RUTCOR
RESEARCH
REPORT

PEBBL: An Object-Oriented Framework for Scalable Parallel Branch and Bound

Jonathan Eckstein[a] William E. Hart[b]
Cythia A. Phillips[b]

RRR 9-2013, September 2013

RUTCOR
Rutgers Center for
Operations Research
Rutgers University
640 Bartholomew Road
Piscataway, New Jersey
08854-8003
Telephone: 732-445-3804
Telefax: 732-445-5472
Email: rrr@rutcor.rutgers.edu
http://rutcor.rutgers.edu/∼rrr

[a]Department of Management Science and Information Systems and RUTCOR, Rutgers University, 100 Rockafeller Road, Piscataway, NJ 08854, USA
[b]Sandia National Laboratories, Mail Stop 1318, P.O. Box 5800, Albuquerque, NM 87185-1318, USA

Rutcor Research Report
RRR 9-2013, September 2013

PEBBL: An Object-Oriented Framework for Scalable Parallel Branch and Bound

Jonathan Eckstein William E. Hart Cynthia A. Phillips

Abstract. PEBBL is a C++ class library implementing the underlying operations needed to support a wide variety of branch-and-bound algorithms in a message-passing parallel computing environment. Deriving application-specific classes from PEBBL, one may create parallel branch-and-bound applications through a process focused on the unique aspects of the application, while relying on PEBBL for generic aspects of branch and bound, such as managing the active subproblem pool across multiple processors, load balancing, and termination detection. PEBBL is designed to provide highly scalable performance on large numbers of processor cores, using a distributed-memory programming model and MPI message passing.

We describe the basics of PEBBL's architecture and usage, with emphasis on the library's key innovations and contributions, including the notion of branch and bound as a set of operators that manipulate subproblem state transitions. We also describe PEBBL's special features, such as parallel checkpointing, support for specialized ramp-up procedures, and the ability to exhaustively enumerate specified sets of near-optimal solutions.

Finally, we present an example application, the maximum monomial agreement (MMA) problem arising in machine learning applications. For sufficiently difficult problem instances, we show essentially linear speedup on over 6,000 processor cores. We also show how processor cache effects can produce reproducibly superlinear speedups.

Acknowledgements: We thank Noam Goldberg for contributing the code to solve the MMA problem in serial using PEBBL. We also thank John Siirola for his stewardship of PEBBL within the ACRO framework. Jonathan Eckstein's work on PEBBL was supported in part by NSF grant CCR-9902092. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1 Introduction

Branch and bound is a fundamental method of numerical optimization with numerous applications in both discrete optimization and continuous nonconvex global optimization. See for example [11] for a general tutorial on branch-and-bound algorithms. Branch and bound is potentially well-suited to parallel computing, since exploring a branch-and-bound tree often generates a large number of weakly coupled tasks. Despite this suitability, branch-and-bound algorithms are not "embarrassingly parallel" in the sense that parallel implementation is immediate and straightforward. This is particularly the case on architectures lacking shared memory.

It is therefore useful to have parallel branch-and-bound *shells*, *frameworks*, or *libraries*: software tools that provide parallel implementations of the basic, generic functionality common to a broad set of branch-and-bound applications. Those seeking to create new parallel branch-and-bound applications can avoid "reinventing the wheel" by using these tools to handle the generic aspects of search management, while concentrating their programming effort primarily on functions unique to their applications. The need for such tools is greater in parallel than in serial, because managing the pool of active search nodes and selecting the next one to process is relatively straightforward in serial. By contrast, parallel branch-and-bound frameworks require more complex logic for tasks like pool management, distributed termination and load balancing.

This paper describes the PEBBL (Parallel Enumeration and Branch-and-Bound Library) software framework. PEBBL began its existence as the "core" layer of the parallel mixed integer programming (MIP) solver PICO (Parallel Integer and Combinatorial Optimizer). However, increasing use of this core layer for applications with bounding procedures not involving linear programming — for example, the application described in Section 4 below — suggested that PICO be separated into two distinct packages. The core layer, which supports generic parallel branch-and-bound algorithms in an application-independent way, became PEBBL. The remainder of PICO is specific to problems formulated directly as MIPs.

We designed PEBBL with the needs of Sandia National Laboratories (SNL) in mind, with high performance on large-scale, non-shared-memory massively parallel processing (MPP) systems being a key goal. SNL has many current and past MIP-based applications including sensor placement, infrastructure analysis, sensor network management, bioinformatics, scheduling and logistics. Frequently, commercial MIP solvers cannot solve these problems. SNL also has non-MIP-based branch-and-bound applications in areas such as bioinformatics, SAT, and PDE-constrained optimization.

For efficiency and portability, we implemented PEBBL in C++, using the MPI message-passing API [46] to communicate between processors. PEBBL may also be used on shared-memory platforms by configuring standard MPI implementations such as MPICH [28] and Open MPI [27] to emulate message passing through shared memory. Shared-memory systems can often operate efficiently when programmed in this way, since the programming environment naturally tends to limit memory contention.

PEBBL is neither the first nor the only parallel branch-and-bound implementation frame-

work. Early, C-language general parallel branch-and-bound libraries include PUBB [44, 45], BoB [4], and PPBB-Lib [47]. MINTO [39] is an earlier, C-language framework specifically addressing MIP problems. SYMPHONY and BCP, both described in [36, 40], are two related frameworks that support parallelism, respectively written in C and C++. However, they do not support large-scale parallelism, and for applications requiring extensive scalability are being superseded by tools based on CHiPPS/ALPS [41]. Many aspects of CHiPPS/ALPS [41] were influenced by the early development of PICO and PEBBL.

In more recent work, FTH-B&B [5] focuses on fault-tolerance mechanisms for grid environments. For example, the system includes heartbeat communication between parents and children in a multi-level hierarchy to detect faults and checkpointing and recovery mechanisms. Experiments in [5] focus on fault-tolerance performance such as percent of time spent doing fault-tolerance overhead rather than considering scalability and search efficiency.

Some elements of PEBBL's design can in turn be traced back to ABACUS [25, 31, 32], a serial C++ branch-and-bound framework aimed not at generic problems, but at complicated LP-based branch-and-bound methods involving dynamic constraint and variable generation. Some aspects of PEBBL's task distribution and load balancing schemes are based on CMMIP [17, 18, 19, 20], an implementation specific both to mixed integer programming and to the Thinking Machines CM-5 MPP system of the early 1990's.

Sections 2 and 3 of this paper describe PEBBL's design. Section 2 describes PEBBL's *serial layer*, essentially its aspects that are not directly related to parallelism. Section 3 then covers PEBBL's approach to parallelism. The emphasis in Sections 2 and 3 is on PEBBL's innovative contributions and unique features. Innovations introduced in the development of PEBBL include

- Variable-size processor clustering and rendezvous load balancing (although found earlier and in an application-specific and less general and configurable form in CMMIP)
- Variable amounts of subproblem exchange between "hub" and "worker" processors
- Division into serial and parallel layers
- Subproblems that store a *state*, and the notion of describing a branch-and-bound implementation as a collection of operators on these states; this feature allows one to easily change search "protocols" — see Section 2.2
- Use of "threads" (more properly referred to as coroutines) and a nonpreemptive scheduler module to manage tasks on each individual processor.
- Support for application-specific non-tree parallelism during search ramp-up
- A parallel checkpointing feature that can periodically save the entire state of a parallel run
- Support for enumeration of multiple optimal and near-optimal solutions meeting a variety of configurable criteria.

An early description of PICO [24] includes a description of the "PICO core", which evolved into PEBBL. However, this publication is not widely available, and, since its appearance, PEBBL's design has evolved and its scalability has improved significantly. Similar but more recent material is included as part of [3], and a somewhat more recent but very condensed description of the internals of PEBBL and PICO is also embedded in [22]. Neither [3] nor [22] contain computational results.

Here, we present a comprehensive description of PEBBL and describe, with full computational results scaling to thousands of processor cores, its application to a branch-and-bound algorithm that does not use a linear-programming-based bound. Section 4 presents this application, the maximum monomial agreement (MMA) problem described in [16, 21, 26]. For a simple knapsack-solver application of PEBBL, we also show in Section 5 that it is possible for PEBBL to attain reproducibly superlinear speedups due to the cache architecture of modern processors. Section 6 presents some conclusions and discusses the significance of our results.

Finally, both PEBBL source (under the BSD license) and user documentation may be downloaded from `http://software.sandia.gov/acro` — PEBBL is part of ACRO (A Common Repository for Optimizers), a collection of interconnected, mostly optimization-related software projects maintained by Sandia National Laboratories and various collaborators. The PEBBL source available at this URL includes the maximum monomial agreement (MMA) test problem instances and algorithm implementation presented in this paper. At the time of writing, the current PEBBL release is version 1.4.

2 The serial layer

PEBBL consists of two "layers," the *serial layer* and the *parallel layer*. The serial layer provides an object-oriented means of describing branch-and-bound algorithms, with essentially no reference to parallel implementation. The parallel layer contains the core code necessary to create parallel versions of serial applications. Creation of a parallel application can thus proceed in two steps:

1. Create a serial application by defining new classes derived from the serial layer base classes. Development may take place in an environment without parallel processing capabilities or an MPI library.

2. "Parallelize" the application by defining new classes derived from both the serial application and the parallel layer. These classes can inherit most of their methods from their parents — only a few new methods are required, principally to tell PEBBL how to pack application-specific problem and subproblem data into MPI message buffers, and later unpack them. PEBBL uses a data structure called a `PackBuffer` from ACRO's UTILIB utility library to simplify the message packing and unpacking process.

Figure 1 shows the resulting inheritance pattern. Any parallel PEBBL application constructed in this way inherits the full capabilities of the parallel layer, including a highly

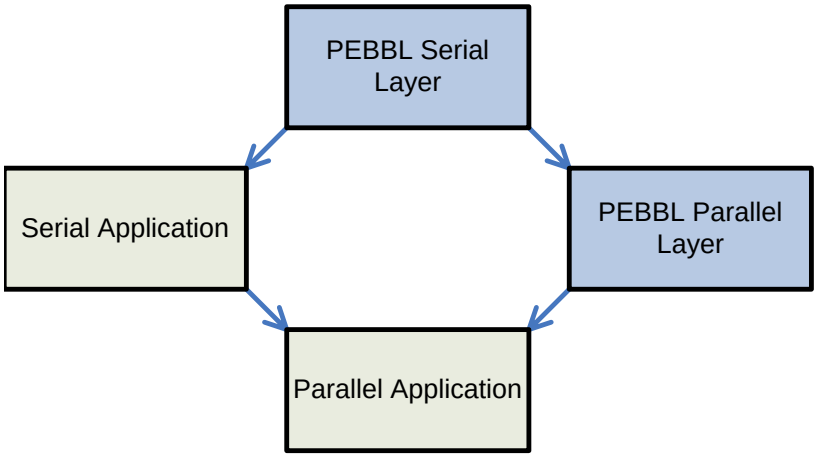

Figure 1: The conceptual relationships of PEBBL's serial layer, the parallel layer, a serial application, and the corresponding parallel application.

configurable spectrum of parallel work distribution and load balancing strategies.

To define a serial branch-and-bound algorithm, a PEBBL user extends two principal classes in the serial layer, `branching` and `branchSub`. The `branching` class stores global information about a problem instance and contains methods that implement various kinds of serial branch-and-bound algorithms, as described below. The `branchSub` class stores data about each subproblem in the branch-and-bound tree, and contains methods that perform generic operations on subproblems. This basic organization is borrowed from ABACUS [25, 31, 32], but it is more general, since there is no assumption that linear programming or cutting planes are involved.

For instance, the simple binary knapsack solver that is included as an example application in the PEBBL distribution derives a class `binaryKnapsack` from `branching`, to store the capacity of the knapsack and the possible items to be placed in it. It also defines a class `binKnapSub`, derived from `branchSub`, which describes nodes of the branch-and-bound tree. Each object instantiating a subproblem class like `binKnapSub` contains a pointer back to the corresponding instance of the global class, in this case `binaryKnapsack`. Through this pointer, each subproblem object can find global information about the problem instance being solved and invoke methods from the global class. Figure 2 illustrates the class relationships.

2.1 Subproblem states and transition methods

Every PEBBL subproblem stores a `state` indicator which progresses through some subset of six states called `boundable`, `beingBounded`, `bounded`, `beingSeparated`, `separated`, and `dead`. Figure 3 illustrates the possible transitions between these states.

The `branchSub` class has three abstract virtual methods, called `boundComputation`, `splitComputation`, and `makeChild`, which are responsible for implementing subproblem

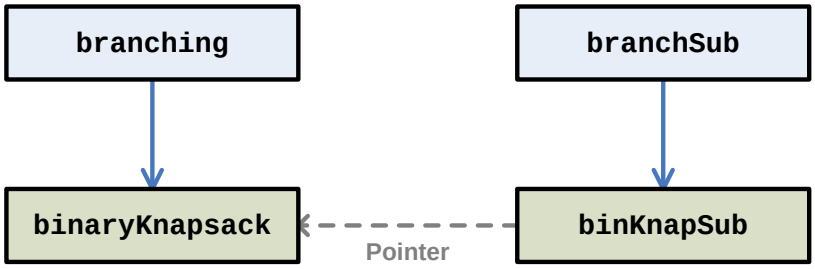

Figure 2: Basic class relationships for a serial PEBBL application: in this case, `binaryKnapsack`, with corresponding subproblem class `binKnapSub`.

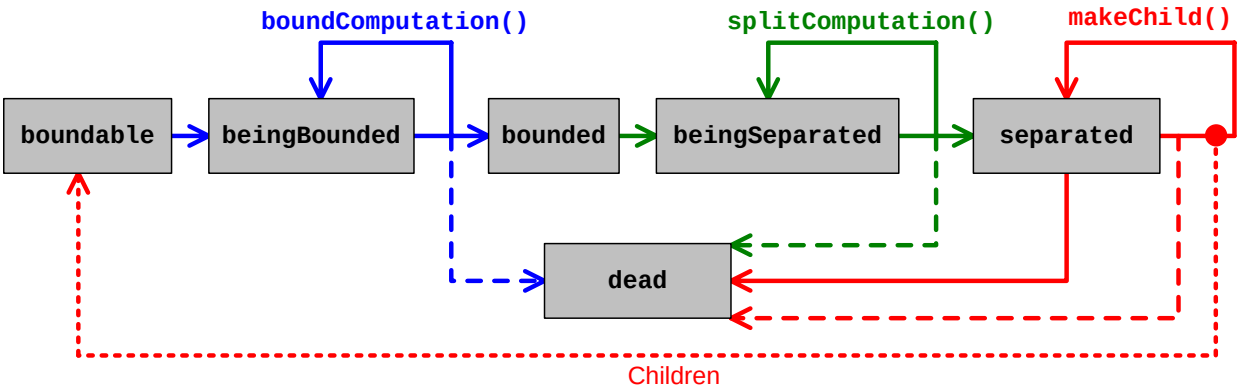

Figure 3: PEBBL subproblem state transition diagram.

state transitions. PEBBL interacts with applications primarily through these methods, and defining a serial PEBBL application consists largely and most importantly of providing definitions for these three methods in the application subproblem class (*e.g.* `binKnapSub`). Each subproblem also has a `bound` data member which may be updated at any time by any of the three state-transition methods, but is most typically set by `boundComputation`.

A subproblem always comes into existence in the `boundable` state, meaning that little or no bounding work has been done for it. A `boundable` subproblem still has an associated bound value, but this value is typically inherited from the parent subproblem. When it starts bounding the subproblem, PEBBL changes its state to `beingBounded` and invokes `boundComputation`. If `boundComputation` sets the subproblem state to `bounded`, that signals that the bounding operation is complete. If it leaves the state as `beingBounded`, the bounding operation is incomplete: to complete the bounding operation, PEBBL must call `boundComputation` at least one more time. Depending on the bound values in the active pool and the search protocol being used (see Section 2.2 below), PEBBL might do so immediately, or might instead turn its attention to a different subproblem and return later. Thus, PEBBL can support applications with highly compute-intensive or incrementally improving bounding methods which one might want to suspend in order to attend to other tasks. This capability could be useful, for example, in implementing branch-and-cut algorithms.

`BoundComputation`, `splitComputation`, and `makeChild` also have the option, at any time, of setting the subproblem state to `dead`. Doing so indicates that PEBBL may immediately remove the subproblem from further consideration — for example, because it has been proven infeasible.

A subproblem that reaches the `bounded` state and is not subsequently fathomed must be separated into child subproblems. When PEBBL elects to separate a subproblem S, it sets S's state to `beingSeparated` and calls S's `splitComputation` method. Each invocation of `splitComputation` has two principal options. First, it can leave S's state as `beingSeparated`, indicating that the operation is not complete and `splitComputation` must be called at least one more time to complete the separation process. Second, it can set S's state to `separated`, indicating that separation process is complete. In this case it should set S's data member `numChildren` to the number of child subproblems produced. Thus, PEBBL can also support applications with variable branching factors, or in which the separation process is a long, potentially interruptible computatation. `SplitComputation` also has the option of setting the subproblem state to `dead` in case it encounters some form of infeasibility or has otherwise determined that the subproblem may be fathomed.

Once a subproblem S reaches the `separated` state, PEBBL calls its `makeChild` method up to S.`numChildren` times, in order to create S's children. Each call to `makeChild` should return a pointer to a single new child subproblem. Once all of S's children have been created, PEBBL automatically sets its state to `dead`.

2.2 Pools, handlers, and the search framework

PEBBL's serial layer orchestrates serial branch-and-bound search through a module called the "search framework," which acts as an attachment point for two user-specifiable objects, the *pool* and the *handler*. The combination of pool and handler determines the exact kind of branch-and-bound algorithm to be executed. Essentially, the framework executes a loop in which it extracts a subproblem S from the pool and passes it to the handler, which may in some cases create children of S and insert them into the pool. If S is not `dead` after processing by the handler, the framework returns it to the pool. Figure 4 illustrates the relationship between the search framework, pool object, and handler object.

The pool object dictates how the currently active subproblems are stored and accessed, which effectively determines the branch-and-bound search order. Currently, there are essentially three kinds of pool: a heap sorted by subproblem bound, a stack, and a FIFO queue. If the user specifies the heap pool, then PEBBL will follow a best-first search order. The stack pool induces a depth-first order, and the queue pool induces a breadth-first order. For particular applications, however, users may implement additional kinds of pools, thus specifying other search orders. The heap pool also has a "diving" option to give priority to an application-defined "integrality measure" until an incumbent solution is found, and then revert to standard best-bound ordering.

