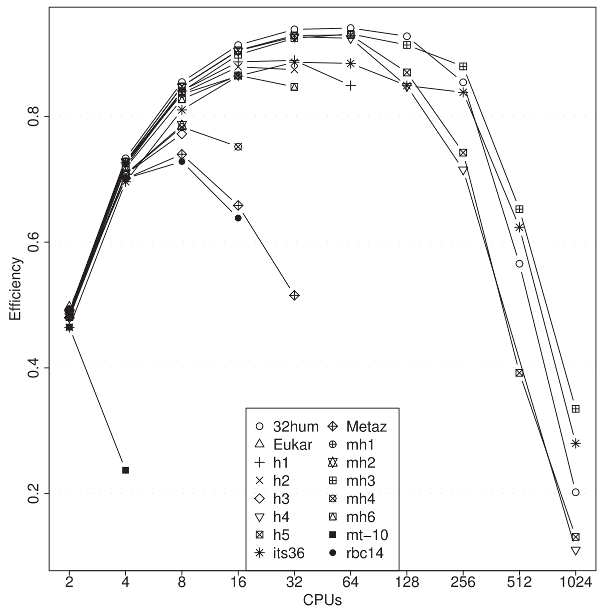

Fig. 2. Efficiency of XMP `-Bp` on the BG/L supercomputer as the number of CPUs is increased. An efficiency of 1.0 indicates overhead-free parallelization with respect to the serial version. Each dataset is shown separately; only runs taking >5 s are shown.

run that required more than 5 s. Efficiency is defined as $t_1 \div (mt_m)$, where m is the number of CPUs, t_1 is the elapsed time used by the serial version and t_m is the elapsed time used by the m-CPU parallel version. Because BG/L CPUs are always assigned exactly one process to run, the boss process necessarily consumes one full

CPU, so efficiency starts out near 0.5 for two CPUs and climbs. We show only results for the `-Bp` lower bound strategy; this can be considered conservative, since all lower bound processing takes place in serial beforehand, and `-Bp` has the highest overheads for this phase.

For the five hardest datasets (mh3, its36, 32hum, h5 and h4), we find that efficiency remains well above 0.8 for up to 128 processors, and for three of these datasets it remains above this level even for 256 processors. Above this point, performance begins to drop off sharply, presumably due to saturation of the boss node.

As a final test, we compared the running times of XMP `-Bp` on 256 BG/L CPUs and PAUP* on a 2.66 GHz Core2 CPU, using progressively larger subsets of the 53humans dataset from Holland *et al.* (2005). (Our 32hum dataset consists of the first 32 taxa from this dataset.) We find that analysing the first 39 taxa takes 5452.56 s (roughly 1.5 h) to complete with XMP in this configuration, while the same dataset takes PAUP* 168341 s (roughly 46 h, 45 min). The ratio of elapsed times improves from 27.17 at 32 taxa to 30.87 at 39 taxa, suggesting that the parallel version of XMP continues to become more efficient as problems grow in size. Adding one more taxon to the dataset results in XMP taking 19 h, 55 min; the corresponding PAUP* analysis was aborted after 2 weeks, but is projected to require more than 25 days—an impractical amount of time for most researchers to spend on a single analysis.

4 DISCUSSION AND CONCLUSIONS

XMP is at least as scalable on shared-memory multiprocessors as ExactMP, is faster in absolute terms and also runs efficiently on distributed-memory multiprocessors where ExactMP will not run at all. On almost all datasets tested, the serial version of XMP convincingly beats TNT, which in turn is faster than PAUP* on a majority of datasets, although there remain cases where PAUP*—a program now 9 years old—still holds out. Naturally, we were very interested in discovering the internal workings of TNT and PAUP*, but very little solid information could be found. We agree with Goloboff (1993) and Ronquist (1998) in calling for details of fast computational techniques to be made public, with the goal of advancing the state of the art: it seems likely that a person with knowledge of all three programs could design a program that outperforms all of them. We hope that the strategies, algorithms and tricks described in this article, and the free source code to XMP, contribute towards this goal.

