Math. Prog. Comp. (2015) 7:429–469
DOI 10.1007/s12532-015-0087-1

FULL LENGTH PAPER

PEBBL: an object-oriented framework for scalable parallel branch and bound

Jonathan Eckstein[1] · **William E. Hart**[2] · **Cynthia A. Phillips**[3]

Received: 18 September 2013 / Accepted: 30 June 2015 / Published online: 23 July 2015

Abstract Parallel Enumeration and Branch-and-Bound Library (PEBBL) is a C++ class library implementing the underlying operations needed to support a wide variety of branch-and-bound algorithms on MPI-based message-passing distributed-memory parallel computing environments. PEBBL can be customized to support application-specific operations, while managing the generic aspects of branch and bound, such as maintaining 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. We describe the basics of PEBBL's architecture, with emphasis on the features most critical to is high scalability, including its flexible two-level load balancing architecture and its support for a synchronously parallel ramp-

This work was performed in part at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94-AL85000. This work was also supported in part by National Science Foundation Grant CCR-9902092.

✉ Jonathan Eckstein
jeckstei@rci.rutgers.edu

William E. Hart
wehart@sandia.gov

Cynthia A. Phillips
caphill@sandia.gov

[1] Department of Management Science and Information Systems and RUTCOR, Rutgers University, 100 Rockafeller Road, Piscataway, NJ 08854, USA

[2] Center for Computing Research, Sandia National Laboratories, Mail Stop 1327, P.O. Box 5800, Albuquerque, NM 87185-1327, USA

[3] Center for Computing Research, Sandia National Laboratories, Mail Stop 1326, P.O. Box 5800, Albuquerque, NM 87185-1326, USA

up phase. We also present an example application: the maximum monomial agreement problem arising from certain machine learning applications. For sufficiently difficult problem instances, we show essentially linear speedup on over 6000 processor cores, demonstrating a new state of the art in scalability for branch-and-bound implementations. We also show how processor cache effects can lead to reproducibly superlinear speedups.

Keywords Branch and bound · Parallel computation

Mathematics Subject Classification 90C57 · 65Y05

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 [9] 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 efficient parallel implementation is immediate and straightforward.

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 range 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 tasks unique to their applications. The need for branch-and-bound frameworks is greater in parallel than in serial, because managing the pool of active search nodes is relatively straightforward in serial. By contrast, parallel branch-and-bound frameworks may require more complex logic for pool management, distributed termination and load balancing.

This paper describes the Parallel Enumeration and Branch-and-Bound Library (PEBBL) software framework. The goal of PEBBL, influenced by the needs of Sandia National Laboratories, is to provide extreme scalability in implementing branch-and-bound methods on distributed-memory computing systems.

For efficiency and portability, we implemented PEBBL in C++, using the MPI message-passing API [49] to communicate between processors. PEBBL may also be used on shared-memory platforms by configuring standard MPI implementations such as MPICH [27] and Open MPI [26] 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.

The most distinctive features of PEBBL are:

– An extremely flexible work distribution and load balancing scheme that achieves unmatched scalability and has only two strata in its processor hierarchy. That is, there are just two possibly overlapping kinds of processors: "workers" and "hubs".

When scaling to large numbers of processors, the hubs interact in a peer-to-peer manner and do not require extra levels of controller processors such as "masters of masters" or "masters of masters of masters".
- Support for application-specific non-tree parallelism during the search ramp-up phase, followed by a "crossover" to using tree-based parallelism.
- Support for enumeration of multiple optimal and near-optimal solutions meeting a variety of configurable criteria.