Critically, at any instant in time, the subproblems in the pool may in principle represent any mix of states. For example, some might be `boundable`, and others `separated`. This

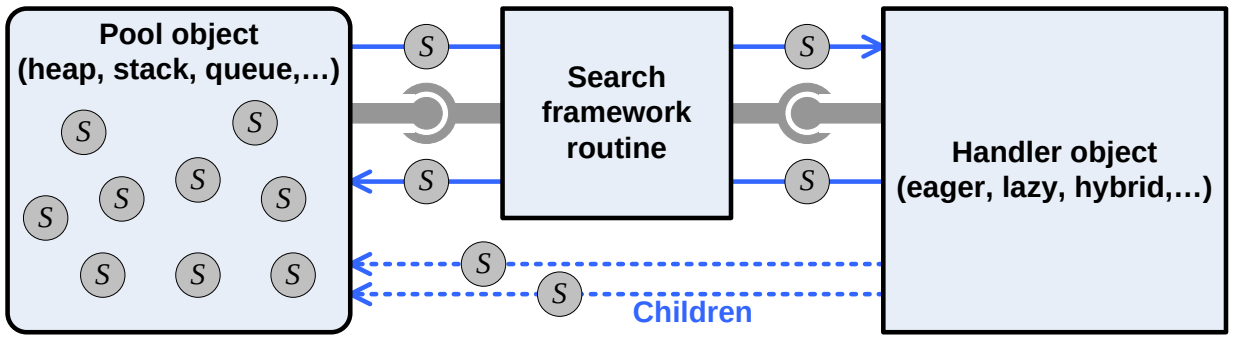

Figure 4: The search framework, pool, and handler. Each S indicates a subproblem.

feature gives the user flexibility in specifying the search *protocol*, a term we use to denote a concept distinct from the search order. The handler object's responsibility is to implement a search bounding protocol.

To illustrate the search protocol concept, consider the classical branch-and-bound method for MIP, as typically formulated by operations researchers: one removes a subproblem from the currently active pool and computes its linear programming relaxation bound. If the bound is strong enough to fathom the subproblem, the subproblem is discarded. Otherwise, one selects a branching variable, creates two child subproblems by restricting the range of this variable, and inserts these children into the pool. This type of procedure is called *lazy* bounding [12]. In the PEBBL framework, lazy bounding is implemented by a handler called `lazyHandler`, which inserts subproblems into the pool upon creation, in the `boundable` state. After removing a subproblem from the pool, it advances it all the way to the `dead` state, unless either `boundComputation` or `splitComputation` needs to be applied more than once.

An alternative approach, more common in work originating from the computer science community, is usually called *eager* bounding [12]. Here, all subproblems in the pool have already been bounded. One picks a subproblem out of the pool, immediately separates it, and then creates and bounds each of its children. Children whose bounds do not cause them to be fathomed are returned to the pool. Such an approach makes the most sense when the bounding procedure is relatively quick: it reduces the pool insertion and deletion overhead per bounding operation at the expense of a larger task granularity. This larger granularity somewhat reduces the potential for parallelism, but also reduces potential task-scheduling overhead. In PEBBL, eager search is implemented by a handler called `eagerHandler` that attempts one call of `boundComputation` for each subproblem before inserting it into the pool.

PEBBL also contains a third standard handler, called `hybridHandler`, which implements a strategy that is somewhere between eager and lazy bounding, and is perhaps the most simple and natural given PEBBL's concept of subproblem states. Upon removing a subproblem from the pool, `hybridHandler` simply inspects its state and performs a single application of whichever of the methods `boundComputation`, `splitComputation`, or `makeChild` is appro-

priate. In the `makeChild` case, the handler stores each resulting child in the pool.

2.3 Solution management and representation

The `branchSub` class contains two more abstract methods that the application subproblem class must define: `candidateSolution` and `extractSolution`. Once a subproblem S enters the `bounded` state, PEBBL invokes S.`candidateSolution`. A return value of `true` indicates that the bounding operation produced a feasible solution. In this case, PEBBL calls S.`extractSolution`, which should return an object with type derived from the PEBBL base class `solution`. An application may either derive its own representation class from `solution`, or use a standard PEBBL template class that encapsulates one-dimensional arrays of standard numeric types.

Optionally, the application subproblem class may also define a method called `incumbentHeuristic` that attempts to derive a feasible solution from a `bounded` subproblem. For example, `incumbentHeuristic` might attempt to round fractional variables in the LP solution for an application whose bound is based on solving a linear programming relaxation of a combinatorial problem. In the serial layer, PEBBL calls `incumbentHeuristic` for every `bounded` subproblem for which `candidateSolution` returns `false`. `IncumbentHeuristic` has the option of simply returning immediately if the application does not wish to run its heuristic at every search node.

PEBBL sends each solution s obtained by `extractSolution` or `incumbentHeuristic` to the method `branching::foundSolution`. In the serial layer, if enumeration of multiple solutions is not active, this method simply replaces the current incumbent with s if s has a better objective value. Note that application code also has the option of calling `foundSolution` directly if it generates a new solution at a time not ordinarily anticipated by PEBBL's handlers.

2.4 Solution enumeration

The extent of PEBBL's search process is controlled by two nonnegative parameters, `relTolerance` and `absTolerance`. Normally, if a subproblem's bound is within either an absolute distance `absTolerance` or relative distance `relTolerance` of the current incumbent, PEBBL fathoms and deletes it. Thus the final incumbent is guaranteed to be either within `absTolerance` objective function units or a multiplicative factor `relTolerance` of the true optimum. The exception to this process is when PEBBL is configured to enumerate multiple solutions. Such enumeration is controlled by four further parameters: `enumRelTol`, `enumAbsTol`, `enumCutoff`, `enumCount`. When set, these parameters have the following meaning:

`enumAbsTol` $= a$**:** retain all solutions no more than a units worse than the optimum.

`enumRelTol` $= r$**:** retain all solutions no more than a multiplicative factor r worse than the optimum.

`enumCutoff` $= c$: retain all solutions whose objective value is better than c.

`enumCount` $= n$: retain the best n solutions. If the n^{th}-best solution is not uniquely determined, PEBBL breaks objective-value ties arbitrarily.

If more than one of these parameters is set, then PEBBL retains only solutions jointly meeting all the specified criteria. If any of these parameters are set, then enumeration is considered active, and PEBBL stores not only the single incumbent solution of classical branch and bound, but also a *repository* of multiple solutions. PEBBL adds solutions to the repository as they are found, and removes them whenever it deduces they cannot meet the enumeration criteria.

In serial, the effect of the enmeration parameters is to alter the criteria PEBBL uses to fathom subproblems and the behavior of the `foundSolution` method. When using just the `enumAbsTol` or `enumRelTol` parameters, fathoming only occurs when a subproblems bound is worse than the incumbent by at least the amount specified by one of the active criteria. For `enumCount`, subproblems are compared not to the incumbent, but to the `enumCount`$^{\text{th}}$-best solution in the repository. When enumeration is active, `foundSolution` enters any solution meeting the enumeration criteria into the repository. If `enumCount` is set and `enumCount` solutions have already accumulated, this operation may also result in `foundSolution` deleting the worst element in the repository.

For enumeration to work properly, the application classes need to have two capabilities: "branch-through" and solution duplicate detection. *Branch-through* means separating search nodes that would never be split if only one solution were desired. Let us call a search node S *tight* if S.`candidateSolution` returns `true` and S.`extractSolution` returns a solution whose objective value is equal to the bound of S. In integer programming, for example, a node whose LP relaxation solution contains only integer values is tight. If enumeration is not active, tight nodes are always fathomed immediately after their `extractSolution` method is called, and so the `splitComputation` method may assume that it will never be applied to tight nodes. With enumeration active, however, PEBBL may attempt to split tight nodes. In some applications, the procedure for splitting tight nodes may have to be completely different from the normal separation procedure. In integer programming, for example, one normally branches by fixing fractional variables, but for a tight node there will be no such variables.

The second capability normally needed to support enumeration is duplicate solution detection. To prevent duplicate solutions from accumulating in the repository, PEBBL uses two features of the `solution` class: a hash value calculation and the `duplicateOf` method. `DuplicateOf` should compare two solution objects and return `true` if they are identical. By storing the repository as a hash table, PEBBL uses the hash value to keep the number of `duplicateOf` invocations manageable. In addition to the hash table, the repository is also stored as a heap organized by reverse objective function value. The heap is helpful in implementing the `enumCount` criterion and writing the final repository state in sorted order.

If enumeration is active and one is using an application-specific solution representation class, then that class must correctly implement `duplicateOf`. For efficiency over large repositories, one also needs a functional hash value calculation. PEBBL provides default hash

value and `duplicateOf` implementations that require only that the class correctly support two methods, `sequenceLength` and `sequenceData`. These methods should convert the solution to a *sequence representation*: a unique finite sequence of `sequenceData` values of type `double`. Thus, full enumeration functionality is automatically obtained for any application supporting branch-through whose solution representation class correctly implements the methods `sequenceLength` and `sequenceData`.

There is one situation in which only branch-through is required to support enumeration, and duplicate detection is not necessary: if `candidateSolution` returns `true` only for tight nodes, and the application lacks an incumbent heuristic. Then, presuming that the solution returned by S.`extractSolution` is contained in the subset of the solutions corresponding to the search node S, it is easily proven that duplicate solutions will never be offered to the repository, and in principle one could simple define `duplicateOf` to return `false`.

At present, PEBBL only filters out solutions that are *exact* duplicates as defined by the `duplicateOf` method. PEBBL currently lacks any gradated notion of solution distance or "diversity", and has only a boolean sense of "exactly the same" or "not the same". For applications that can generate large numbers of symmetric or nearly identical solutions, it may be desirable to monitor and control the diversity of the repository in such a non-boolean sense. For an example of this kind of technique for MIP problems, see [13]. Unlike PEBBL's enumeration scheme, however, the technique described in [13] does not guarantee full enumeration of specific sets of solutions.

3 The parallel layer

The parallel layer's principal capabilities are implemented by the classes `parallelBranching` and `parallelBranchSub`, which have similar roles to `branching` and `branchSub`, from which they are respectively derived.

To create a parallel PEBBL application, one must define two new classes. The first is derived from `parallelBranching` and the serial application `branching` class. In the knapsack example application, we thus define a new class called `parallelBinaryKnapsack` which has both `parallelBranching` and `binaryKnapsack` as `virtual` base classes. A similar pattern is repeated for the subproblem classes: in the knapsack example, we define `parBinKnapSub` with `virtual` base classes `binKnapSub` and `parallelBranchSub`. Figure 5 depicts the entire inheritance structure for the example parallel knapsack application. After such an inheritance structure has been established, the parallel version of the application will acquire full parallel PEBBL functionality once one provides implementations of a few abstract methods in the parallel layer. These methods primarily concern packing problem instance and subproblem descriptions into MPI buffers, and unpacking the corresponding received messages.

PEBBL's parallel layer organizes processors similarly to the later versions of CMMIP [17, 19]. Processors are organized into *clusters*, each consisting of one *hub* processor and one or more *worker* processors. The number of processors in each cluster is determined by the parameter `clusterSize`, except that the last cluster may be smaller if the total number

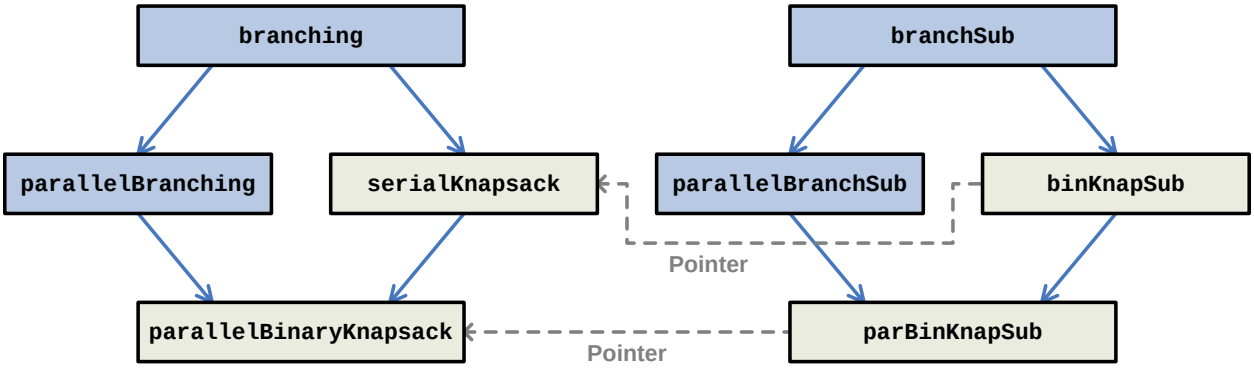

Figure 5: Inheritance and pointer structure of the parallel knapsack application.

of processors is not a multiple of `clusterSize`. In clusters with fewer processors than the parameter `hubsDontWorkSize`, the hub processor simultaneously functions as a worker; otherwise it does not. It is possible for a cluster, or even all clusters, to consist of only a single processor.

3.1 Tokens and work distribution within a cluster

The workers in a PEBBL cluster are generally not pure "slaves." Each worker can maintain its own pool of active subproblems. Depending on how various run-time parameters are set, however, the pool might be small, in the extreme case never holding more than one subproblem. Each worker processes its pool in the same general manner as the serial layer: it picks subproblems out of the pool and passes them to a search handler until the pool is empty. The parallel layer uses the same search handler objects as the serial layer, but in parallel they also have the ability to *release* subproblems from the worker. Released subproblems do not return to the local pool. Instead, the worker cedes control over these subproblems to the hub. Eventually, the hub may send control of the subproblem back to the worker, or to another worker.

For simplicity throughout the remainder of this subsection, we describe the work distribution scheme as if a single cluster spans all the available processors. In the next subsection, we will amend the description for the case of more than one cluster.

In parallel settings, `eagerHandler` decides whether to release a subproblem as soon as it has become `bounded`, whereas `lazyHandler` and `hybridHandler` make the release decision when they create a subproblem. The decision is a random one, with the probability of release controlled by run-time parameters. If PEBBL is configured so that the release probability is 100%, then every subproblem is released, and control of a subproblem is always returned to the hub at a certain point in its lifetime. In this case, the hub and its workers function like a classic "master-slave" system. When the probability is lower, the hub and its workers are less tightly coupled. The release probability is in general not constant: on a worker

w, it typically varies between parameter-determined bounds depending on the fraction of the estimated total work w controls. To promote even work distribution early in the run, the release probability is also set to 100% for the first `startScatter` (another run-time parameter) subproblems encountered at each worker.

When a subproblem is released, only a small portion of its data, called a *token* [18, 42], is actually sent to the hub. The subproblem itself may then move to a secondary pool, called the *server pool*, that resides on the worker. A token consists of only the information needed to identify a subproblem, locate it in the server pool, and schedule it for execution. On a 64-bit processor, a token requires 80 bytes of storage, much less than the full data for a subproblem in most applications. Since the hub receives only tokens from its workers, as opposed to entire subproblems, these space savings translate into reduced storage requirements and communication load at the hub. Depending on which handler is being used, PEBBL can gain further efficiencies by configuring tokens to be references either to a subproblem, to an as-yet-uncreated child of a subproblem, or to several such children.

The hub processor maintains a pool of subproblem tokens that it has received from workers. Each time it learns of a change in workload status from one of its workers, the hub reevaluates the work distribution in the cluster. Essentially, the hub maintains information about its workers in a heap, with the most "deserving" worker in the root position. The notion of "deserving" takes into account both subproblem quantity and quality (as measured by the distance of bound to the incumbent), although run-time parameters can optionally disable the quality criterion.

If the worker w^* at the top of the heap is sufficiently "deserving" and the hub pool is nonempty, the hub picks the best subproblem token t from its pool and assigns it to w^*. However, the *dispatch* message assigning the subproblem may not go directly to w^*. Instead, it goes to the worker w that originally released t. If $w \neq w^*$, then when w receives the token, it forwards the necessary subproblem information to w^*, much as in [17, 18, 19, 42]. The hubs make subproblem dispatch decisions one at a time. To save system overhead, however, when a single activation of the hub logic results in multiple dispatch messages to be sent from the hub to the same worker, the hub attempts to pack them into a single MPI message, subject to an overall buffer length limit.

In addition to sending subproblems, the hub periodically broadcasts overall workload information to its workers, so the workers know the approximate relation of their own workloads to other workers'. This information allows each worker to adjust its probability of releasing subproblems.

Depending on the settings of the parameters controlling subproblem release and dispatch, PEBBL's intracluster work distribution system can range from a classic "master-slave" configuration to one in which the workers "own" the vast majority of the subproblems, and the hub controls only a small "working set" it tries to use to correct imbalances as they occur. A spectrum of settings between these two extremes are also possible. For example, there is a configuration in which the hub controls the majority of work within the cluster, but each worker has a small "buffer" of subproblems to prevent idleness while waiting for the hub to make work-scheduling decisions. The best configuration along this spectrum depends

on both the application and the relative communication/computation speed of the system hardware. In practice, some run-time experimentation may be necessary to find the best settings for a given combination of computing enviroment and problem class.

One difficulty with the work distribution scheme occurs when the workers have workloads that are seriously out of balance, yet the hub's token pool is empty. In our practical experience, such situations occur rarely, and usually near the end of a run, unless the subproblem release and dispatch parameters are set in a pathological way. In any event, there is a secondary mechanism, which we call *rebalancing*, by which workers can send blocks of subproblem tokens to the hub outside of the usual handler-release cycle. If a worker detects that it has a number of subproblems exceeding a user-specifiable factor `workerMaxLoadFactor` times the average worker load, it selects a group of subproblems in its local pool and releases them to the hub. The hub can then redistribute these subproblems to other workers.

The subproblem release probabilities and rebalancing operations at the workers, along with the calculation of when workers are "deserving" in hub dispatch process are controlled and calibrated to keep the fraction of subproblems in the cluster controlled by the hub close to the parameter `hubLoadFac`, and the remaining subproblems relatively evenly distributed among the workers. In this calculation, an adaptively computed "discount" is applied to a worker process colocated on the hub processor. Specifically, if the hub processor appears to be spending a fraction h of its time on its hub functions but is also a worker, then the target number of subproblems for its worker process is a factor $1 - h$ less than for the other worker processors.

3.2 Work distribution between clusters

For any given combination of computing environment, application, and problem instance, there will be a limit to the number of processors that can operate efficiently as a single cluster. Depending on the application and the hardware, the hub may simply not be able to keep up with all the messages from its workers, or it may develop excessively long queues of incoming messages. Having a pure hub that is not also a worker and adjusting the parameters described in the previous section to reduce communication between the workers and the hub will help reduce the communication load, but reduced communication frequency can also impair the hub's ability to distribute work and correct load imbalances. At a certain point, adding more processors to a single cluster will not improve performance appreciably no matter how one manipulates the relevant parameters. To take advantage of even more processors, PEBBL provides the ability to divide the overall processor set into multiple clusters. The size of each cluster is determined by the run-time parameter `clusterSize`, as mentioned above.

PEBBL's method for distributing work between clusters resembles CMMIP's [17, 19], with some additional generality: there are two mechanisms for transferring work between clusters, *scattering* and *load balancing*. Scattering comes into play when subproblems are released by the workers. If there are multiple clusters, and a worker has decided to release a subproblem, the worker makes a secondary random decision, under control of some additional parameters, as to whether the subproblems should be released to the worker's own hub or

to the hub of a cluster chosen at random. When choosing the cluster to scatter to, the probability of picking any particular cluster is proportional to the number of workers it contains, with the same adaptively determined "discount" mentioned in the previous section applied to workers that are also functioning as hubs. The worker's own cluster is not excluded from consideration. When subproblems are forced to be released through the `startScatter` mechanism at the beginning of a run, they are always scattered to a random cluster.

To supplement scattering, PEBBL also uses a form of *rendezvous* load balancing similar to CMMIP [17, 19]. Mahanti and Daniel [37] and Karypis and Kumar [33] describe earlier, synchronous applications of the same basic idea, but to individual processors instead of clusters. This procedure also has the important side effect of gathering and distributing global information on the amount of work in the system, which in turn facilitates control of the scattering process, and is also critical to termination detection in the multi-hub case.