Because of the superexponential complexity of exact MP search, the improvements realized in XMP will not usually allow many more taxa to be analysed, but they do dramatically increase the speed of existing searches. For example, our analysis of the its36 dataset on a modern Linux PC takes 17 min, 14 s using 1 CPU. This drops to 2 min, 23 s when all 8 CPUs of the machine can be used. Using 256 CPUs on a BG/L, the analysis takes only 38.55 s. These speed increases will also accelerate heuristic searches for large datasets that internally rely on exact searches on subsets of taxa, such as Rec-I-DCM3 (Roshan *et al.*, 2004), or the sectorial search of Goloboff (1999) when configured to use exact search for small sectors.

While raw performance is important, it must be noted that mature programs like PAUP* and TNT offer a fuller set of features than XMP currently does. Perhaps the most important feature absent from XMP but offered by both PAUP* and TNT is the ability to collapse edges according to various criteria, which can lead to sizeable reductions in output. This can still be done by an external program after an XMP run completes, but it is an inconvenience for the user. Other advantages of these existing programs include the ability to impose topology constraints, and to save suboptimal trees.

Holland *et al.* (2005) suggest an application of the MinMax Squeeze to B&B search, which XMP fulfils. The `-Bp` partition bound achieves modest speedups on two datasets, but otherwise does not materially improve execution times, and for wide datasets like mt-10 actually produces an overall slowdown. This is despite the fact that the lower bounds produced for each tree size are always greater than or equal to those produced by other bounds (data not shown). These results suggest that `-Bdi` is a good default setting, with `-Bp` as an option to consider for larger datasets.

4.1 Possible future directions

Section 4 of Bachrach *et al.* (2005) describes a lower bound based on a circular ordering approximation algorithm for the Path-Constrained Travelling Salesman Problem, a generalization of the TSP. This bound is unlike the lower bounds used in XMP in that it depends on the topology of the partial tree, which makes it not only potentially stronger but also much slower to compute. It would be interesting to incorporate this bound into XMP.

Felsenstein (2004; pp. 65–66) discusses rules for reducing the search space needed for exact MP search, which he attributes to a Russian language paper by A.Zharkikh, 1977. However, some of these rules do not appear to be compatible with XMP's enumeration scheme, which produces trees that are always binary but may contain branches that have length zero under every possible assignment of mutations to edges (A.Zharkikh, J.Felsenstein, personal communication). In order to accommodate Zharkikh's rules, two approaches seem possible: either avoid introducing zero-length edges into partial trees in the first place or create a collapsed copy of each partial tree considered and apply the rules to it. The former approach can be realized by performing binary tree B&B as usual but ignoring all zero-length edges (and, in particular, never inserting a taxon into such an edge). This would speed computation by potentially reducing the number of partial tree subproblems spawned by any given parent subproblem, but it can be shown that doing so sometimes causes (collapsed) MP trees to be missed. In contrast, the latter approach is safe, but whether the overhead entailed would pay for itself is likely to be dataset dependent.

Regarding applications, we look to corrected parsimony (Hendy and Penny, 1993; Penny *et al.*, 1996; Steel *et al.*, 1993). Despite outperforming other non-ML methods in simulation tests (Charleston *et al.*, 1994), it appears that no reconstruction-accuracy comparison of corrected parsimony with ML has yet been done. We think this is an important oversight. Although XMP currently requires integer site pattern weights, the non-integer weights involved in corrected parsimony can be accommodated by scaling and truncation. In contrast to ML methods, which rely on heuristics like the Nelder–Mead algorithm (Nelder and Mead, 1965) to optimize the final ML score, this approach presents the intriguing possibility of recovering trees in which the error in the optimality criterion (total tree length) is *bounded*. Higher scaling and lower truncation thresholds produce tighter bounds at the expense of wider datasets and increased running time.