The rest of this paper is organized as follows: Sect. 2 summarizes PEBBL's history, discusses some more of its innovative contributions, and presents a brief literature review relating it to other work on branch-and-bound computational frameworks. Section 3 then presents key aspects of PEBBL's design. Section 4 then presents a computational study based on the maximum monomial agreement (MMA) problem described in [13,18,25], showing excellent scaling to over 6000 processor cores. Section 5 then presents some computational observervations showing how processor cache behavior can lead to superlinear speedups for some branch-and-bound applications. Our main conclusions from our computational experiments are:

- PEBBL definitively demonstrates scalability of parallel branch and bound over a significantly wider range of processor counts than previously published work.
- When enumerating multiple optimal or near-optimal solutions, PEBBL's scalability is nearly as good as when seeking a single solution.
- Despite its apparent complexity, PEBBL's parallelization strategy adds relatively little overhead to underlying serial branch-and-bound algorithms.

PEBBL is a large, complex software package, so this paper necessarily omits many details. For a more detailed description, the reader should refer to the technical report [22], which is an expanded version of this article, and [23], the current PEBBL user guide. PEBBL is part of A Common Repository for Optimizers (ACRO), a collection of optimization-related software projects maintained by Sandia National Laboratories. PEBBL may be downloaded from http://software.sandia.gov/acro (under the BSD license). The PEBBL source files include 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 1.4.1.

2 Literature review and history

PEBBL began its existence as the "core" layer of the parallel mixed integer programming (MIP) solver Parallel Integer and Combinatorial Optimizer (PICO). However, we separated PICO into two distinct packages to facilitate using its core layer for applications with bounding procedures not involving linear programming—for example, the application described in Sect. 4 below. The core layer, which supported generic parallel branch-and-bound algorithms in an application-independent way, became PEBBL. The remainder of PICO is specific to problems formulated explicitly as MIPs. An early description of PICO [21] includes a description of the PICO core, which evolved into PEBBL. However, PEBBL's design has evolved from this description, and its scalability has improved significantly. Similar but more recent material is included as part of [2], and a somewhat more recent but very condensed description of the internals of PEBBL and PICO is also embedded in [19]. Neither [2] nor [19] contain computational results or describe PEBBL's enumeration capabilities.

Some elements of PEBBL's design can be traced back to ABACUS [24,30,31], a serial C++ framework supporting serial 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 [14–17], a parallel branch-and-bound solver specific both to mixed integer programming and to the Thinking Machines CM-5 parallel computing system of the early 1990's.

PEBBL is neither the first nor the only parallel branch-and-bound implementation framework. Early C-language general parallel branch-and-bound libraries include PUBB [47,48], BoB [3], and PPBB-Lib [50]. MINTO [40] is an earlier, C-language framework specifically addressing MIP problems. SYMPHONY and BCP, both described in [36,41], 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 have been superseded by tools based on CHiPPS/ALPS [42]. Many aspects of CHiPPS/ALPS [42] were influenced by the early development of PICO and PEBBL.

To our knowledge, the most recently reported scalability studies for general-purpose parallel branch-and-bound engines are for BOBPP, a successor to Bob, and ALPS. Manouer et al. [39] use the general BOBPP framework to parallelize Google's OR-Tools constraint solver. Compared to a serial run, they report a speedup of 38.14 on 48 cores (79 % relative efficiency). Xu et al. report computational experience using ALPS for knapsack problems [54]. Based on average total wall-clock time over 26 difficult knapsack problems on a BlueGene system, they report efficiency of 77 % on 2048 processors relative to the same problems running on 64 processors. The largest numbers of processors reported for branch-and-bound-based runs are specific to integer linear programming: Koch et al. [35] solved a single difficult integer program with the SCIP integer programming solver (using CPLEX for linear programming solutions) and MPI on an HLRN-II SGI Altix ICE 8200 Plus supercomputer. They report 79 % efficiency on 7168 cores relative to a base of 4096 cores. Using such a large base number of processors may not give an accurate picture of true efficiency relative to a single processor. In a recent, less formal setting, Shinano [45] reported efficiencies of 76 % solving a particular integer program using paraSCIP [46] with 4095 processors compared to a base of 239 processors on an HLRN-II. But for another integer programming problem, the efficiency for the same processor count and base was only 38 %. The same source also reports attempts to run on 35,000 processors of a Titan Cray XK7 and reported a run time for almost 10,000 processors, but with no scalability results.