Critical to the operation of the load balancing mechanism is the concept of the *workload* at a cluster c at time t, which we define as

$$L(c,t) \quad = \quad \sum_{P \in C(c,t)} |\overline{z}(c,t) - z(P,c,t)|^{\pi}.$$

Here, $C(c,t)$ denotes the set of subproblems that c's hub knows are controlled by the cluster at time t, $\overline{z}(c,t)$ represents the fathoming value known to cluster c's hub at time t, and $z(P,c,t)$ is the best bound on the objective value of subproblem P known to cluster c's hub at time t. The fathoming value is the objective value that allows a subproblem to be fathomed, which is simply the incumbent when enumeration is not active, but can be different if enumeration is in use. The exponent π is either 0, 1, or 2, at the discretion of the user. If $\pi = 0$, only the number of subproblems in the cluster matters. Values of $\pi = 1$ or $\pi = 2$ give progressively higher "weight" to subproblems farther from the incumbent. The default value of π is 1. Essentially the same metric is used to measure the workload balance within clusters.

The rendezvous load balancing mechanism organizes all the hub processors into a balanced tree whose branching factor is controlled by a run-time parameter that defaults to 2. Periodically, messages "sweep" semi-synchronously through this entire tree, from the leaves to the root, and then back down to the leaves. These messages repeat a pattern of a *survey sweep* followed by a *balance sweep*. The frequency of these sweeps is controlled by a timer, with the minimum spacing between survey sweeps being set by a run-time parameter. If the total workload on the system appears to be zero, then this minimum spacing is not observed and sweeps are performed as rapidly as possible, to facilitate rapid termination detection. Under certain conditions, including at least once at the end of every run, a *termination check* sweep is substituted for the balance sweep; see Section 3.10 for a discussion of the termination check procedure.

The survey sweep gathers and distributes system-wide workload information. This sweep provides all hubs with an overall system workload estimate, essentially the sum of the $L(c,t)$ over all clusters c. However, if the sweep detects that these values were based on inconsistent incumbent values, it immediately repeats itself, a situation we call a *survey restart*.

At the end of a successful survey sweep, each hub determines whether its cluster should be a potential *donor* of work, a potential *receiver* of work, or (typically) neither. Donors are clusters whose workload exceeds the average by a factor of at least `loadBalDonorFac`, while receivers must be below the average by at least `loadBalReceiverFac`. Next, the balance sweep begins. A form of parallel prefix operation [6], this single up-and-down message sweep of the tree counts the total number of donors d and receivers r, assigns each donor a unique number in the range $0, \ldots, d-1$, and assigns each receiver a unique number in the range $0, \ldots, r-1$. The first $y = \min\{d, r\}$ donors and receivers then "pair up" via a rendezvous procedure involving $3y$ point-to-point messages. Specifically, donor i and receiver i each send a message to the hub for cluster i, for $i = 0, \ldots, y-1$. Hub i then sends a message to donor i, telling it the processor number and load information for receiver i. See [29, Section 6.3] or [17, 19] for a more detailed description of this process. Within each pair, the donor sends a single message containing subproblem tokens to the receiver. Thus, the sweep messages are followed by up to $4y$ additional point-to-point messages, with at most 6 messages being sent or received by any single processor — this worst case occurs when a hub is both a donor and a rendezvous point. Both the survey and balancing sweeps involve at most $2(b+1)$ messages being sent or received at any given hub processor, where b is the branching factor of the load-balancing tree. Thus, the total number of messages per processor per round of load balancing is bounded above by the constant $2(b+1) + 2(b+1) + 6 = 4b + 10$. This kind of constant upper bound on the number of messages per processor required to perform a global operation is instrumental in designing scalable parallel algorithms.

Interprocessor load balancing mechanisms are sometimes classified as having the form of either "work stealing," that is, initiated by the receiver, or "work sharing," that is, initiated by the donor. The rendezvous method is neither. Instead, donors and receivers efficiently locate one another on an equal, peer-to-peer basis, possibly across a large collection of processors.

3.3 Thread and scheduler architecture

As described above, the parallel layer requires each processor to perform a certain degree of multitasking. To manage such multitasking in as portable a manner as possible, PEBBL uses its own system of nonpreemptive "threads", more precisely described as coroutines. These threads are called by a scheduler module, and are not interruptible at the application level. They simply return to the scheduler when they wish to relinquish control. Within each processor, the threads share a common memory space through a pointer to the instance of `parallelBranching` being solved.

Depending on how it is configured, PEBBL may create the following threads:

- *Compute threads* that handle the fundamental computations:
  - A *worker thread*, for processing subproblems
  - Optionally, an *incumbent search thread* to be dedicated to running incumbent heuristics

- Threads related to managing the incumbent:
  - An *incumbent broadcast thread* for distributing information about new incumbents
  - An optional *early output thread*, to output provisional solution information before the end of long runs that may be aborted.
- Threads associated with work distribution and load balancing:
  - A *hub thread* that handles the main hub functions
  - A *subproblem server thread* responsible for forwarding subproblem information from one worker to another
  - A *subproblem receiver thread* that receives subproblem information at workers
  - A *worker auxiliary thread*, which workers use to receive global workload information and miscellaneous commands from their hubs
- Threads related to enumerating multiple solutions:
  - A *repository receiver thread* which receives hashed solutions from other processors
  - A *repository merger thread* which manages `enumCount` enumeration in a parallel setting.

Furthermore, applications derived from PEBBL have the ability to incorporate their own additional, application-specific threads into PEBBL's multitasking framework. For mixed integer programming, for example, PICO creates an additional thread to manage communication of "pseudocost" branching quality information between processors. Figure 6 depicts PEBBL's thread structure. Note that some threads (those with the dashed outlines) are optional and their presence depends on which PEBBL features are active. Some threads are present on some kinds of processors (for example, hubs) and not on others.

3.4 The scheduling algorithm

At any given time, the scheduler considers each thread to be in one of three states — *ready*, *waiting*, or *blocked* — and only runs threads that are in the ready state. Waiting threads wait for the arrival of a particular kind of message, as identified by an MPI tag. The scheduler periodically tests for message arrivals, changing corresponding thread states from waiting back to ready as necessary. Threads in the blocked state are waiting for some event other than a message arrival. The scheduler periodically polls these threads by calling their `ready` virtual methods; once a blocked thread's `ready` method returns `true`, the scheduler reverts it to the ready state.

Threads are organized into *groups*, each group having a priority. Group priorities are absolute, in the sense that the scheduler only runs threads from the highest priority group that contains any ready threads.

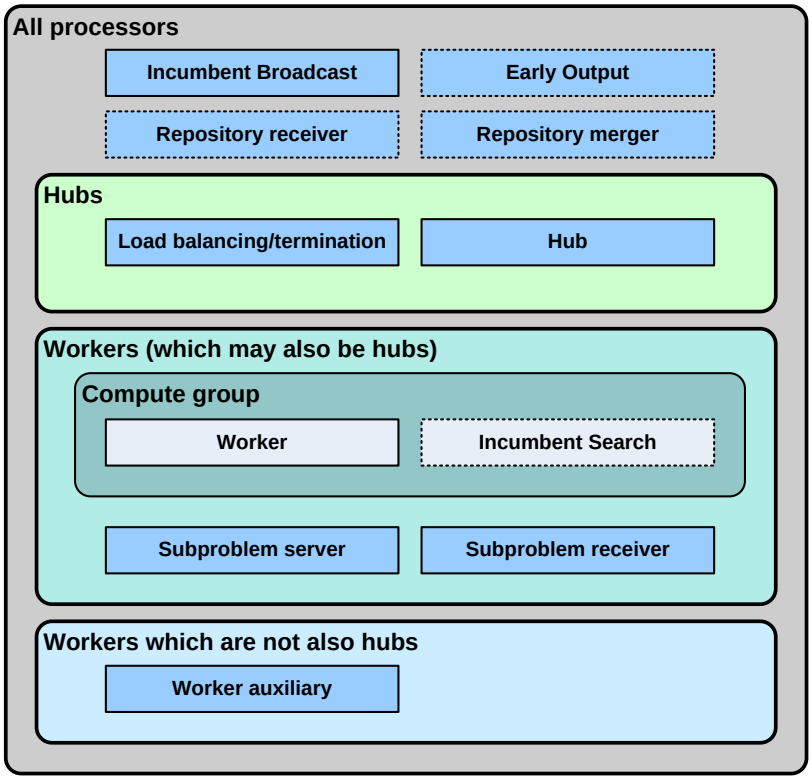

Figure 6: PEBBL's threads. Applications may add threads of their own.

Each group may use one of two scheduling disciplines. The first is a simple round-robin scheme, in which ready threads are selected in a repeating cyclic order. The second possibility is a variant of *stride scheduling* [48, 35]. This scheme allows the user to specify the approximate fraction of CPU resources that should be allocated to each thread; see [23, 24] for details.

PEBBL configures this generic scheduler model to use two thread groups: a round-robin-scheduled high-priority group primarily for *message-triggered* threads, and a stride-scheduled low-priority group for *compute* threads. A message-triggered thread typically spends most of its time in the waiting state, and activates only when at least one message with the thread's specified message tag arrives. Once activated, the thread processes all its pending messages — which may in turn involve sending some messages — and then returns to the waiting state. Since message-triggered threads are in the high-priority group, they tend to run soon after their messages arrive.

Compute threads are usually in the ready state, but may be in the blocked state if they have exhausted all their available work. These threads are stride-scheduled, so long as no message-triggered threads need to run. By default, all of PEBBL's compute threads contain logic to actively manage their "granularity", the average time u they consume before returning to the scheduler. Compute threads try to manage the amount of work they do at each invocation so that the average value of u is approximately equal to the run-time

parameter `timeSlice`. The best value of `timeSlice` depends on the hardware, the MPI implementation, and the application. A small value means that message-triggered threads will run soon after their messages arrive, giving fast communication response, whereas a large value will reduce the overhead expended on the scheduler and entering and exiting compute threads. Ideally, one attempts to balance these two goals; in preliminary testing, we have had good results with a value approximately 20 times the time the scheduler needs to check for arriving messages.

3.5 Compute threads: worker and incumbent heuristic

The worker and optional incumbent heuristic thread comprise the compute group, which is present only on worker processors, and only worker processors. The worker thread simply extracts subproblems from the worker's local pool and passes them to the search handler. If the local pool is empty, the worker thread enters the blocked state.

The incumbent heuristic thread is optional, its existence controlled by virtual methods and run-time parameters. When it exists, it provides a way for PEBBL to control the amount of computational effort expended in running heuristics intended to produce improved incumbent solutions. In both the serial and parallel layers, a PEBBL application's subproblem-processing methods, primarily `boundComputation` and `splitComputation`, are free to run incumbent heuristics and call the `foundSolution` method at any time. In the parallel layer, the incumbent heuristic thread provides a second mechanism for running heuristics.

Since the compute group is stride-scheduled, the scheduler attempts to maintain a specified ratio between the CPU effort expended on the worker and incumbent search threads. The parameters controlling this process are configured so the effort ratio can be sensitive to the overall gap between the incumbent and the best-known active subproblem. When this gap is large, more time is devoted to the incumbent heuristic, in the hope of finding a better solution. There is also a parameter-specified gap threshold below which the incumbent thread is disabled completely. Sometimes, most often early in the search process, the gap may be large but the worker thread on a particular worker may be blocked for lack of subproblems in its local pool. In this case, the incumbent search thread will usefully absorb available CPU resources on the worker until subproblems arrive for the worker thread. PEBBL also provides a mechanism for the worker thread to send partial solution information to the incumbent heuristic thread.

3.6 Single incumbent management threads: incumbent broadcast and early output

The incumbent broadcast thread is a message-triggered thread that runs on all processors, and listens for messages carrying new incumbent values. Each processor stores both the best objective value z^* it currently knows for the incumbent, and the rank p^* of the processor that found that value. The full incumbent solution itself is stored only on the processor that generated it. PEBBL's incumbent broadcasting scheme is similar to CMMIP's: the

parallel layer overrides the serial-layer `foundSolution` method so that whenever a processor p detects what it believes to be an improved incumbent, it commences a broadcast along a balanced tree spanning all processors and rooted at p. The tree's branching factor controlled by a run-time parameter and defaults to 32. The broadcast messages contain not only the proposed new incumbent value z, but also p, the rank of the originating processor.

The incumbent broadcast thread is a message-triggered thread activated by these messages. When it receives such a message, containing a pair (z, p), it lexicographically compares (z, p) to the local processor's stored value of (z^*, p^*). If the objective value z is inferior to z^*, or $z = z^*$ but $p^* \geq p$, then the thread essentially ignores the message. Otherwise, if z^* is preferable to z or $z = z^*$ but $p^* < p$, it performs the update $(z^*, p^*) \leftarrow (z, p)$ and forwards the (z, p) message along the tree. This procedure guarantees that all processors will eventually agree on both z^* and p^*. Two processors may simultaneously initiate broadcasts of (z, p) pairs, but only the lexicographically preferred one will reach all processors; the other broadcast tree will "wither" when it encounters nodes first reached by the lexicographically preferred one.

The *early output thread* is another thread related to incumbent management. Normally, PEBBL outputs the contents of the incumbent solution only at the end of a run. However, by setting some run-time parameters, one can configure PEBBL to continually output new incumbent solutions. Essentially, if processor 0, which serves as the hub for the first cluster, detects a new incumbent solution that has not been superseded for some specified time period, it sends a message to its source processor p^*, which is received by the early output thread. Processor p^* then sends the solution back to processor 0 (this message is again received by the early output thread), which writes it to the solution output file. This feature is useful for potentially long runs which may not be assured of executing to completion, at the cost of a modest increase in I/O and interprocessor communication overhead.

3.7 Work distribution and load balancing threads

The *hub thread* is a message-triggered thread that runs on hub processors, and listens for messages from workers to hubs, containing workload status information, tokens for subproblems that are being scattered or rebalanced, and acknowledgements of receipt of subproblems dispatched from the hub. When it awakens, the hub thread processes the contents of all such messages received since its last invocation, making the requisite changes in hub data structures, and then activates the hub work distribution logic. The hub work distribution logic can also be activated by other threads that may be coresident on a hub processor, using a method called `activateHub`. The functions peformed by hub activation were already described in Section 3.1.

When a hub processor h assigns the subproblem represented by a token t to a worker w^*, it sends a message with MPI containing t and w^* to the worker processor w on which the subproblem is stored. The *subproblem server thread*, which exists on each worker, listens for such messages. It then sends a message containing the subproblem to w^*. The *subproblem receiver thread* is responsible for receiving such messages, unpacking their contents, and

placing the corresponding subproblem in w^*'s worker pool. After the subproblem is delivered, processor w^* sends an acknowledgement to h; these acknowledgements are usually blocked together in groups, along with released subproblem tokens and current workload information at w^*, and sent periodically to h in messages received by the hub thread. Multiple messages for the subproblem server or receiver threads are also blocked together to reduce message overhead.

A number of (non-exclusive) special cases may occur in the above scenario. The first is $w = h$, in which case the logic of the subproblem receiver thread is invoked within the hub logic, and no message is actually sent to the subproblem server thread. The second is $w = w^*$, in which case "delivery" of the subproblam may be executed locally on processor w, with no message sent from a suproblem server thread to a subproblem receiver thread. Finally, when $h = w^*$, no message is needed to acknowledge receipt of the subproblem.

The *load balancing/termination thread* orchestrates the load balancing scheme described in Subsection 3.2. It runs on every hub processor, and contains finite-state-machine logic to move all hub processors approximately synchronously through the various message sweeps and other operations described in Subsection 3.2. It is basically a message-triggered thread that listens for various kinds of messages, depending on what phase of the load balancing cycle is currently in progress. One exception is that it may enter the blocked state, based on a timer, in order to enforce the minimum time between load balancing rounds. The load balancer thread is also is responsible for managing search termination detection and checkpointing, as described in Section 3.10 below. If there is only one cluster, then termination and checkpointing are the thread's only function.

The *worker auxiliary thread* is a message-triggered thread that is active only on workers that are not also hubs, and whose function is related to load balancing, checkpoints, and termination. It listens for periodic messages from the worker's hub containing both cluster and system-wide workload information, or information relating to termination and checkpoints.

3.8 Enumeration in parallel

PEBBL's parallel layer supports the same multiple-solution enumeration criteria as its serial layer. For scalability, storage of the repository is partitioned approximately equally among all processors through a mapping based on the solution hash value — every solution s has a unique "owning" processor based on its hash value. The message-triggered *repository receiver thread* listens for and processes all messages related to parallel enumeration.

When enumeration is active and a solution s is passed to `foundSolution` on processor p_0, `foundSolution` immediately uses s's hash value to compute its owning processor $p(s)$. Unless $p(s) = p_0$, `foundSolution` immediately sends the s to processor $p(s)$. Employging the hash table representation of the repository and the `duplicateOf` method, processor $p(s)$ then checks whether s is a duplicate of a solution already in its local segment of the repository. If so, it simply discards s. Otherwise, it gives s a unique ID $(p(s), k)$ consisting of the processor number $p(s)$ and a serial number k, and enters s into the local repository.

If only the `enumRelTol`, `enumAbsTol`, or `enumCutoff` criteria are set, this storage logic

is essentially all that is needed to enumerate solutions in parallel. A synchronous parallel sort-merge operation is used at the end of the run to write the repository to an output file in objective function order. However, if `enumCount` is set, then the implementation becomes more complicated. For proper pruning, one would like to have an estimate of the `enumCount`$^{\text{th}}$-best solution in the union of the repository segments of all processors, which we call the *cutoff solution*. To keep track of the cutoff solution, we organize all processors into a balanced tree. The heap representation of each processor's repository segment is sorted by reverse objective function order (as in the serial layer), and in case of objective value ties, secondarily by solution ID.

Messages sent up the tree contain sorted arrays consisting of at most `enumCount` triples, each triple describing a solution. These triples have the form (z, q, k), where z is the objective value, q is the owning processor, and k is the serial number. Each processor stores the most recent arrays received from its children (if any). Periodically, each processor p performs a sort-merge operation, truncated after `enumCount` elements, combining its own repository segment data with the sorted arrays from its children. The result is a sorted array of the best (up to) `enumCount` solutions in the subtree rooted at p. If p is not the root, it sends this array to its parent. At the root, the last element of the merged array identifies the cutoff solution; this information is broadcast down the tree. This value is used to prune both local repository segments and subproblem pools.

A second enumeration-related thread, the repository merger thread, is responsible for *sending* some of the messages involved in processing `enumCount` in parallel. It is designed to limit total message traffic when large numbers of new solutions are being generated.

It is interesting to consider whether linear-time selection algorithms [7] could be adapted to track the cutoff value more efficiently than in PEBBL's present implementation. There are parallel adaptations of these algorithms [2, 14], but whether they could be further adapted to the dynamic, "online" environment needed to track the cutoff value is an open question.

PEBBL also has an option to enforce strict flow control on messages for the repository receiver thread; this feature is useful when PEBBL applications that generate large numbers of potential solutions are run using older versions of MPI that may malfunction if too many messages are simultaneously received at a single processor. In this case, all processors are organized into a (possibly incomplete) hypercube and messages are relayed only along the edges of this hypercube, with a flow control protocal on each edge. For most PEBBL applications and modern implementations of MPI, this optional feature is not necessary; it is controlled by a run-time parameter and disabled by default.

3.9 Synchronous ramp-up support

PEBBL provides support for specialized, synchronous ramp-up procedures at the beginning of the search process. This feature was developed to support the particular ramp-up procedure implemented for MIP in PICO, but has proved useful in other applications such as the MMA application described in Section 4.