ACKNOWLEDGEMENTS

We wish to thank Joe Felsenstein, Andrey Zharkikh and Mike Hendy for helpful discussion concerning when an edge must, or must not, appear in all MP trees; David Bader and Rouven Naujoks for helpfully supplying datasets and program code; and the University of Canterbury's BlueFern supercomputing facility.

Conflict of Interest: none declared.

REFERENCES

Althaus,E. and Naujoks,R. (2006) Computing steiner minimum trees in Hamming metric. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, ACM, Miami, Florida.

Bachrach,A. *et al.* (2005) Lower bounds for maximum parsimony with gene order data. In *Comparative Genomics*, Vol. 3678 of *Lecture Notes in Computer Science*, pp. 1–10.

Bader,D.A. *et al.* (2006) ExactMP: an efficient parallel exact solver for phylogenetic tree reconstruction using maximum parsimony. In *Proceedings of the International Conference on Parallel Processing*, pp. 65–73.

Blumofe,R.D. and Leiserson,C.E. (1999) Scheduling multithreaded computations by work stealing. *J. ACM*, **46**, 720–748.

Bruen,T.C. and Bryant,D. (2008) A subdivision approach to maximum parsimony. *Ann. Combinatorics*, **12**, 45–51.

Charleston,M.A. *et al.* (1994) The effects of sequence length, tree topology, and number of taxa on the performance of phylogenetic methods. *J. Comput. Biol.*, **1**, 133–151.

Dinan,J. *et al.* (2008) A message passing benchmark for unbalanced applications. *Simul. Model. Pract. Theory*, **16**, 1177–1189.

Dinan,J. *et al.* (2009) Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, Portland, Oregon.

Felsenstein,J. (1978) Cases in which parsimony or compatibility will be positively misleading. *Syst. Zool.*, **27**, 401–410.

Farris,J.S. (1989) Hennig86, version 1.5. *Cladistics*, **5**, 163.

Felsenstein,J. (1989) PHYLIP – Phylogeny Inference Package (Version 3.2). *Cladistics*, **5**, 164–166.

Felsenstein,J. (2004) *Inferring Phylogenies*. Sinauer Associates, Inc., Sunderland, MA.

Fitch,W.M. (1971) Toward defining the course of evolution: Minimum change for a specified tree topology. *Syst. Zool.*, **20**, 406–416.

Goloboff,P.A. (1993) Character optimization and calculation of tree lengths. *Cladistics*, **9**, 433–436.

Goloboff,P.A. (1996) Methods for faster parsimony analysis. *Cladistics*, **12**, 199–220.

Goloboff,P.A. (1999) Analyzing large data sets in reasonable times: Solutions for composite optima. *Cladistics*, **15**, 415–428.

Goloboff,P.A. (2002) Optimization of polytomies: State set and parallel operations. *Mol. Phylogenet. Evol.*, **22**, 269–275.

Goloboff,P.A. *et al.* (2008) TNT, a free program for phylogenetic analysis. *Cladistics*, **24**, 774–786.

Graham,R.L. and Foulds,L.R. (1982) Unlikelihood that minimal phylogenies for a realistic biological study can be constructed in reasonable computational time. *Math. Biosci.*, **60**, 133–142.

Hartigan,J.A. (1973) Minimum mutation fits to a given tree. *Biometrics*, **29**, 53–65.

Hendy,M.D. and Penny,D. (1982) Branch and bound algorithms to determine minimal evolutionary trees. *Math. Biosci.*, **59**, 277–290.

Hendy,M. and Penny,D. (1993) Spectral analysis of phylogenetic data. *J. Classif.*, **10**, 5–24.

Hendy,M.D. *et al.* (1980) Proving phylogenetic trees minimal with l-clustering and set partitioning. *Math. Biosci.*, **51**, 71–88.

Holland,B.R. *et al.* (2005) The minmax squeeze: guaranteeing a minimal tree for population data. *Mol. Biol. Evol.*, **22**, 235–242.