In other recent work, the FTH-B&B [4] package focuses on fault-tolerant branch-and-bound mechanisms for grid environments. It includes fault detection through heartbeat communication between parents and children in a multi-level control hierarchy, as well as checkpointing and recovery mechanisms. Experiments in [4] focus on measuring fault-tolerance-related overhead, rather than considering scalability and search efficiency.

3 Software architecture

We now describe PEBBL's software design. Included in this description are a number of original contributions not mentioned in Sect. 1, namely:

– Division of the software package 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 transforming these states; this feature allows one to easily change search "protocols"—see Sect. 3.2
– Variable amounts of subproblem exchange between "hub" and "worker" processors
– Use of "threads" (more properly referred to as coroutines) and a nonpreemptive scheduler module to manage tasks on each individual processor.

Regarding the first item above, the PEBBL software 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 additional methods are required, principally to tell PEBBL how to pack application-specific problem and subproblem data into MPI message buffers, and later unpack them.

Figure 1 shows the conceptual inheritance pattern used by PEBBL. Any parallel PEBBL application constructed through this multiple-inheritance scheme has the full capabilities of the parallel layer, including a highly configurable spectrum of parallel work distribution and load balancing strategies.

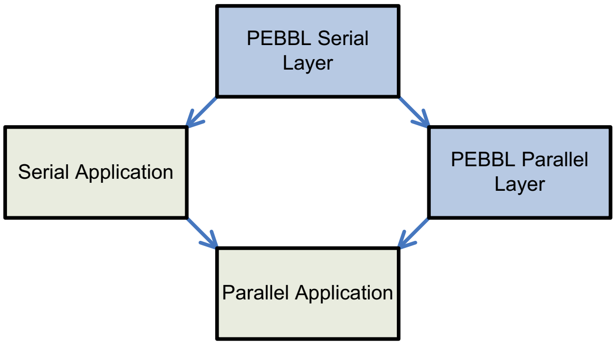

Fig. 1 The conceptual relationships of PEBBL's serial layer, the parallel layer, a serial application, and the corresponding parallel application

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 it 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 it contains methods that perform generic operations on subproblems. This basic organization is borrowed from ABACUS [24,30,31], but PEBBL's design is more general, since there is no assumption that linear programming or cutting planes are involved.

The remainder of this section refers to numerous parameters controlling the details of the branch-and-bound search. All such parameters have default values so users can begin using PEBBL without having to explicitly specify their values. The user may then tune specific parameters based on the runtime behavior of their application.

3.1 Subproblem states, subproblem transition methods, and solution generation

Every PEBBL subproblem stores a state indicator that progresses through some subset of six states called `boundable`, `beingBounded`, `bounded`, `beingSeparated`, `separated`, and `dead`.

The `branchSub` class has three abstract virtual methods which are responsible for implementing subproblem state transitions: `boundComputation`, `splitComputation`, and `makeChild`. A serial PEBBL application is primarily defined by the instantiations of these three methods in the application subproblem class.

Figure 2 illustrates the subproblem state operator methods and the possible subproblem state transitions. The `boundComputation` method advances subproblems from the initial `boundable` state through to the `bounded` state, while the `splitComputation` method advances subproblems from the `bounded` state to the `separated` state. Each application of the `makeChild` method extracts a child subproblem.