For many branch-and-bound applications, the search tree should eventually grow to a

size that provides the most extensive and easily-exploited source of parallelism. Early in the search process, the tree is small, and particular applications may have different opportunities for parallelism that are superior. For example, PICO chooses mixed integer programming branching variables through a scheme that involves solving two essentially independent auxiliary linear programs for each newly encountered fractional variables in the LP relaxation. Early in the search, there are few branch-and-bound nodes, and many newly fractional variables per node, so these auxiliary LP's may provide significantly more parallelism than the search tree.

PEBBL provides support for a synchronous ramp-up phase with an application-defined *crossover* point. During the ramp-up phase, every processor redundantly develops an exactly identical branch-and-bound tree. As they synchronously process each subproblem, the processors are free to exploit any parallelism they wish in executing the `boundComputation`, `splitComputation`, or `makeChild`, or the incumbent heuristic. In general during this process, different processors will find different incumbents by, for example, using different seeds for randomized searches. Therefore, PEBBL provides a `rampUpIncumbentSync` method to make sure incumbent information and repository information (if any) is synchronized between processors. Otherwise, processors typically start processing their trees in divergent ways, and the ramp-up procedure tends to deadlock. The complexity of `rampUpIncumbentSync` depends on whether enumeration is being used and whether `enumCount` is enabled. When enumeration is active, the method must perform an all-to-all communication to make sure that each solution is hashed to the correct "owner" processor. The complexity of this step is potentially high and depends on the number of different solutions generated, so applications may wish to avoid generating large number of candidate solutions during ramp-up while enumerating. Depending on whether `enumCount` is in use, the hashing step is followed either by a tree-based merge operation or some simple MPI reductions. If enumeration is not active, `rampUpIncumbentSync` consists only of some simple MPI reductions to make sure all processors agree on the incumbent value and which processor is storing the current incumbent.

The boolean method `parallelBranching::continueRampUp` controls termination of the ramp-up phase. PEBBL provides a default implementation, but applications may override it. PEBBL terminates the ramp-up phase when this method returns `false`. At this point, since they all have copies of the same leaf nodes of the search tree, the worker processors are able to partition these subproblems nearly equally without any communication: a simple scheme to accomplish this is to number the subproblems from best to worst objective value, starting at 0, and have the first worker take ownership of all subproblems numbered 0 (mod W), where W is the number of workers, and discard all other subproblems. The second worker takes ownership of all subproblems numbered 1 (mod W), discarding all others, and so forth. The actual scheme used by PEBBL is similar but slightly more complicted in a manner that promotes even distribution of workload on the cluster level as well as on the level of individual workers. After partitioning of the active subproblems, PEBBL starts its scheduler, and enters its principle, asychronous, thread-based search mode.

Typically, the criterion for the `continueRampUp` method should not be that the tree has

grown large enough to fill the available processors — that could extend the ramp-up phase far too long. Instead, the criterion should be that the the parallelism in the search tree appears to exceed any alternative source of parallelism available to the application.

3.10 Phases of execution and termination detection

A typical PEBBL run is a sequence of four phases:

1. Problem read-in and broadcast
2. Synchronous ramp-up
3. Main asynchronous search
4. Solution and statistics output.

The read-in stage is straightforward: processor 0 reads the problem instance data and broadcasts it to all other processors. The synchronous ramp-up stage was described in the previous section. The next step is the main search and usually accounts for the vast majority of the run time. It is the only phase in which the scheduler and threads are activated, and the only one in which PEBBL operates asynchronously. Finally, the solution output phase is also relatively straightforward: in the absence of enumeration, the processor p^* holding the final incumbent solution simply outputs it directly or through the MPI-specified I/O processor. When enumeration is being used, this phase is more complicated, but consists essentially of a synchronous parallel sort-merge operation to output the entire repository in objective-value order.

One of the most delicate aspects of this sequence concerns detecting the end of the asynchronous search phase. Detecting termination is simple for many parallel programs that are organized on a strictly "master-slave" or hierarchical principle. PEBBL has a more complicated messaging pattern with multiple asychronous processes, even when there is only a single cluster, and this property introduces subtleties into the detection of termination.

Essentially, PEBBL terminates when it has detected and confirmed *quiescence*, the situation in which all worker subproblem and hub token pools are empty, and all sent messages have been received. To confirm quiesence, PEBBL uses a method derived from [38]. All PEBBL processors keep count of the total number of messages they have sent and received (other than load-balancing sweep and termination detection messages). Within each cluster, this information in accumulated and monitored at the hub. Between clusters, the load balancing survey sweep sums the total workload information and message counts as it propagates data up the load balancing tree. Thus, it is straightforward for processor 0, the hub of the cluster at the root of the load balancing tree, to detect the situation in which the subproblem workload in the system sums to zero, and the total counts of sent and received messages are in balance. We call this situation *pseudoquiesence*. Pseudoquiesense is necessary for true termination of the search, but it is not sufficient because measurements making up the various sums involved are typically not taken at exactly the same time.

Detection of pseudoquiesence is a two-stage process in PEBBL. The first level of pseudoquiesence includes only messages involved in work distribution and is easily detected at processor 0 as part of the ordinary process of load balancing. For efficiency during enumeration, this first level of detection does not include messages involved in incumbent and repository management. If the first level of pseudoquiesence is detected, then the load balancing threads on the hubs and worker auxiliary threads on the workers cooperate to actively check whether the total count of all messages sent and received appears to balance. This check involves a complete sweep of the load-balancing tree of all hub processors, and a poll-and-reply communication between every hub and each of its workers. This process is called a *quiesence check*. If the quiesence check confirms that pseudoquiesence has indeed occurred, PEBBL proceeds to a second check, the *termination check*.

Suppose the quiescence check has confirmed pseudoquiesence, and the total number of messages sent and received is m. It is shown in [38] that to confirm that true quiesence has occurred, it is sufficient to perform one additional measurement of the total number of messages sent at every processor, and verify that its sum is still m. Thus, the downward phase of each load-balancer survey sweep that detects pseudoquiesence initiates precisely this additional measurement: each hub queries each of its workers, and adds their replies to its own message-sent count. In a process we call the *termination-check sweep*, the load balancing thread adds these values up the load balancing tree, and then broadcasts the sum down. If the sum exceeds m, quiesence was not truly achieved, and PEBBL resumes normal operation until it detects pseudoquiesence again. If the sum is m, each hub sends a termination message to each of its workers, and then terminates the search.

The basic form of PEBBL's termination detection scheme is taken from [38]. We added a second level of checking to improve efficiency under our particular clustered form of processor organization. Note also that when there is just one hub, the scheme is somewhat streamlined and simplified.

3.11 Checkpointing

PEBBL's checkpointing feature is implemented only in the parallel layer, and it is closely related to its termination detection mechanism. Its purpose is to allow partial runs of a PEBBL branch-and-bound search to be resumed at a later time without a large waste of computational resources. Large-scale multiprocessing systems may only allocate limited amounts of time to each user, and they can be less reliable than workstation-scale systems. Thus, it is useful to have some means of recovery from a job time-out or hardware failure. PEBBL's design is focused on maximum scalability and speedup, and it does not provide true fault tolerance as in some alternative packages. For example, there is no redundancy in subproblem computations and no special checks for potential corruption of messages. For high-consequence computations, we are planning for PEBBL-based systems such as PICO to instead produce a certificate of correctness that can be externally checked; however, this system is not yet operational.

PEBBL can be configured to write a checkpoint approximately every t minutes after

the synchronous ramp-up phase. When processor 0, which serves as the hub for the first cluster, detects that it is time to write a checkpoint, it sets a flag which it propagates to all hubs through the downward phase of the load-balancing survey sweep. The hubs in turn propagate the flag to their respective workers. When the checkpointing flag is set on a worker processor, all compute threads assume the blocked state. On hub processors, the checkpointing flag also disables all assignment of subproblems to workers. At this point, the load balancing begins operating in the same manner as in termination detection, except that it does not require the number of active subproblems on every processor to be zero. Once the termination-detection mechanism has determined that all messages sent have been received, the system is in a stable state from which it is possible to write a consistent checkpoint. Each processor then writes its own binary checkpoint file. If each processor has its own disk, or the system has parallel I/O capabilities, this operation is potentially parallel. The checkpoint-writing process uses the same binary pack and unpack operations employed for MPI message operations. PEBBL also provides virtual methods so that the checkpoint can include additional application-specific information that might need to be saved. Once it has written its checkpoint file, each processor clears its checkpoint flag, allowing any compute threads to return to the ready state, and permitting the distribution of subproblems to resume.

PEBBL provides two methods, called *restart* and *reconfigure*, for restarting a run from a collection of checkpoint files. For a restart, the number of processors and the processor cluster configuration must exactly match the run that wrote the checkpoint. In this case, each processor p reads checkpoint file p, a potentially parallel operation.

For a reconfigure, the number of processors and the clustering arrangement may be different from the one that wrote the checkpoint. Processor 0 simply reads the checkpoint files one-by-one, and distributes subproblems as evenly as possible to the worker processors using a round-robin message pattern. The reconfigure mechanism is more flexible than the restart mechanism, but potentially much slower, since it reads all checkpoint files serially.

Note that some implementations of MPI provide a means of checkpointing large parallel runs without having to include specific supporting code in the application. Such mechanisms could in some cases substitute for PEBBL's checkpointing scheme, but might be less efficient.

4 Application to maximum monomial agreement

4.1 The MMA problem and algorithm

To provide an example of PEBBL's performance and capabilities, we now describe its application to the maximum monomial agreement (MMA) problem. Eckstein and Goldberg [21] describe this problem and an efficient serial branch-and-bound method solving it. An earlier, less efficient algorithm is described in [26], and a slightly more general formulation of the same problem class may be found in [16]. The algorithm described in [21] uses a combinatorial bound not based on linear programming, and it significantly outperforms approaches based on modeling the problem as a MIP and using a standard professional-quality MIP

solver. This property makes MMA a practical example of applying branch-and-bound in a non-MIP setting. Conveniently, the algorithm in [21] had already been coded using the PEBBL serial layer, so it was only necessary to extend the implementation to incorporate the parallel layer. We now give a condensed description of the MMA problem and solution algorithm. Further details may be found in [21].

The MMA problem arises as a natural subproblem in various machine learning applications. Each MMA instance consists of set of M binary N-vectors in the form of a matrix $A \in \{0,1\}^{M\times N}$, along with a partition of its rows into "positive" observations $\Omega^+ \subset \{1,...,M\}$ and "negative" observations $\Omega^- = \{1,...,M\} \setminus \Omega^+$. Row i of A, denoted by A_i, indicates which of N binary features are possessed by observation i. In a medical machine learning application, for example, each feature could represent the presence of a particular gene variant or the detection of a particular antibody, while Ω^+ could represent a set of patients with a given disease and Ω^- a set of patients without the disease.

Each MMA instance also includes a vector of weights $w_i \geq 0$ on the observations $i = 1,\ldots,M$. A *monomial* on $\{0,1\}^N$ is simply some logical conjunction of features and their complements, that is, a function $m_{J,C} : \{0,1\}^N \to \{0,1\}$ of the form

$$m_{J,C}(x) = \prod_{j\in J} x_j \prod_{c\in C}(1-x_c), \tag{1}$$

where J and C are (usually disjoint) subsets of $\{1,\ldots,N\}$. The monomial $m_{J,C}$ is said to *cover* a vector $x \in \{0,1\}^N$ if $m_{J,C}(x) = 1$, that is, if x has all the features in J and does not have any of the features in C. We define the *coverage* of a monomial $m_{J,C}$ to be

$$c_{J,C} \stackrel{\text{def}}{=} \big\{i \in \{1,...,M\} \;\mid\; m_{J,C}(A_i) = 1\big\}.$$

Define the *weight* of a set of observations $S \subseteq \{1,\ldots,M\}$ to be $w(S) = \sum_{i\in S} w_i$. The MMA problem is to find a monomial whose coverage "best fits" the dataset A as weighted by the weights w, in the sense of matching a large net weight of observations in Ω^+ less those matched in Ω^-, or *vice versa*. Formally, we wish to find subsets $J, C \subseteq \{1,\ldots,N\}$ solving

$$\begin{array}{ll} \max & \big|w(c_{J,C}\cap\Omega^+) - w(c_{J,C}\cap\Omega^-)\big| \\ \text{s.t.} & J, C \subseteq \{1,\ldots,N\}. \end{array} \tag{2}$$

When the problem dimension N is part of the input, [21] proves that this problem formulation is $\mathcal{NP}$-hard, using techniques derived from [34].

The main ingredients in any branch-and-bound algorithm are a subproblem description, a bounding function — for a maximization problem like the MMA, an upper bounding function — and a branching rule. For MMA, each possible subproblem is described by some partition (J, C, E, F) of $\{1,\ldots,N\}$. Here, J and C respectively indicate the features which must be in the monomial, or whose complements must be in the monomial. E indicates a set of "excluded" features: neither they nor their complements may appear in the monomial. Finally, $F = \{1,\ldots,N\}\backslash(J\cup C\cup E)$ is the set of "free", undetermined features. The root of the

branch-and-bound tree is the subproblem $(J, C, E, F) = (\emptyset, \emptyset, \emptyset, \{1, \ldots, N\})$. Subproblems with $F = \emptyset$ correspond to only one possible monomial and cannot be subdivided.

For each subproblem (J, C, E, F) where $F \neq \emptyset$, the upper bounding function described in [21] is

$$b(J, C, E, F) = \max \left\{ \begin{array}{l} \sum_{\eta=1}^{q(E)} \left(w(V_\eta^E \cap\ c_{J,C} \cap \Omega^+) - w(V_\eta^E \cap c_{J,C} \cap \Omega^-) \right)_+, \\ \sum_{\eta=1}^{q(E)} \left(w(V_\eta^E \cap\ c_{J,C} \cap \Omega^-) - w(V_\eta^E \cap c_{J,C} \cap \Omega^+) \right)_+ \end{array} \right\}. \quad (3)$$

Here, we use the notation $(x)_+ = \max\{0, x\}$, and the sets $V_1^E, \ldots, V_{q(E)}^E \subseteq \{1, \ldots, M\}$ denote the equivalence classes of the observations induced by the set of excluded features E: essentially, i and i' are in the same equivalence class V_η^E if and only if A_i and $A_{i'}$ differ only in features j that are in E. The analysis in [21] shows that $b(J, C, E, F)$ is a valid upper bounding function, that is, that the objective value of any solution (J', C') consistent with (J, C, E, F) is no higher than $b(J, C, E, F)$. Formally,

$$\left| w(c_{J',C'} \cap \Omega^+) - w(c_{J',C'} \cap \Omega^-) \right| \leq b(J, C, E, F) \quad \forall\, J', C' : \begin{array}{l} J \subseteq J' \subseteq J \cup F, \\ C \subseteq C' \subseteq C \cup F. \end{array}$$

In general, the bound is somewhat time consuming to compute due to the effort required to determine the equivalence classes $V_1^E, \ldots, V_{q(E)}^E$, but it yields much better practical results than simpler bounding functions. Finally, when $F = \emptyset$, the suproblem (J, C, E, F) describes a subset of the set of monomials consisting of just one element, $m_{J,C}$, so we set $b(J, C, E, F) = \left| w(c_{J,C} \cap \Omega^+) - w(c_{J,C} \cap \Omega^-) \right|$ instead of using (3).

The final ingredient required to describe the branch-and-bound method is the branching procedure. Here we only use the most efficient of the branching schemes tested in [21], a ternary lexical strong branching rule. Given a subproblem (J, C, E, F), this method evaluates $|F|$ possible branches, one for each element j of F. Given some $j \in F$, there are three possibilities: either j will be in the monomial, its complement will be in the monomial, or j will not be used in the monomial. These possibilities respectively correspond to the three subproblems, $(J \cup \{j\}, C, E, F \backslash \{j\})$, $(J, C \cup \{j\}, E, F \backslash \{j\})$, and $(J, C, E \cup \{j\}, F \backslash \{j\})$. We use this three-way branching of (J, C, E, F), selecting j through a lexicographic strong branching procedure. Specifically, for each member of j, we compute the three prospective child bounds $b(J \cup \{j\}, C, E, F \backslash \{j\})$, $b(J, C \cup \{j\}, E, F \backslash \{j\})$, and $b(J, C, E \cup \{j\}, F \backslash \{j\})$, round them to 5 decimal digits of accuracy, place them in a triple sorted in descending order, and then select the j leading to the lexicographically smallest triple. As a byproduct of this procedure, we also easily obtain a potentially tighter bound on the the objective value of any solution (J', C') corresponding to (J, C, E, F), namely the "lookahead" bound

$$\bar{b}(J, C, E, F) = \min_{j \in F} \left\{ \max \left\{ \begin{array}{l} b(J \cup \{j\}, C, E, F \backslash \{j\}) \\ b(J, C \cup \{j\}, E, F \backslash \{j\}) \\ b(J, C, E \cup \{j\}, F \backslash \{j\}) \end{array} \right\} \right\}. \quad (4)$$

A fourth possible component of a branch-and-bound scheme is an incumbent heuristic. For the MMA problem, there is no difficulty in identifying feasible solutions, and we use

the straightforward strategy of [21], which is simply to use (J, C) as a trial feasible solution whenever processing a subproblem (J, C, E, F). It is possible that the application could benefit from some kind of "smart" heuristic search in identifying sets J', C' with a good objective value, where $J \subseteq J' \subseteq J \cup F$, $C \subseteq C' \subseteq C \cup F$. We have not explored this topic because our primary focus here is not extending or improving the underlying algorithm of [21], but showing how PEBBL can provide an efficient parallel implementation.

4.2 Parallel implementation in PEBBL

We took the implementation tested in [21], which already used the PEBBL serial layer, and extended it to use the PEBBL parallel layer. The fundamental part of this effort was the creation of `pack` and `unpack` routines to allow problem instance and subproblem data to be respectively written to and read from MPI buffers. This step provides basic parallel functionality. In converting a serial PEBBL application to a parallel one there are in general three additional, optional implementional steps which may be taken to improve parallel performance:

1. Implementating a synchronous ramp-up procedure, if the application is amenable to one.

2. Creating an enhanced incumbent heuristic that can run semi-independently from the search process as part of the incumbent heuristic thread.

3. Implementing a strategy for interprocessor communication of application-specific information other than information that can be included in the initial problem instance broadcast or within individual subproblems.

The second and third of these steps are not applicable to the MMA algorithm described above, because of its simple procedure for generating incumbent solutions, and its lack of any "extra" data structures that cannot be naturally embedded into the initial problem instance data or the description of each subproblem. Consequently, we did not enable the incumbent heuristic thread, and we did not have to add any application-specific threads to the PEBBL parallel layer's standard complement of threads.

However, because of the strong branching procedure, the MMA algorithm of Section 4.1 does have a secondary source of parallelism that lends itself naturally to PEBBL's synchronous ramp-up phase. The time needed to process each subproblem (J, C, E, F), especially near the root of the branch-and-bound tree, tends to be dominated by the $|F|$ calculations of $b(J \cup \{j\}, C, E, F \backslash \{j\})$, $b(J, C \cup \{j\}, E, F \backslash \{j\})$, and $b(J, C, E \cup \{j\}, F \backslash \{j\})$, one such triple for each $j \in F$, required by the strong branching procedure. These calculations are independent, and thus readily parallelizable. Therefore, we used the PEBBL parallel layer's specialized ramp-up capabilities to implement a synchronous ramp-up phase that takes advantage of this parallelism.