Holzmann,G.J. (1997) The model checker SPIN. *IEEE Trans. Softw. Eng.*, **23**, 279–295.

Kimura,M. (1981) Estimation of evolutionary distances between homologous nucleotide sequences. In *Proc. Natl Acad. Sci. USA*, **78**, 454–458.

Kruskal,J.B.J. (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.*, **7**, 48–50.

Lamport,L. (1975) Multiple byte processing with full-word instructions. *Commun. ACM*, **18**, 471–475.

Moilanen,A. (1999) Searching for most parsimonious trees with simulated evolutionary optimization. *Cladistics*, **15**, 39–50.

Nei,M. and Kumar,S. (2000) *Molecular Evolution and Phylogenetics*. Oxford University Press, Oxford.

Nelder,J.A. and Mead,R. (1965) A simplex method for function minimization. *Computer J.*, **7**, 308–313.

Nixon,K.C. (1999) The parsimony ratchet, a new method for rapid parsimony analysis. *Cladistics*, **15**, 407–414.

Penny,D. and Hendy,M.D. (1987) Turbo tree: a fast algorithm for minimal trees. *Comput. Appl. Biosci.*, **3**, 183–187.

Penny,D. *et al.* (1996) Corrected parsimony, minimum evolution, and hadamard conjugations. *Syst. Biol.*, **45**, 596–606.

Purdom,P.W. *et al.* (2000) Single column discrepancy and dynamic max-mini optimizations for quickly finding the most parsimonious evolutionary trees. *Bioinformatics*, **16**, 140–151.

Ronquist,F. (1998) Fast fitch-parsimony algorithms for large data sets. *Cladistics*, **14**, 387–400.

Roshan,U.W. *et al.* (2004) Rec-I-DCM3: a fast algorithmic technique for reconstructing large phylogenetic trees. In *Proceedings of the IEEE Computational Systems Bioinformatics Conference, Stanford, CA*. IEEE Computer Society, CA, USA.

Schulmeister,S. (2004) Inconsistency of maximum parsimony revisited. *Syst. Biol.*, **53**, 521–521.

Semple,C. and Steel,M. (2003) *Phylogenetics*. Oxford Lecture Series in Mathematics. Oxford University Press, Oxford.

Siegel,S.F. (2007) Model checking nonblocking MPI programs. *Proceedings of Verification, Model Checking, and Abstract Interpretation*, Vol. 4349 of *Lecture Notes in Computer Science*, pp. 44–58.

Sridhar,S. *et al.* (2008) Mixed integer linear programming for maximum-parsimony phylogeny inference. *IEEE-ACM Trans. Comput. Biol. Bioinformatics*, **5**, 323–331.

Steel,M. (2001) Sufficient conditions for two tree reconstruction techniques to succeed on sufficiently long sequences. *SIAM J. Discrete Math.*, **14**, 36–48.

Steel,M. and Penny,D. (2000) Parsimony, likelihood, and the role of models in molecular phylogenetics. *Mol. Biol. Evol.*, **17**, 839–850.

Steel,M.A. and Penny,D. (2005) Maximum parsimony and the phylogenetic information in multistate characters. In Albert,V.A. (ed) *Parsimony, Phylogeny and Genomics*. Oxford University Press, Oxford, pp. 163–178.

Steel,M.A. *et al.* (1993) Parsimony can be consistent. *Syst. Biol.*, **42**, 581–587.

Swofford,D.L. *et al.* (1996) Phylogenetic inference. In Hillis,D.M., Moritz,C. and Mable,B.K. (eds) *Molecular Systematics*, 2nd edn., Sinauer Associates, Sunderland, MA, pp. 407–514.

Yan,M. and Bader,D.A. (2003) Fast character optimization in parsimony phylogeny reconstruction. *Technical report TR-CS-2003-53 from the University of New Mexico*. Available at http://www.cs.unm.edu/research/tech-reports/.