Once all of a subproblem's children have been created, it becomes `dead`. Any subproblem operator can also set a subproblem's state to `dead` if it determines that the subproblem represents no relevant solutions. The `beingBounded` state is included to allow for incremental calculations that might compute progressively tighter bounds

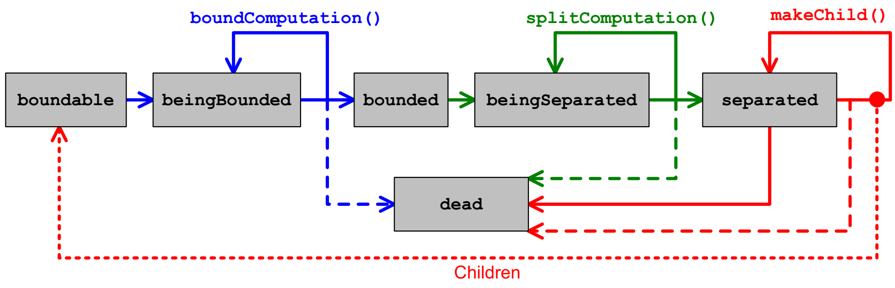

Fig. 2 PEBBL subproblem state transition diagram

for a subproblem, or for lengthy bound calculations that one might want to temporarily suspend. The `beingSeparated` state is provided for similar reasons.

Each subproblem also has a `bound` data member which may be updated at any time by any of the three state-transition methods, although it is typically set by `boundComputation`. Furthermore, the `branchSub` class has an `incumbent-Heuristic` method that is called once a subproblem becomes bounded. This method is intended to generate feasible solutions, perhaps using information generated during bounding. However, any of the subproblem operators is free to generate possible new incumbent solutions at any point in the life of a subproblem. The `branchSub` class also contains methods for identifying "terminal" subproblems, and extracting solutions from them. Terminal subproblems are those for which the computed bound is exact and a matching feasible solution is readily available: in integer programming, for example, a subproblem is terminal if solving its linear programming relaxation returns an integral solution. A terminal subproblem does not require further exploration and becomes `dead` once PEBBL extracts a solution matching its bound value (unless PEBBL is enumerating multiple solutions—see Sect. 3.6 below).

PEBBL also provides a number of classes for representing problem solutions. If a specialized solution representation is needed, the user may derive one from the PEBBL-provided base class `solution`.

3.2 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 variant of branch-and-bound being applied. Essentially, the framework executes a loop in which it extracts a subproblem S from the pool and passes it to the handler, which may create children of S and insert them into the pool. If subproblem S is not `dead` after processing by the handler, the framework returns it to the pool. Figure 3 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 three kinds of pools:

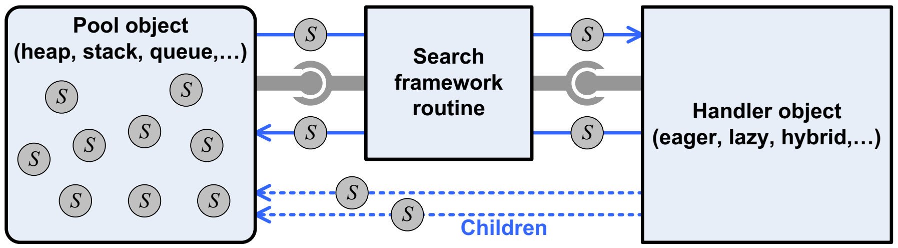

Fig. 3 The search framework, pool, and handler. Each S indicates a subproblem

– A heap sorted by subproblem bound, which results in a best-first search order. This pool also has a “diving” option to give priority to an application-defined “integrality measure” (in general, some measure of closeness to being a terminal node) until an incumbent solution is found, and then revert to standard best-bound ordering.
– A FIFO queue, which results in a breadth-first search order.
– A stack, which results in a depth-first search order.

For particular applications, users may implement customized pools that specify other search orders.

In general, subproblems in the pool may in be in any mix of states. The handler, the other component “plugged into” the search framework, implements a “search protocol” specifying how subproblems are advanced through the state diagram, thus controlling the mix of states present in the pool. Three handlers are currently available: “eager”, “lazy”, and “hybrid”.