During the ramp-up phase, all processors develop the same search tree in an essentially synchronous and partially redundant manner. In the strong-branching phase of processing

each subproblem, however, we divide the $|F|$ undecided features as evenly as possible between the available processors. Then, for the features j which it has been allocated, each processor computes the triple

$$\begin{pmatrix} b(J \cup \{j\}, C, E, F \backslash \{j\}) \\ b(J, C \cup \{j\}, E, F \backslash \{j\}) \\ b(J, C, E \cup \{j\}, F \backslash \{j\}) \end{pmatrix}.$$

It rounds the elements of these triples, sorts each one in descending order, and determines the lexicographically smallest one. Next using MPI's `Allreduce` reduction function with a customized MPI datatype and reduction operator, we determine the $j \in F$, across all processors, leading to the lexicographically minimum triple. This j becomes the branching variable. Another MPI `Allreduce` operation computes the tightened "lookahead" bound (4).

This synchronous ramp-up phase proceeds until the the size of the pool of active search nodes becomes comparable to the number of features M in the problem instance. At this point, we perform the crossover operation described in Section 3.9, and enter PEBBL's main asynchronous search phase. The crossover decision was implemented by overriding PEBBL's `continueRampUp` virtual method. The specific test performed is to terminate the synchronous ramp-up phase when $|\mathcal{P}| > \rho M$, where $\mathcal{P}$ is the set of active search nodes and ρ is a run-time parameter. In some brief experimentation, we found that $\rho = 1$ yielded good performance, so we used that value in most of our experimental tests. Below, we refer to ρ as the "ramp-up factor".

We contemplated making the ramp-up procedure even more parallel by dividing the body of the strong branching procedure into $3\,|F|$ rather than $|F|$ parallel tasks, one for each potential child bound. We decided not to attempt this enhancement, however, because of the additional complexity of computing sorted triples of bounds computed on different processors, and because of the potential lack of balance between the tasks created: of the bounds of the three children corresponding to each potential branching feature $j \in F$, the calculation of $b(J, C, E \cup \{j\}, F \backslash \{j\})$ is typically the most time-consuming because it is the only one that requires recalculation of the equivalence classes V_η^E used in (3). Thus, the speedup attained by dividing each triple of potential child bounds into three subtasks could be strongly sublinear, making it uncertain whether such an enhancement would have a significant impact on performance.

4.3 Computational testing and scalability results

To demonstrate PEBBL's scalability, we tested our parallel MMA solver on "Red Sky", a parallel computing system at Sandia National Laboratories that consists of 2,816 compute nodes, each with two quad-core Intel Xeon X5570 processors sharing 48GB of RAM. The full system thus has 22,528 processing cores and 132TB of RAM. The system has various compute partitions, some dedicated to specific uses. The "general" partition has 2,144 nodes, and thus 17,152 processing cores and 100.5TB of RAM. This entire resource is usually not simultaneously available to a single user, but waiting times to gain access to hundreds of nodes, and thus thousands of cores, are typically short. Red Sky's compute nodes are

connected by an Infiniband interconnect arranged as a torroidal three-dimensional mesh, with a 10GB/second link data rate and end-to-end message latencies on the order of one microsecond.

Each Red Sky compute node runs a version of the Red Hat 5 Linux operating system. For this system, we compiled release 1.4 of PEBBL with the Intel 11.1 C++ compiler with `-O2` optimization, using the Open MPI 1.4.3 implementation of the MPI library.

The simplest way to use MPI on Red Sky is to launch 8 independent MPI processes on each processing node, one for each of the 8 cores. Thus, a job allocating ν compute nodes behaves as an MPI job with 8ν parallel "processors". By using shared memory areas rather than the Infiniband interconnect, MPI processes on the same node can communicate faster than processes on different nodes. However, we made no particular attempt to exploit this property in implementing PEBBL.

For runs on less than 8 "processors", we simply allocated a single compute node, but launched fewer than 8 MPI processes on it. Otherwise, we always used a multiple of 8 processes. Essentially, we tested all configurations with either $p = 2^k$ or $p = 3 \cdot 2^k$ processor cores in the range $1 \leq p \leq 8192$, with the exception of $p = 12$, since 12 is neither less than 8 nor a multiple of 8.

For our tests, we used a collection of nine difficult MMA test instances derived from two data sets in the UC Irvine machine learning repository [1], each converted to an all-binary format using a procedure described in [8], and dropping observations with missing fields. Four of the problems are derived from the Hungarian heart disease dataset, the most challenging one tested in [21], and have 294 observations and 72 features. The remaining five instances were derived from the larger "spambase" dataset of spam and legitimate e-mails, and have 4,601 observations described by 75 binary features.

The only difference between different MMA instances derived from the same dataset is in the weights w_i, which strongly influence the difficulty of the instance. To generate realistic weights, we embedded our MMA solver within the LP-Boost column-generation method for creating weighted voting classifiers [15]. This procedure starts by applying its "weak learner", in this case the MMA solver, with all weights set equal to one another. As it proceeds, it uses a linear program to construct the best possible dataset classification thresholding function that is a linear combination of the weak learner solutions obtained so far. The dual variables of this linear program provide a new set of weights which are used as input to the weak learner to create a new term to add to the classification rule, and then the process repeats. As observed in [21], the MMA instances generated in this manner tend to become progressively more difficult to solve as the algorithm proceeds. The problems derived from the Hungarian heart disease dataset, which we denote `hung23`, `hung46`, `hung110`, and `hung253`, respectively use the weights from iterations 23, 46, 110, and 253 of the LP-Boost procedure. Note that the problems derived from iterations before the 23$^{\text{rd}}$ were too easy to test in a large-scale parallel setting. The spam-derived instances are `spam`, `spam5`, `spam6`, `spam12`, and `spam26`, and respectively use the weights from iterations 1, 5, 6, 12, and 26 of LP-Boost.

In performing our testing, we left most of PEBBL's parameters in their default con-

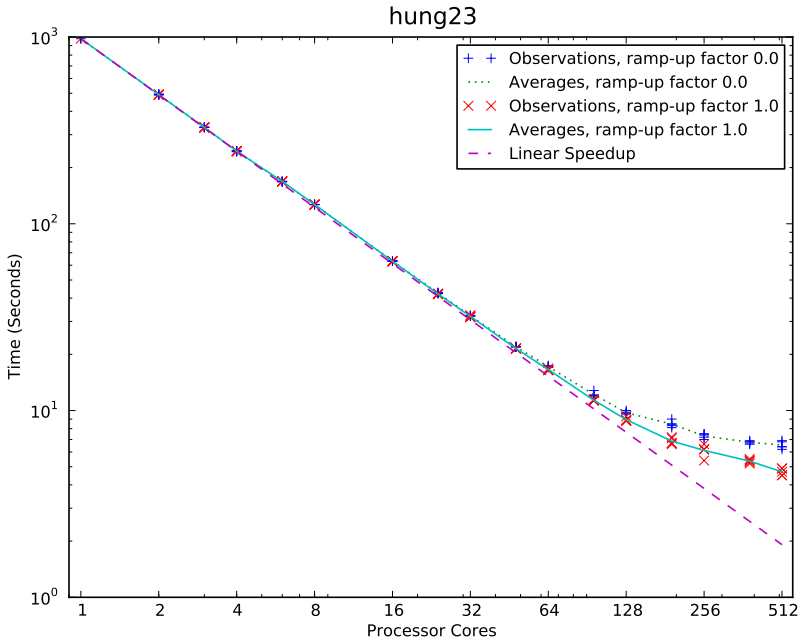

Figure 7: Speedup behavior on problem instance `hung23`.

figuration, except for those concerned with sizing processor clusters. Processing an MMA subproblem tends to be fairly time-consuing, due to the computational effort involved in computing the equivalence classes used in (3), which is multiplied by the requirements of the strong branching procedure. Because subproblems take relatively long to process — on the order of 0.01 to 0.02 seconds for the heart disease problems, and 0.7 to 0.8 seconds for the problems derived from the much larger spam dataset — a single hub can handle a large number of workers without becoming overloaded. Consequently, we set the cluster size to 128 processors (which in our Red Sky configuration means processor cores). In clusters below 64 processors, we configured the hub to double as a worker.

Our first set of tests did not use enumeration. To demonstrate the usefulness of PEBBL's synchronous ramp-up procedure, we ran each problem instance in two different modes, one with the default value of $\rho = 1$, and one with $\rho = 0$, which essentially disables the synchronous ramp-up procedure and starts the asynchronous search from the root subproblem. Because PEBBL's run-time behavior is not strictly deterministic, we ran each combination of problem instance and ρ value at least 5 times, with the exception of runs on a single processor core. Such single-core runs used only PEBBL's serial layer, which has deterministic behavior, so we timed them only once.

Figures 7-15 show the results, one figure per dataset. Each graph uses a log-log scale, with processor cores on the horizontal access and run time (in seconds) on the vertical axis. On such a graph, perfectly linear speedup is represented by a straight line. On each graph, the dashed straight line shows an extrapolation of perfect linear relative speedup from the smallest procssor configuration tested. The "+" marks depict the runs with $\rho = 0$,

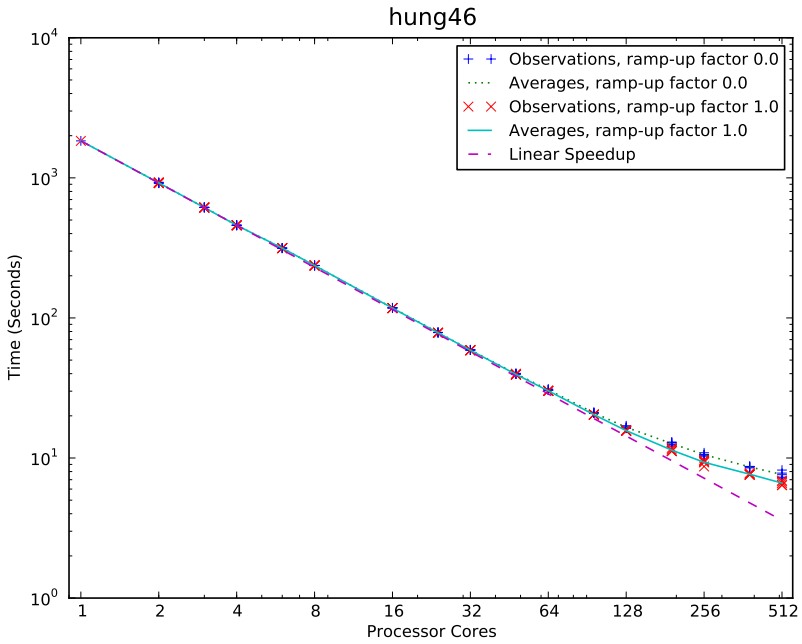

Figure 8: Speedup behavior on problem instance `hung46`.

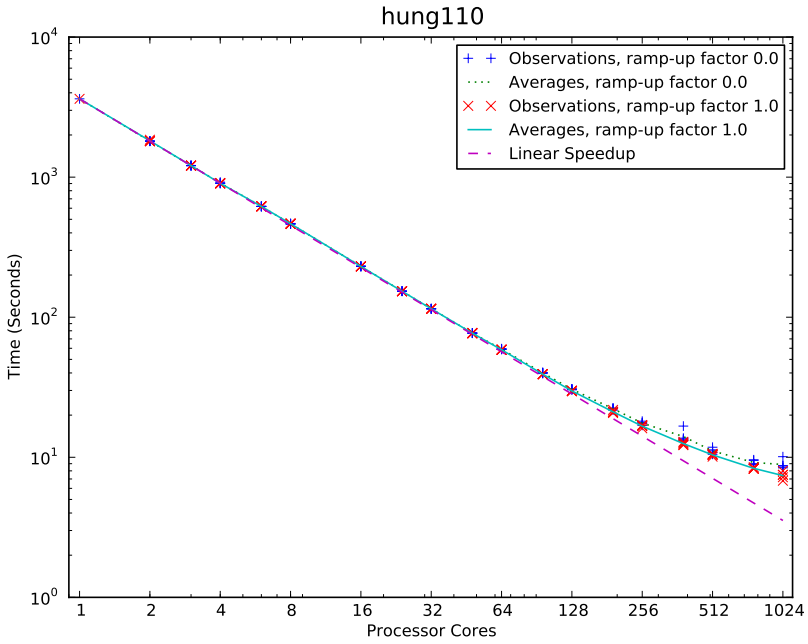

Figure 9: Speedup behavior on problem instance `hung110`.

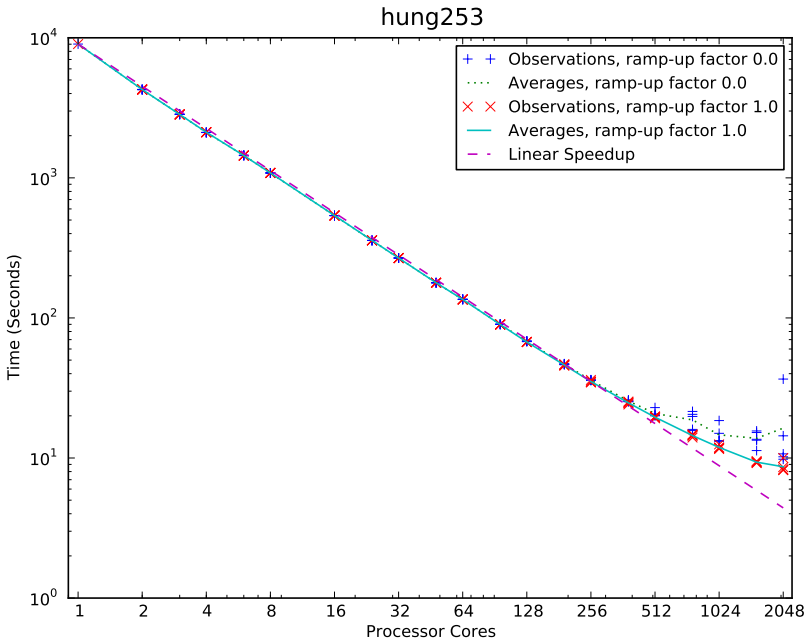

Figure 10: Speedup behavior on problem instance `hung253`.

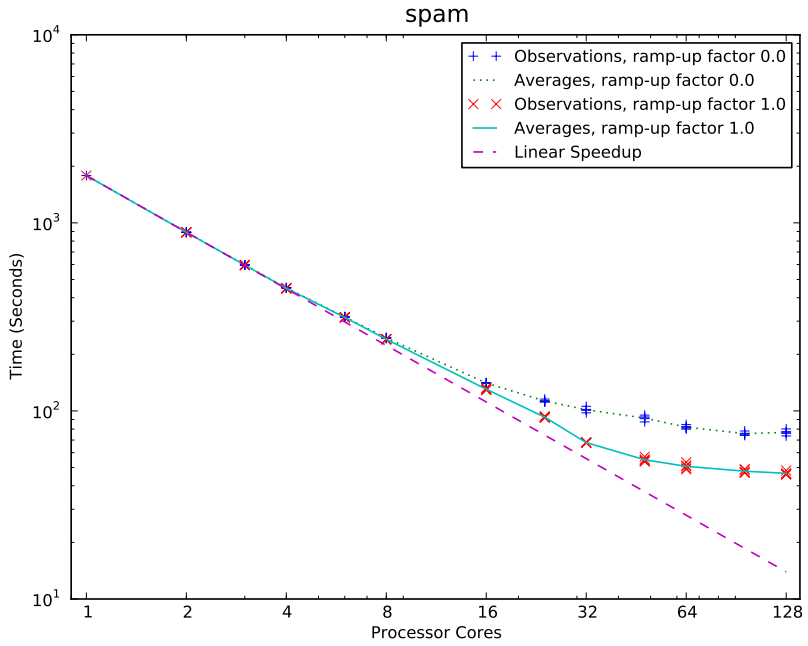

Figure 11: Speedup behavior on problem instance `spam`.

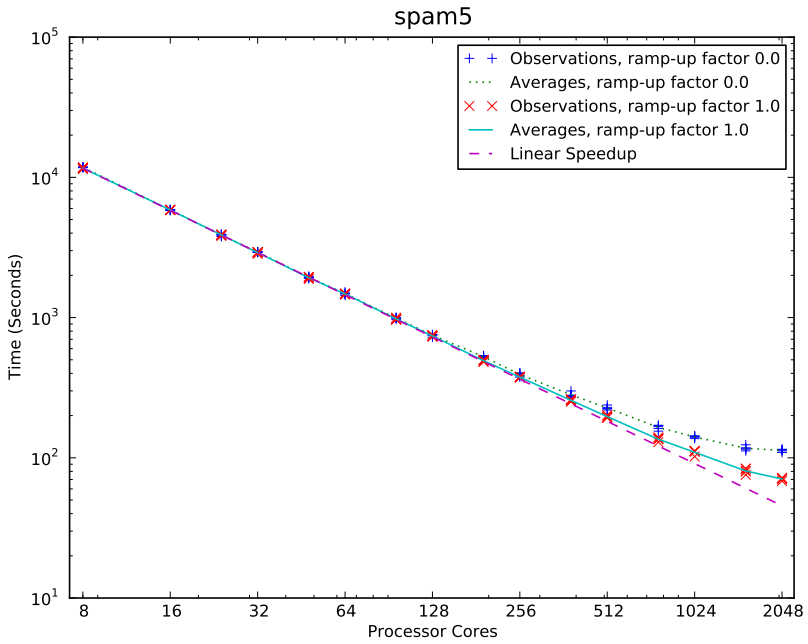

Figure 12: Speedup behavior on problem instance `spam5`.

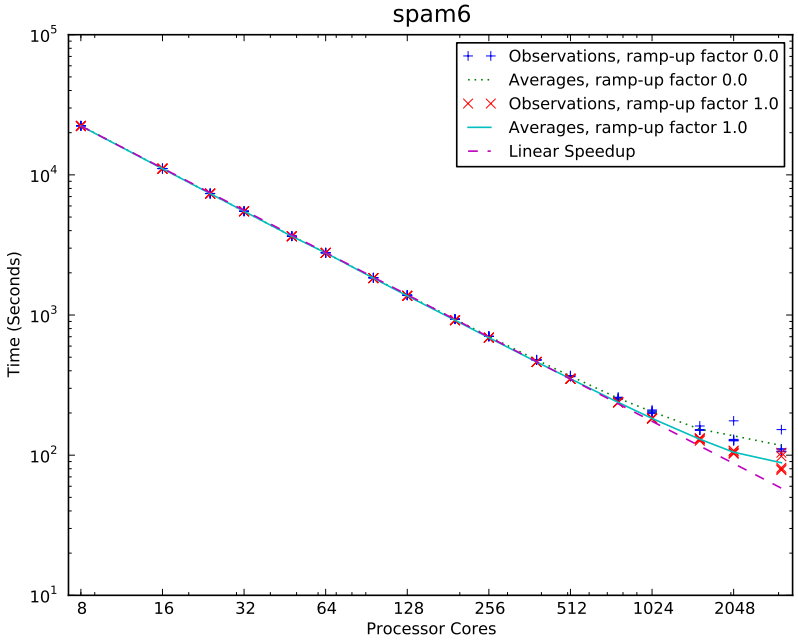

Figure 13: Speedup behavior on problem instance `spam6`.