The lazy handler implements lazy search, as defined in [10]. In lazy search, the handler removes a subproblem from the pool and computes its bound. If the subproblem cannot be fathomed, the handler extracts its children and places them back in the pool without computing their bounds. This variant of branch and bound is typical of MIP solvers. The eager handler instead implements an eager search protocol [10]. In eager search, the handler picks a subproblem from the pool, immediately separates it, and then extracts all its children and calculates their bounds. Children whose bounds do not cause them to be fathomed are returned to the pool. This type of search is more typically used in situations in which the bound may be calculated quickly.

PEBBL also contains a third handler, called the hybrid handler. This handler implements a strategy that is somewhere between eager and lazy search, and is perhaps the most simple and natural given PEBBL’s concept of subproblem states. Upon removing a subproblem from the pool, the hybrid handler performs one application of the appropriate method to advance it in the state diagram. It then places the subproblem back in the pool if it is not `dead`, along with any children generated in the process. With the hybrid handler, the pool contains subproblems in an arbitrary mix of states, whereas the lazy and eager handler keep the entire pool in the `boundable` or `bounded` state, respectively, unless the application uses the `beingBounded` or `beingSeparated` states to suspend bounding or separation operations.

3.3 Parallelism: tokens and work distribution within a processor cluster

PEBBL’s parallel layer organizes processors similarly to the later versions of CMMIP [14,16]. Processors are organized into “clusters”, each consisting of one “hub” processor that controls one one or more “worker” processors. Through run-time parameters, the user may control the number of processors per cluster, and whether hub processors are “pure” hubs or simultaneously function as hubs and workers. It is possible for a cluster, or even all clusters, to consist of only a single processor.

Although there is a limit to the number of processors that may function efficiently within a centrally controlled cluster, PEBBL’s philosophy is to make this limit as large as possible. To this end, its design used two basic principles: first, a hub should be able to “guide” rather than “micromanage” its workers, without having to interpose itself into every worker subproblem-processing decision. Second, the communication and memory resources of the hub should not be wasted by storing and transmitting subproblem information irrelevant to the hub’s main purpose of scheduling and distributing work.

To allow for “guidance” rather than “micromanagement,” workers in a PEBBL cluster are generally not pure “slaves.” Each worker maintains its own pool of active subproblems. Depending on various run-time parameters, the pool might be small (e.g. a single subproblem). Workers use essentially the same search handler objects present in the serial layer, but in parallel these handlers also have the ability to “release” subproblems from the worker. The decision whether to release a subproblem is usually taken when it is created, except when performing eager search, in which case the decision is taken immediately after the subproblem is bounded. Released subproblems do not return to the local pool: instead, the worker cedes control over these subproblems to the hub. Eventually, the hub will send control of the subproblem either back to the worker or to another worker.

When a processor is both a hub and a worker, it maintains two distinct pools of subproblems, one under control of the hub thread and one under control of the worker thread. The two coresident threads communicate in much the same way as a worker and hub located on different processors, but through local memory operations rather than MPI messages.

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

A worker’s decision to release a subproblem is randomized, with the probability of release controlled by run-time parameters and the current distribution of work among processors. If PEBBL is configured so that the release probability is always 100 %, then control of every subproblem returns to the hub at some state in its lifecycle. In this case, the hub and its workers function like a classic “master-slave” system. When the release probability is lower, the hub and its workers are less tightly coupled. The release probability can vary between parameter-determined bounds depending on the fraction of the estimated total work controlled by the worker processor. To promote even work distribution early in a run, the release probability is temporarily set to 100 % for the first s subproblems each worker encounters after the initial synchronous ramp-up phase (see Sect. 3.7 below), where s is a run-time parameter.

As mentioned above, the second major principle in PEBBL’s intracluster architecture is avoiding passing unnecessary information through the hub: when a subproblem is released, only a small portion of its data, called a “token” [15,43], is actually sent to the hub. A token consists of the information needed to identify a subproblem, locate it in memory, and schedule it for execution. On a 64-bit processor, a token occupies 80 bytes of storage, which is much less than typically required to store the full data for a subproblem in most applications. Since the hub receives only tokens from its workers, these space savings translate into reduced storage requirements and communication load at the hub. In certain cases, PEBBL can gain further efficiency by using a single token to refer to several sibling subproblems.