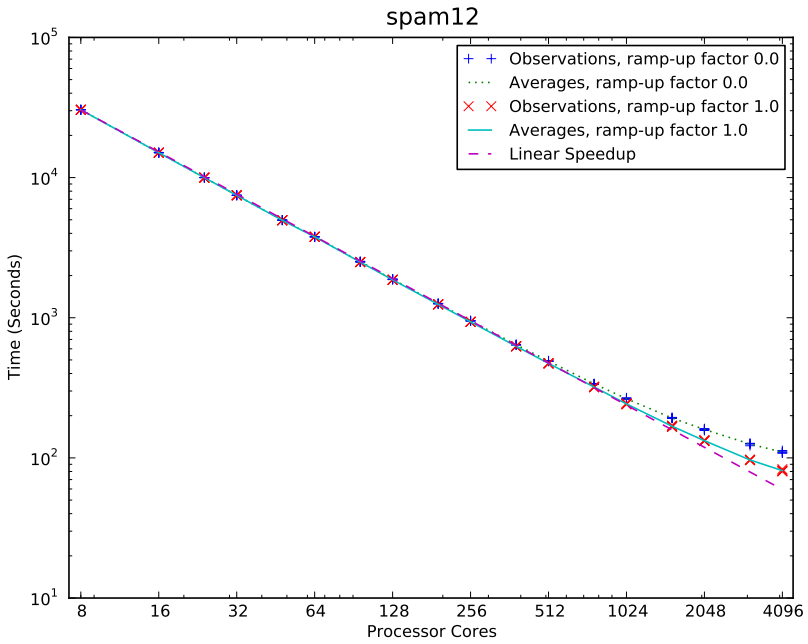

Figure 14: Speedup behavior on problem instance `spam12`.

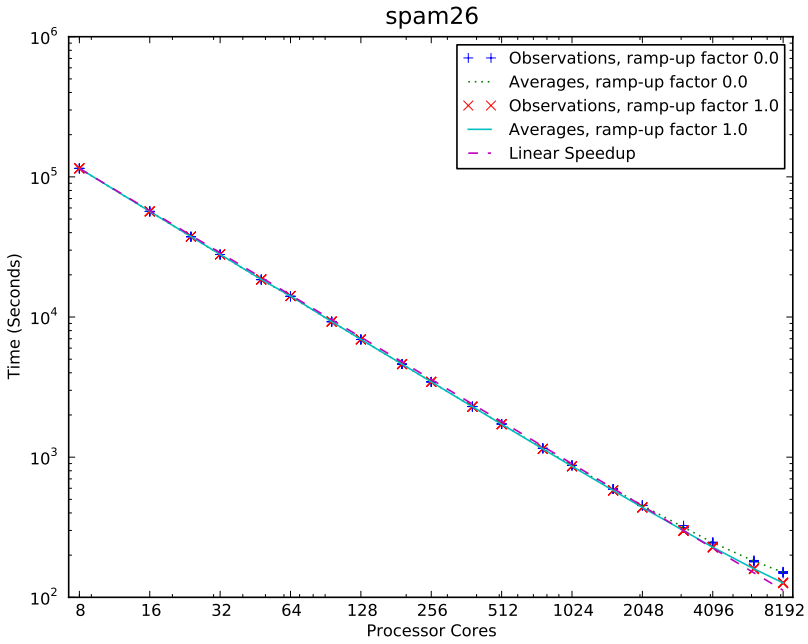

Figure 15: Speedup behavior on problem instance `spam26`.

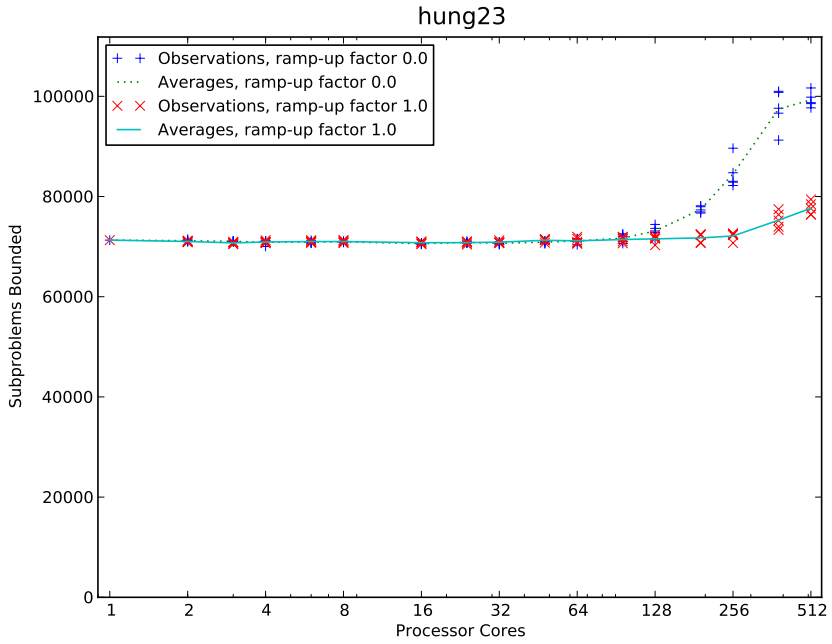

Figure 16: Number of nodes bounded for problem instance `hung23`.

that is, with the synchronous ramp-up phase of the run eliminated, while the the "×" marks represent the runs with $\rho = 1$, and thus a full synchronous ramp-up phase. For each combination of problem instance, number of processors, and ρ value, we computed the arithmetic mean of the sampled run times. The solid lines in each figure trace these means for $\rho = 1$, and the dotted lines show them for $\rho = 0$. The range of processor cores shown varies by problem instance: four of the spambase problems are too time-consuming to run on small processor configurations, and so their smallest number of processors is 8. For all the problem instances, we also stopped increasing the number of processor cores after the relative speedup level had departed significantly from linear, or after 8192 processor cores.

Figures 16-24 show the number of subproblems bounded for a representative subset of the runs. These charts have the same format as the speedup graphs, except that the vertical axis is linear and there is no ideal speedup line. The number of subproblems bounded varies from approximately 2400 for the `spam` problem to approximately two million for `spam26`.

The following observations are apparent from the graphs:

- Using a synchronous ramp-up phase significantly improves scalability for all problem instances. For the spambase-derived problem instances, the synchronous ramp-up phase at least doubles the size of the largest processor configuration on which near-linear speedup persists. The synchronous ramp-up phase reduces "tree inflation", the tendency of the explored search tree to grow when more processors are applied.

- Speedups are close to linear over a wide range of processor configurations for all the problem instances, with the point of departure from linear speedup depending on

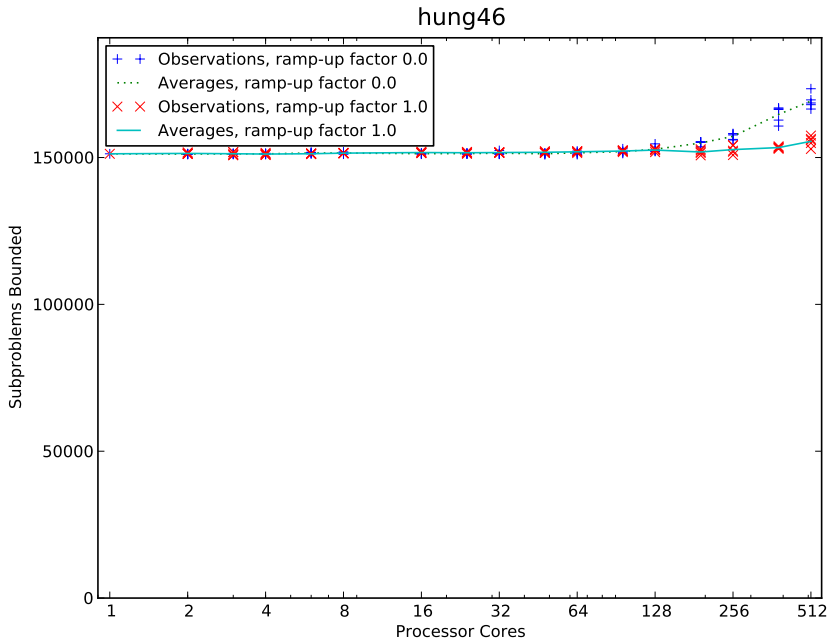

Figure 17: Number of nodes bounded for problem instance `hung46`.

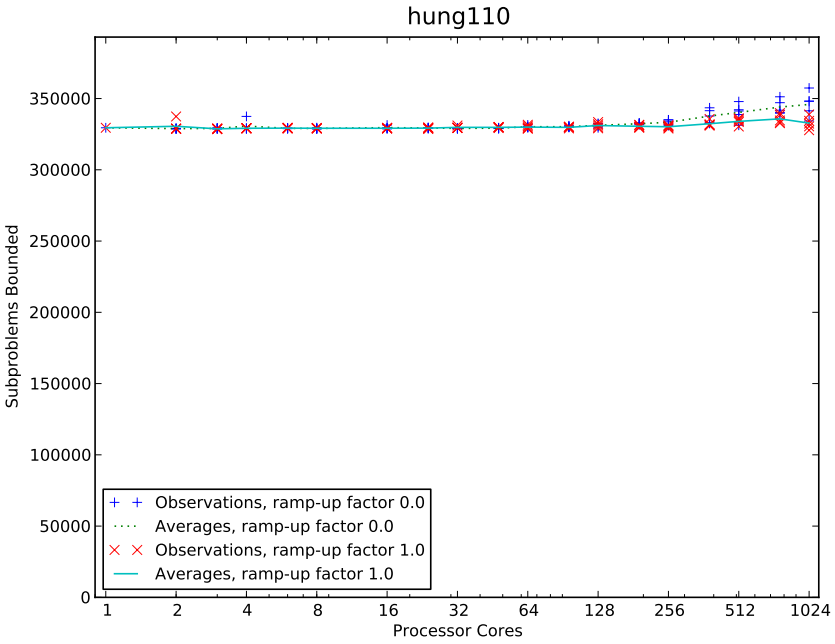

Figure 18: Number of nodes bounded for problem instance `hung110`.

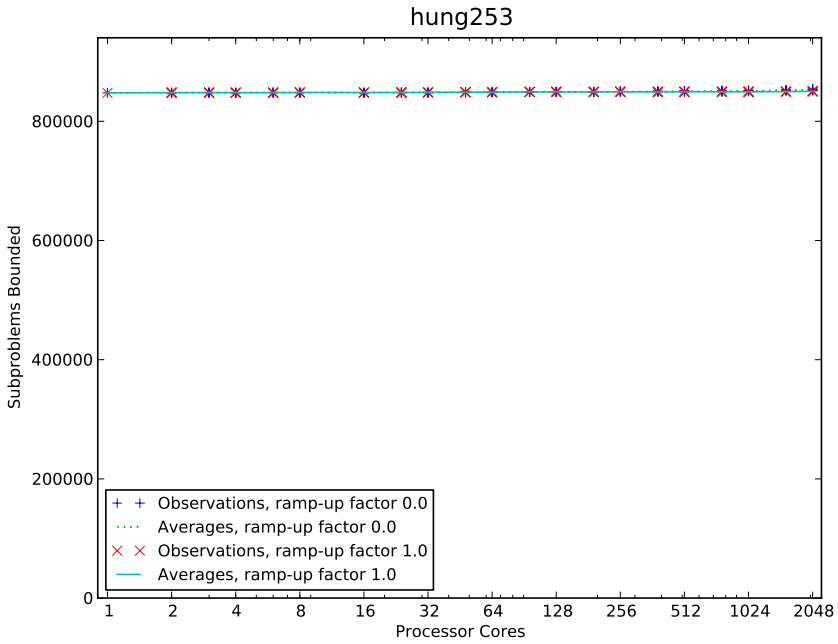

Figure 19: Number of nodes bounded for problem instance `hung253`.

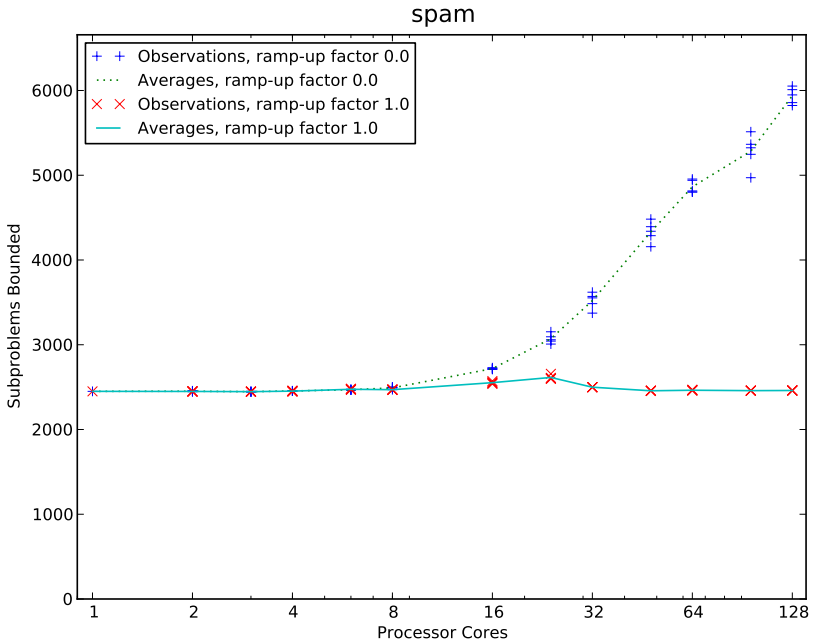

Figure 20: Number of nodes bounded for problem instance `spam`.

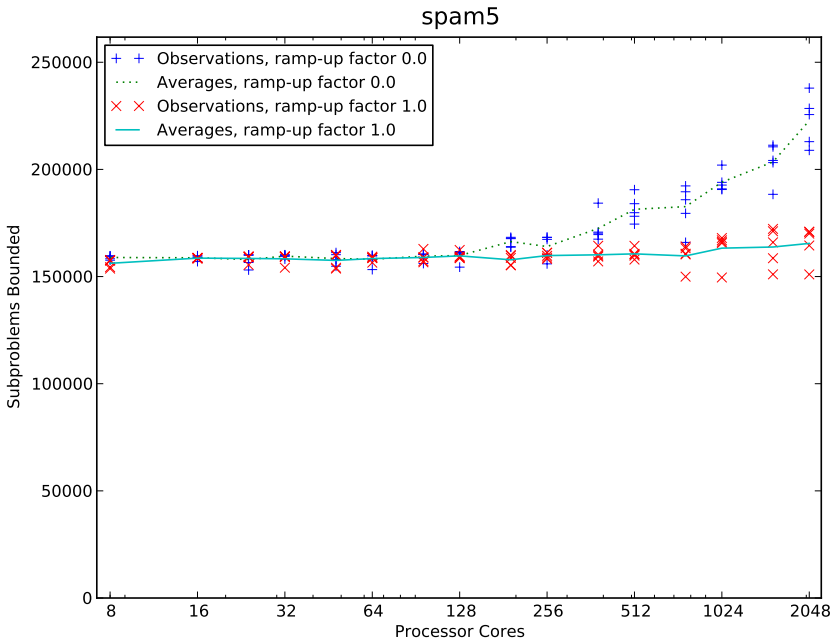

Figure 21: Number of nodes bounded for problem instance `spam5`.

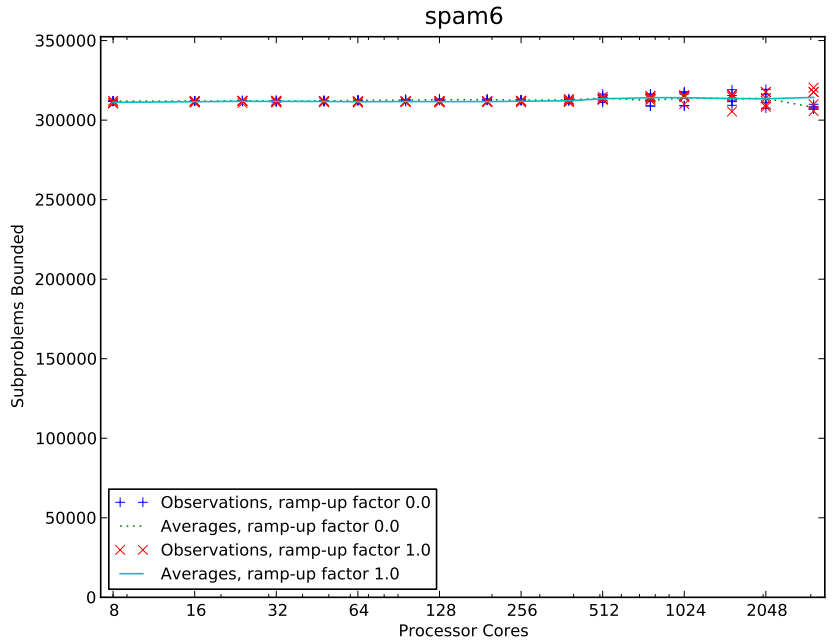

Figure 22: Number of nodes bounded for problem instance `spam6`.

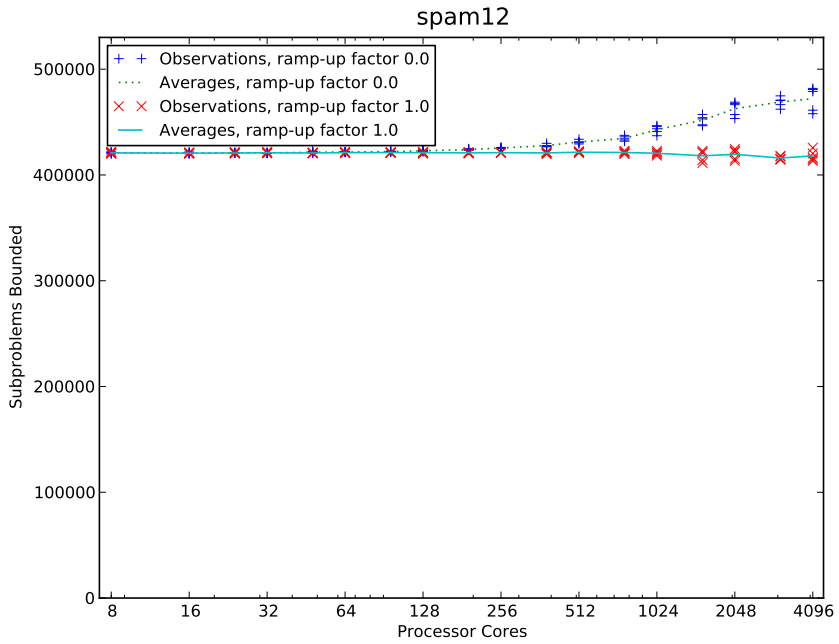

Figure 23: Number of nodes bounded for problem instance `spam12`.

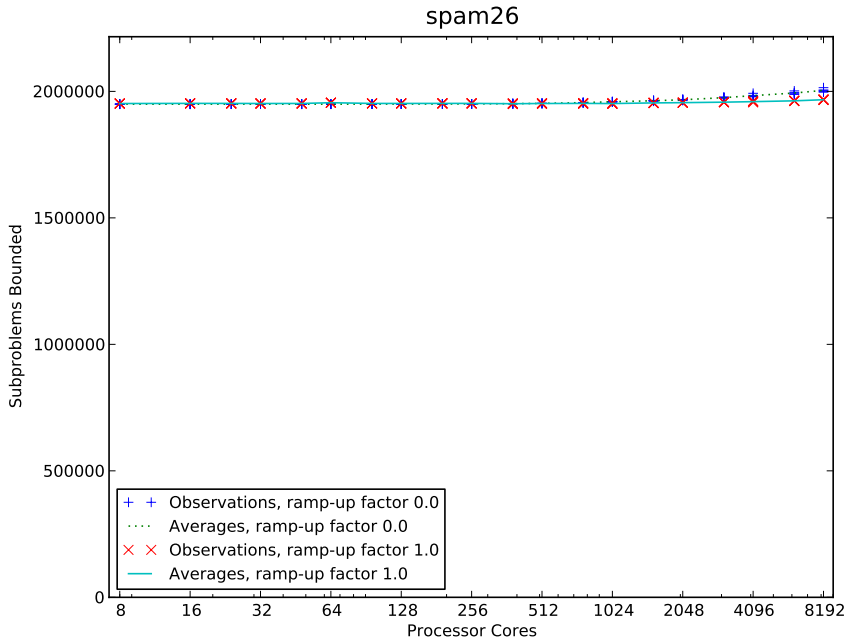

Figure 24: Number of nodes bounded for problem instance `spam26`.

the difficulty of the problem and the size of its search tree. For the problem with the smallest search tree, `spam`, behavior departs signficantly from linear at about 48 processor cores for $\rho = 1$ and 16-24 processor cores for $\rho = 0$. For the instance with the largest search tree, `spam26`, relative speedup remains essentially linear even at 6,144 processor cores.

- By applying a sufficient number of processors, the solution time for each instance can be reduced to the range of approximately 1-3 minutes, apparently without regard to the serial solution time. For example, `spam26` can be solved in less than 3 minutes on 6,144 processor cores, whereas solving it on 8 processor cores takes over 27 hours, from which it may be extrapolated that solution on one processor core would require over 9 days.

- Since there is no significant departure from the ideal linear speedup line when moving from 1 processor core to 2 processor cores, it may be inferred that the overhead imposed by moving from PEBBL's serial to parallel layer is essentially negligible for the MMA class of problems.

- There is no noticeable loss of efficiency in moving from a single processor cluster, as in the runs with 128 or fewer cores, and multiple processor clusters. For example, the `spam26` instance shows linear relative speedup between 128 and 4,096 processor cores, even though the 128-processor configuration has a single cluster and the 4,096-processor configuration requires the coordination of 32 such clusters.

- Tree inflation is the primary reason that speedups begin to depart from linear as more processors are applied. At the beginning of the asynchronous search phase, a large number of processors may need to share a relatively small pool of subproblems, with the result that some processors will evaluate subproblems that would have eventually been pruned prior to evaluation in a run with fewer processors. These subproblems may be subdivided multiple times and persist in the work pool until the final incumbent value is found. Essentially, this form of inefficiency results from a combination of the application not having a strong incumbent-generating heuristic and it being increasingly difficult to approximate a best-first search order as one increases the number of processors. In applications, such as the quadratic assignment problem, for which heuristics can typically generate high-quality initial incumbents early in the solution process, the main impediments to scalability would instead be processor idleness caused by a lack of available subproblems, and inefficiencies in moving subproblems between processors.

Our second set of tests used PEBBL's enumeration feature. We used a realistic application of this feature: since one principle application of the MMA is in a column-generation setting, and it is frequent for column generation algorithms to try to add several dozen columns to master problem at a time, we tested the same set of problems as above, but with `enumCount` = 25. In other words, we reran the same problems, but generated "the" 25 best

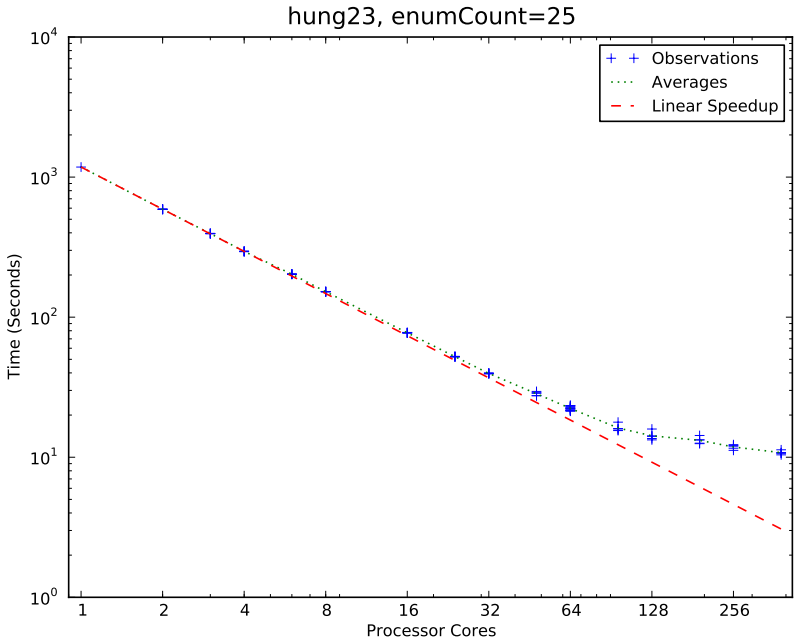

Figure 25: Speedup behavior on problem instance `hung23` with enumeration active.

solutions instead of just one solution ("the" is in quotes because of possible nonuniqueness). The `enumCount` enumeration mode places the largest additional communication burden on the PEBBL parallel layer. Generally, enumerating multiple solutions means exploring a larger search tree. We observed that the size of the tree changed little for the heart disease problems, but increased by roughly 25% for the spam problems.

Figures 25-33 show the speedup results. We tested only $\rho = 1$, the more efficient ramp-up setting from the single-solution tests. Overall, the results are similar to those without enumeration. In general, the heart disease problems' scalability suffers slightly, since the search trees are of similar size but the implementation has more communication overhead, especially during the synchronous ramp-up phase. The results for the spam-detection problems are quite similar to those without enumeration, and in some cases scalability improves slightly due to the larger search tree. Enumeration's extra communication overhead is of less concern for the spam problems, because a similar amount of overhead is "amortized" over more time-consuming subproblems. The size of a solution to the MMA problem is small and bounded by the number of problem features, which is nearly identical in the heart disease and spam problem instance classes. Therefore the extra communication effort required to implement `enumCount` is about the same for all the problems tested. However, the spam detection problem instances have far more observations, so the bounding and separation operations for subproblems take much longer.

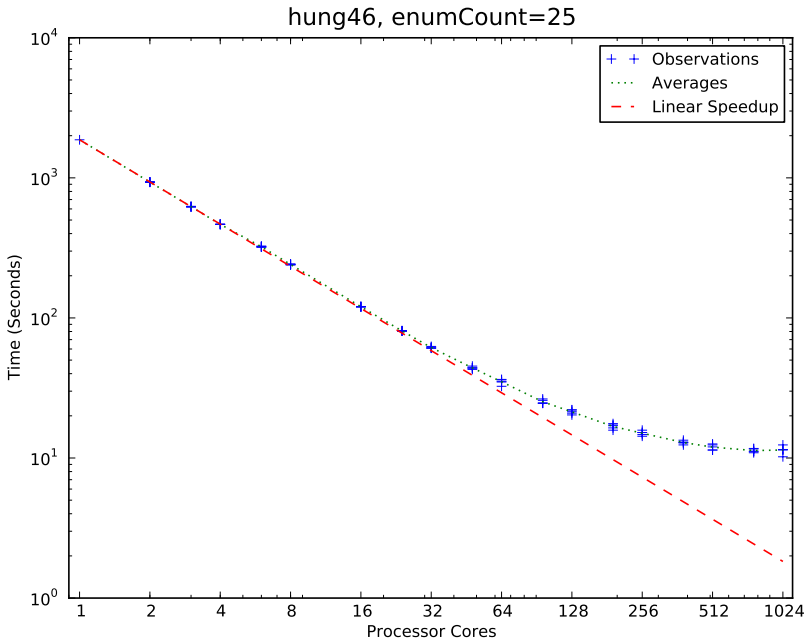

Figure 26: Speedup behavior on problem instance `hung46` with enumeration active.

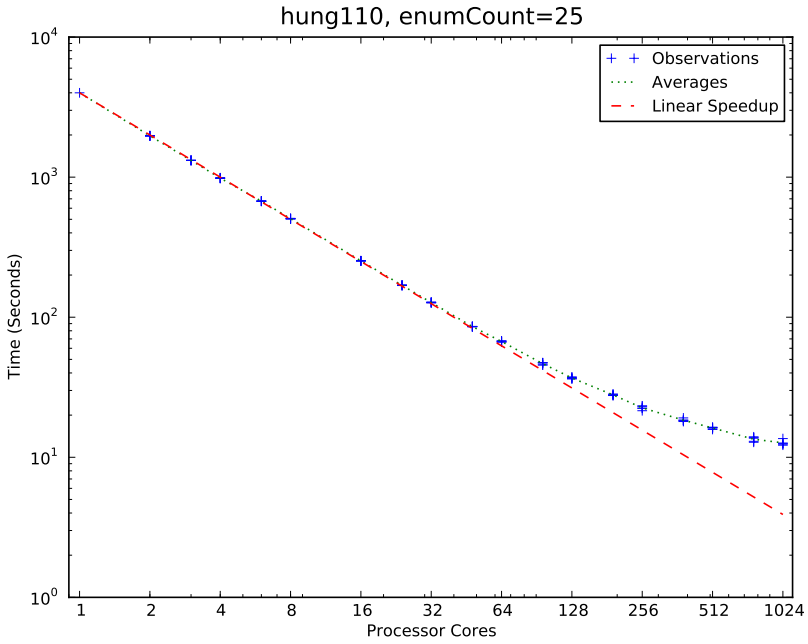

Figure 27: Speedup behavior on problem instance `hung110` with enumeration active.

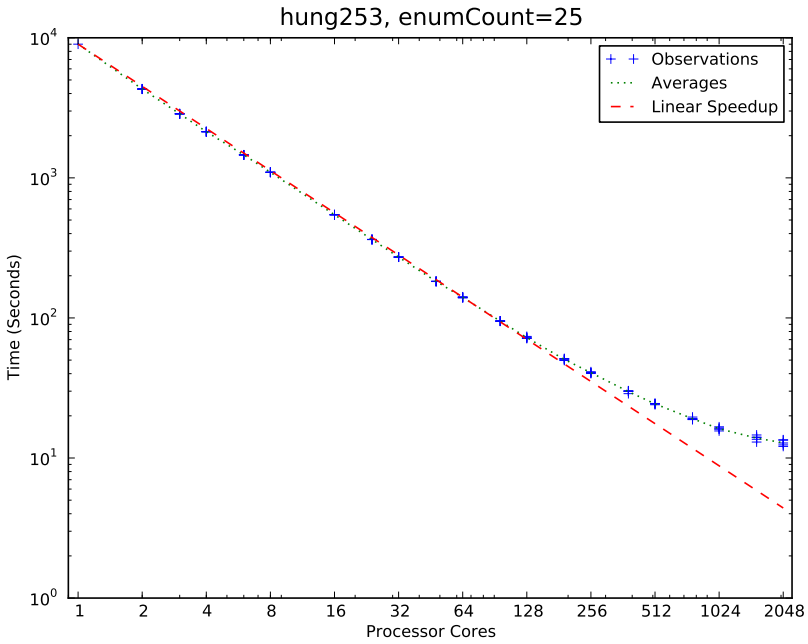

Figure 28: Speedup behavior on problem instance `hung253` with enumeration active.

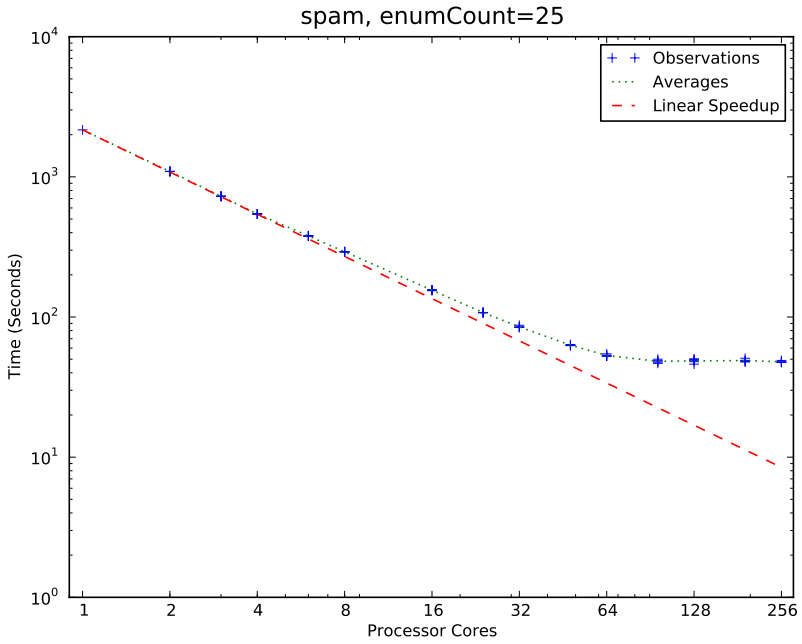

Figure 29: Speedup behavior on problem instance `spam` with enumeration active.

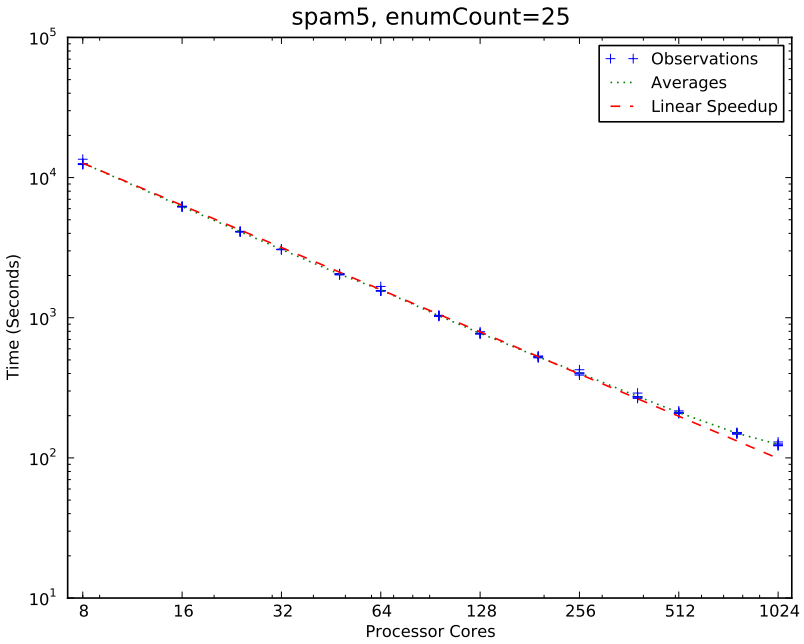

Figure 30: Speedup behavior on problem instance `spam5` with enumeration active.

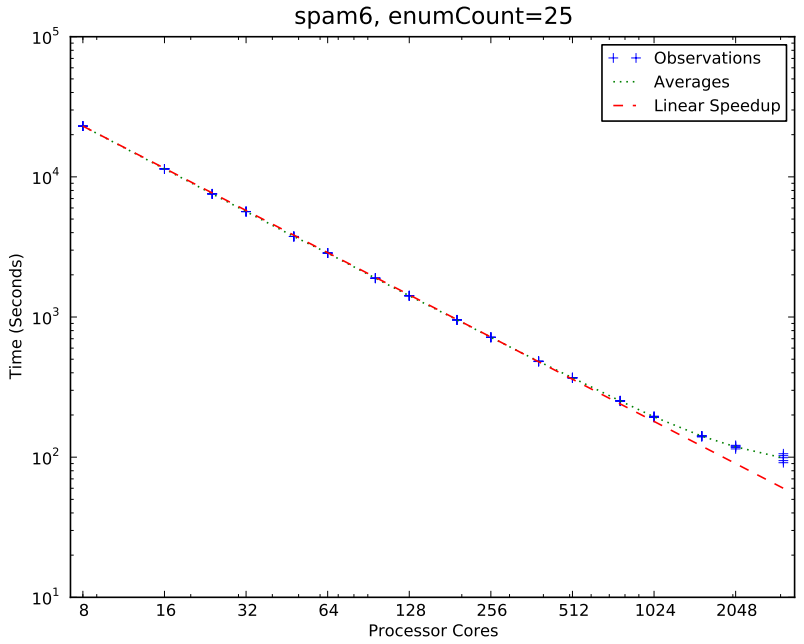

Figure 31: Speedup behavior on problem instance `spam6` with enumeration active.

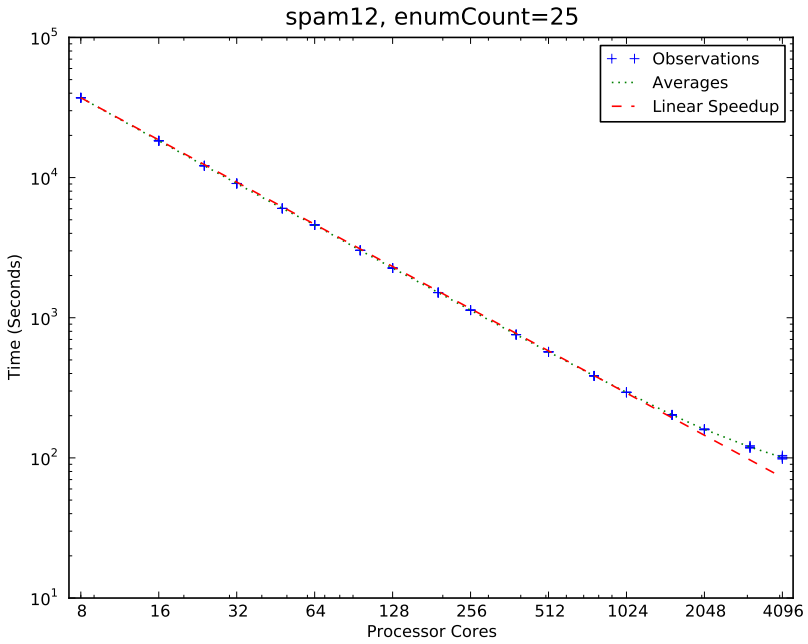

Figure 32: Speedup behavior on problem instance `spam12` with enumeration active.

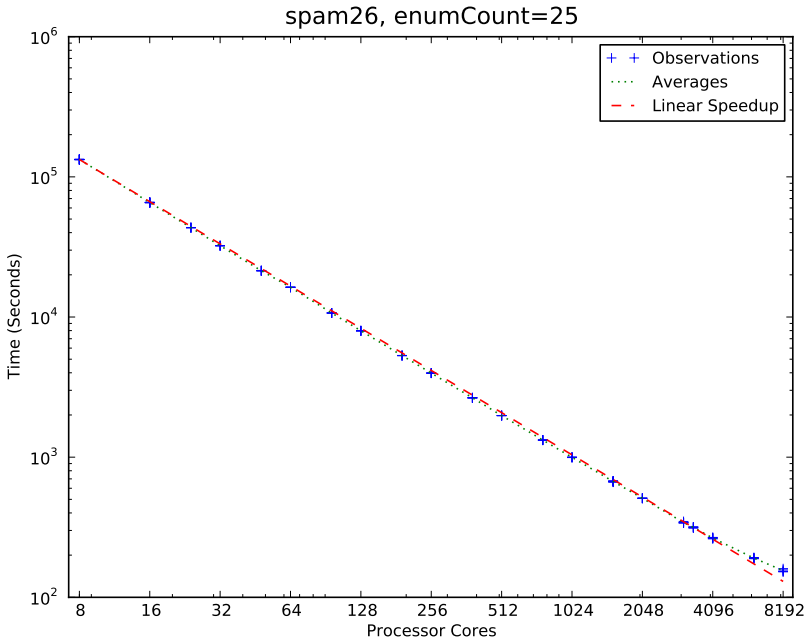

Figure 33: Speedup behavior on problem instance `spam26` with enumeration active.

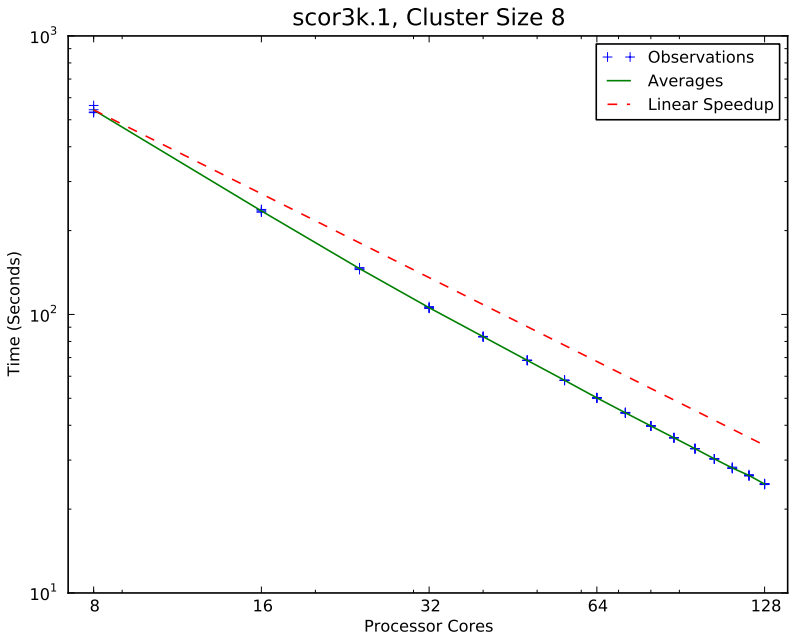

Figure 34: PEBBL speedups on a 3,000-object strongly correlated knapsack problem.

5 Cache-related superlinear speedups

Close scrutiny of the `spam26` results in Figures 15 and 33 reveals a curious phenomenon. For much of the range of the processors tested, relative speedups are slightly *better* than linear. One mechanism through which such a phenomenon can occur, depending on the subproblem pool ordering discipline and the means of generating incumbent solutions, is that using more processors may allow an incumbent solution to be found at a relative earlier point in the search process, shrinking the size of the search tree. However, that is not the case here, as shown in Figure 24. In fact, tree sizes gradually increase with the number of processors, as is more typical.

Instead, the slightly superlinear results are due to processor memory cache behavior. We demonstrate this conclusion by studying an instance of a different problem class in which the effect is far more pronounced. Specifically, Figure 34 shows the speedup behavior, again on the Red Sky system, of the knapsack sample application distributed with PEBBL on a 3,000-object binary knapsack problem in which the object weights and values are strongly correlated. We did not select this application for our main numerical tests because it does not implement a competitive method for solving knapsack problems: its algorithm is equivalent to using the linear programming relaxation of the problem without any cutting planes. However, it does serve as a simple demonstration application of PEBBL that has rather different properties from the MMA problem. In particular, subproblems tend to evaluate extremely quickly for this problem class: the runs shown in Figure 34 all evaluated approximately 200 million subproblems. To avoid congestion at the hubs, we therefore used a much smaller

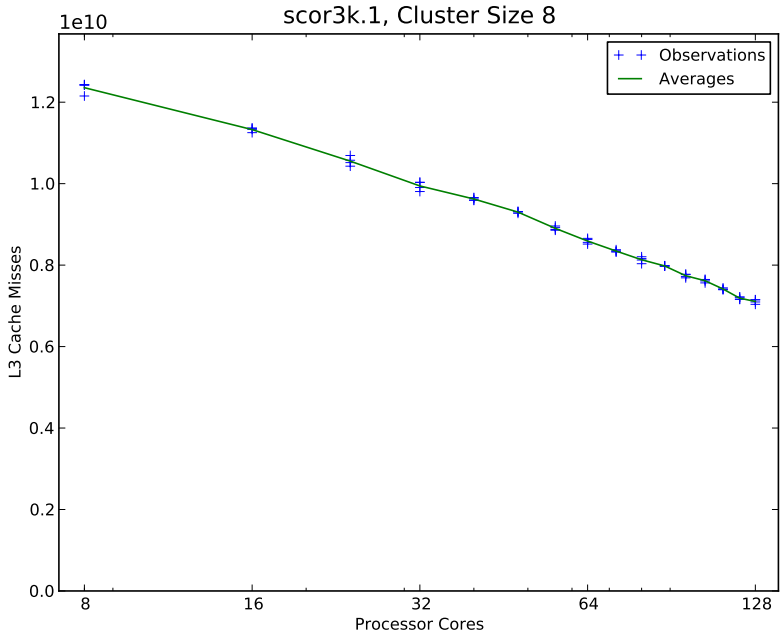

Figure 35: Number of level-3 cache misses, summed across all processors, for the runs shown in Figure 34.

cluster size than in MMA: we selected a cluster size of 8, which matches the number of cores on each Red Sky node, and configured the hubs to be "pure", that is, not to also function as workers. Otherwise, we used PEBBL's default configuration settings, as in Section 4.3. We tried all multiples of 8 processor cores from 8 to 128, solving the problem instance three times for each configuration.

As can be seen in the figure, significant and highly reproducible superlinear speedup occurs. To attempt to explain this phenomenon, we first checked the size of the search tree explored, but it was virtually constant. We then instrumented the executable code using Open|SpeedShop [43], and observed that the total number of machine instructions executed across all processors grew slightly between 8 and 16 processor cores, and then remained essentially flat. Thus, the only possibly explanation for the superlinear speedup is that the average instruction execution time fell as we added more processors. In turn, the only reasonable explanation of this reduction in instruction time is the use of memory cache. Figure 35 shows the total number of level-3 cache misses across all processors, using information also obtained from Open|SpeedShop. Clearly, the larger processor configurations are able to keep a larger fraction of their working data within cache. Even with the higher relative communication overhead arising from the knapsack problem's easier subproblems (as compared to MMA), the communication overhead of PEBBL is small enough that the cache effects dominate and speedups are reproducibly superlinear over a broad range of processor counts.

Although true superlinear speedups are sometimes considered theoretically impossible, it

is important to note that the classical definitions of speedup and efficiency do not consider memory. In practice, as one adds more processors, one is often increasing the memory resources available to the application, including the total amount of each level of cache memory. If the memory usage level and pattern of the application are of the right kind, this extra cache memory may increase efficiency more than interprocessor communication overhead decreases it. Despite its apparent complexity, PEBBL's communication overhead is low enough that this situation can indeed occur.

There are two factors that influence whether such superlinear speedups can happen. One factor is the memory access pattern of the application. On the one hand, if most of the memory references are to a limited set of problem-instance-wide data structures that remain in roughly the same physical memory locations for every subproblem, as occurs in the MMA problem class, most memory references will be cache-resident regardless of how many processors are present. On the other hand, if a significant fraction of memory references are to data stored separately for each subproblem, then cache effects might be noticeable. Additionally, however, the search tree must have the right size: if the storage needed for the tree is too small, then essentially the entire tree will fit in cache regardless of how many processors are present, and cache effects will not be prominent. Conversely, if the tree is so large that only a small fraction of it fits in cache on the range of processors tested, cache effects will be muted. However, if conditions are "just right" and a significantly larger portion of the search tree fits in cache as one increases the processor counts, strong superlinear speedups can result.

For various kinds of algorithms, superlinear speedups attributed to cache effects have been observed since at least the 1990s [9, 50]. More recent examples of such effects are remarked upon in [49], which studies matrix multiplication methods, in [10], which describes matrix factorization methods specifically designed to elicit such effects for particular cache architectures, and in [30], which describes a robot motion planning system. We are not aware of any previous claims of superlinear speedup for branch-and-bound algorithms.

6 Conclusion

A clear conclusion from our results is that with big enough search trees and careful implementation, branch-and-bound algorithms can be highly scalable. We have shown near-linear scaling to over 6,000 processor cores, with every indication that performance could scale even further given sufficiently difficult problem instances. Of course, PEBBL's design is optimized for scalability and lacks features that could be desirable in some applications, such as fault tolerance and determinism. Fault tolerance, the ability to recover from a processor failure in the middle of a run, is highly useful in distributed "grid" computing environments. PEBBL does not provide this functionality, although its checkpointing feature could be a viable substitute in some situations.

Although classical speedup and efficiency can be hard to define and measure in constantly-shifting "grid" environments, we believe that fault-tolerant branch-and-bound implementations should be able to achieve reasonable efficiencies, although there will likely be some

sacrifice in scalability. Such implementations most likely need to be more hierarchical in their organization than PEBBL. We believe extra hierarchy works against scalability and retains vulnerability to processor failures high in the control hierarchy.

The parallel implementations of commercial MIP solvers like CPLEX provide an option for — or in the case of GuRoBi always enforce — deterministic parallel behavior. Deterministic parallelism requires behavior similar to a single-threaded serial program: the set of subproblems examined should be exactly the same in any two complete runs using the same number of processors and the same dataset, exactly the same solution will always be obtained if the solution is nonunique, and if the run is aborted after evaluating a fixed number of subproblems, the same approximate solution will always result. While determinism may have many attractive attributes, we suspect that the performance penalty for requiring determinism in parallel branch-and-bound is substantial and detracts greatly from scalability. Even when solving parallel MIP problems on a handful of processor cores, we have observed that commercial MIP solvers do not appear to exhibit the kind of efficiencies PEBBL displays on thousands of cores.

The march toward parallel computing currently seems inexorable. Desktop computers with 8 cores are now common, and 12-16 cores are typical for workstations. Even mobile telephones now commonly have 4 processing cores. Standard graphics cards, while specialized in their design, already have hundreds of processing elements, and Intel recently introduced a line of Xeon Phi accelerator cards that place roughly 60 general-purpose x86 cores on a single chip and PCI board. Within a decade, we can expect ordinary personal-level computing equipment to provide tens to hundreds of cores, and workstation-level sytems to provide hundreds or thousands. In the 1980's and 1990's, Moore's law worked against widespread deployment of parallel computing technology because the speed of single processors was increasing rapidly. Now, increases in the number of transistors per unit area generally translate into more parallel cores. Branch-and-bound algorithms, correctly implemented, should be able to take advantage of this trend.

A key question is how memory will be organized in future parallel systems. The current trend is toward large, cache-coherent global memories, but the results we have obtained here, especially in Section 5, tend to suggest that fast local memory, including local cache, may be more critical to the performance of applications based on branch and bound or similar search processes.

One clear direction in which our work could be generalized is implementing a general MIP solver, rather than a specialized application like MMA. This is the purpose of the PICO project, of which PEBBL was formerly a part. However, it is harder to produce competitive results in that application domeain, due to the difficulty of replicating the many person-years of work commercial MIP solver implementers have invested in tuning their cutting-plane generators and incumbent heuristics. Nevertheless, there appears to be no fundamental reason scalability results like those shown here could not also be obtained for general MIP. In particular, the technique of synchronous parallel ramp-up seems just as applicable in that setting as in MMA. Within-subproblem parallelism could be exploited near the search tree root through selective strong branching, the related process of initializing pseudocost tables

that help "learn" good branching variables, and generating multiple cutting planes.

References

[1] A. Asuncion and D. J. Newman. UCI machine learning repository, 2007.

[2] D. A. Bader. An improved, randomized algorithm for parallel selection with an experimental study. *J. Parallel Distrib. Comput.*, 64(9):1051–1059, 2004.

[3] D. A. Bader, W. E. Hart, and C. A. Phillips. Parallel algorithm design for branch and bound. In H. J. Greenberg, editor, *Tutorials on Emerging Methodologies and Applications in Operations Research*. Kluwer Academic, 2004.

[4] M. Benaïchouche, V.-D. Cung, S. Dowaji, B. L. Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In *Solving Combinatorial Optimization Problems in Parallel — Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*, pages 201–231, London, UK, 1996. Springer-Verlag.

[5] A. Bendjoudi, N. Melab, and E.-G. Talbi. Fault-tolerant mechanism for hierarchical branch and bound algorithm. In *Workshops Proceedings, 25th IEEE International Parallel and Distributed Processing Symposium, Workshop on Large-Scale Parallel Processing (LSPP)*, pages 1806–1814, Anchorage, Alaska, May 20011.

[6] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538, 1989.

[7] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Linear time bounds for median computations. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 119–124, New York, NY, USA, 1972. ACM.

[8] E. Boros, P. L. Hammer, T. Ibaraki, and A. Kogan. Logical analysis of numerical data. *Math. Programming*, 79(1-3):163–190, 1997.

[9] T. Brewer. A highly scalable system utilizing up to 128 PA-RISC processors. In *COMPCON, Technologies for the Information Superhighway, Digest of Papers*, pages 133–140, 1995.

[10] A. M. Cataldo and R. C. Whaley. Scaling LAPACK panel operations using parallel cache assignment. In *ACM Principles and Practice of Parallel Programming*, 2010.

[11] J. Clausen. Branch and bound algorithms — principles and examples. In A. Migdalas, P. Pardalos, and S. Storøy, editors, *Parallel Computing in Optimization*, volume 7 of *Applied Optimization*, pages 239–267. Kluwer Academic, Dordrecht, 1997.

[12] J. Clausen and M. Perregaard. On the best search strategy in parallel branch-and-bound: best-first search versus lazy depth-first search. *Ann. Oper. Res.*, 90:1–17, 1999.

[13] E. Danna, M. Fenelon, Z. Gu, and R. Wunderling. Generating multiple solutions for mixed integer programming problems. In *Integer Programming and Combinatorial Optimization*, volume 4513 of *Lecture Notes in Computer Science*, pages 280–294. Springer, Berlin, 2007.

[14] M. Daumas and P. Evripidou. Parallel implementations of the selection problem: A case study. *Int. J. Parallel Programming*, 28(1):103–131, 2000.

[15] A. Demiriz, K. P. Bennett, and J. Shawe-Taylor. Linear programming boosting via column generation. *Machine Learning*, 46:225–254, 2002.

[16] D. P. Dobkin, D. Gunopulos, and W. Maass. Computing the maximum bichromatic discrepancy, with applications to computer graphics and machine learning. *J. Comp. Sys. Sci.*, 52(3):453–470, 1996.

[17] J. Eckstein. Control strategies for parallel mixed integer branch and bound. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 41–48, New York, NY, USA, 1994. ACM.

[18] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM J. Optim.*, 4(4):794–814, 1994.

[19] J. Eckstein. Distributed versus centralized storage and control for parallel branch and bound: mixed integer programming on the CM-5. *Comput. Optim. Appl.*, 7(2):199–220, 1997.

[20] J. Eckstein. How much communication does parallel branch and bound need? *INFORMS J. Comput.*, 9(1):15–29, 1997.

[21] J. Eckstein and N. Goldberg. An improved branch-and-bound method for maximum monomial agreement. *INFORMS J. Comput.*, 24(2):328–341, 2012.

[22] J. Eckstein, W. E. Hart, and C. Phillips. Massively parallel mixed-integer programming: Algorithms and applications. In M. A. Heroux, P. Raghavan, , and H. Simon, editors, *Parallel Processing for Scientific Computing*, chapter 17. SIAM, 2006. Based on the 11$^{\text{th}}$ SIAM Conference on Parallel Processing for Scientific Computing.

[23] J. Eckstein, W. E. Hart, and C. A. Phillips. Resource management in a parallel mixed integer programming package. In *Proceedings of the Intel Supercomputer Users Group*, 1997. Online proceedings.

[24] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: an object-oriented framework for parallel branch and bound. In *Inherently parallel algorithms in feasibility and optimization and their applications (Haifa, 2000)*, volume 8 of *Stud. Comput. Math.*, pages 219–265. North-Holland, Amsterdam, 2001.

[25] M. Elf, C. Gutwenger, M. Jünger, and G. Rinaldi. Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions*, pages 157–222, London, UK, 2001. Springer-Verlag.

[26] N. Goldberg and C. Shan. Boosting optimal logical patterns. In *Proceedings of the Seventh SIAM International Conference on Data Mining*, pages 228–236, Minneapolis, 2007. SIAM.

[27] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[28] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Comput.*, 22(6):789–828, 1996.

[29] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, USA, 1985.

[30] J. Ichnowski and R. Alterovitz. Parallel sampling-based motion planning with superlinear speedup. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[31] M. Jünger and S. Thienel. Introduction to ABACUS—a branch-and-cut system. *Oper. Res. Lett.*, 22(2-3):83–95, 1998.

[32] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30:1325–1352, 2000.

[33] G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers: a summary of results. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 453–462, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[34] M. J. Kearns, R. E. Schapire, and L. M. Sellie. Toward efficient agnostic learning. *Machine Learning*, 17(2-3):115–141, 1994.

[35] M. Kim, H. Lee, and J. Lee;. A proportional-share scheduler for multimedia applications. In *IEEE International Conference on Multimedia computing and systems '97*, pages 484–491, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[36] L. Ladányi, T. K. Ralphs, and L. E. Trotter, Jr. Branch, cut, and price: sequential and parallel. In *Computational combinatorial optimization (Schloß Dagstuhl, 2000)*, volume 2241 of *Lecture Notes in Computer Science*, pages 223–260. Springer, Berlin, 2001.

[37] A. Mahanti and C. J. Daniel. A SIMD approach to parallel heuristic search. *Artif. Intel.*, 60:243–282, 1993.

[38] F. Mattern. Algorithms for distributed termination detection. *Distrib. Comput.*, 2:161–175, 1987.

[39] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a mixed INTeger optimizer. *Oper. Res. Lett.*, 15(1):47–58, 1994.

[40] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Math. Programming*, 98(1-3, Ser. B):253–280, 2003. Integer programming (Pittsburgh, PA, 2002).

[41] T. K. Ralphs, L. Ládanyi, and M. J. Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *J. Supercomput.*, 28(2):215–234, 2004.

[42] V. J. Rayward-Smith, S. A. Rush, and G. P. McKeown. Efficiency considerations in the implementation of parallel branch-and-bound. *Ann. Oper. Res.*, 43(1-4):123–145, 1993.

[43] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|speedshop: An open source infrastructure for parallel performance analysis. *Sci. Programming*, 16(2-3):105–121, 2008.

[44] Y. Shinano, K. Harada, and R. Hirabayashi. Control schemes in a generalized utility for parallel branch-and-bound algorithms. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, page 621, Washington, DC, USA, 1997. IEEE Computer Society.

[45] Y. Shinano, M. Higaki, and R. Hirabayashi. A generalized utility for parallel branch and bound algorithms. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 392, Washington, DC, USA, 1995. IEEE Computer Society.

[46] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[47] S. Tschöke and T. Polzer. Portable parallel branch-and-bound library: PPBB-Lib user manual version 2.0. Technical report, Department of Computer Science, University of Paderborn, 1996.

[48] C. A. Waldspurger. *Lottery and stride scheduling: flexibile proportional-share resource management*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1996.

[49] M.-S. Wu, S. Aluru, and R. A. Kendall. Mixed mode matrix multiplication. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[50] X. Wu (editor). *Performance Evaluation, Prediction and Visualization of Parallel Systems, Volume 4 of The Kluwer international series on Asian studies in computer and information science*. Kluwer Academic, Springer, 1999.