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 and dispatches subproblem tokens to workers it deems "deserving". The notion of "deserving" can take into account both subproblem quantity and quality (as measured by the distance between the subproblem bound and the incumbent value).

When the hub dispatches a subproblem token t to a worker w^*, the resulting message might 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 [14–16,43]. To save communication resources, the hub may pack multiple dispatch operations into the same MPI message.

The hub periodically broadcasts overall workload information to its workers so they know the approximate relation of their own workloads to those of other workers. This information allows each worker to adjust its probability of releasing subproblems. PEBBL also has a second mechanism, called "rebalancing", to help maintain the user's desired balance between hub and worker control of subproblems (especially near the end of a run). During rebalancing, workers can send blocks of subproblem tokens to the hub outside of the usual handler-release cycle.

The subproblem release probabilities and rebalancing operations at the workers, along with the calculation of when workers are "deserving", are calibrated to (1) keep the fraction of subproblems in the cluster controlled by the hub close to the run-time parameter `hubLoadFac` and (2) keep 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 $1 - h$ lower than for other worker processors.

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" that it tries to use to correct imbalances as they occur. A spectrum of settings between these two extremes is 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.

3.4 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. Even if PEBBL is configured to maximize the conservation of hub resources, 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. At a certain point, adding more processors to a single cluster will not improve performance. To take advantage of additional processors, PEBBL therefore provides the ability to partition the overall processor set into multiple clusters.

PEBBL's method for distributing work between clusters resembles CMMIP's [14, 16]: there are two mechanisms for transferring work between clusters, "scattering" and "load balancing". Scattering occurs when workers release subproblems. If there are multiple clusters and a worker has decided to release a subproblem, then the worker makes a random decision, controlled by some additional parameters and workload-state information, as to whether the subproblems should be released to the worker's own hub or to the hub of a randomly chosen cluster. At the beginning of a run, PEBBL promotes even work distribution by forcing release of a worker's first s subproblems (using the same notation as in the previous subsection) to random clusters.

To supplement scattering, PEBBL also uses a form of "rendezvous" load balancing similar to CMMIP [14,16]. Earlier related work [32,37] describes synchronous application of the same basic idea 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. This information is also critical to termination detection in the multi-hub case.

The load balancing mechanism defines its estimated "workload" at a cluster c at time t to be

$$L(c,t) = \sum_{P \in C(c,t)} |\overline{z}(c,t) - z(P,c,t)|^{u}. \tag{1}$$

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; this is typically the incumbent value, but it may be different if PEBBL's enumeration feature is active (see Sect. 3.6). The exponent u is either 0, 1, or 2, at the discretion of the user. If $u = 0$, only the number of subproblems in the cluster matters. Values of $u = 1$ or $u = 2$ give progressively higher "weight" to subproblems farther from the incumbent. The default value of u is 1. Using a technique based on binomial coefficients, PEBBL is able to recalculate the estimate (1) in constant time whenever the fathoming value $\overline{z}(c,t)$ changes, individual subproblems are added to or deleted from $C(c,t)$, or individual $z(P,c,t)$ values change.

The rendezvous load balancing mechanism organizes the hub processors into a balanced tree. This tree provides a mechanism for bounding the communication load on each individual hub processor; it is not a hierarchy of control, because all the hubs in the system are essentially peers. 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 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 fathoming values, then it immediately repeats itself.

After each successful survey sweep, every hub determines whether its cluster should be a potential donor of work, a potential receiver of work, or (typically) neither. Donors are clusters whose workloads are above the average workload by some parameter-specified factor, while receivers must have loads below average by another parameter-specified factor. The balance sweep operation then counts the total number of donors d and receivers r. It assigns to each donor a unique number in the range $0, \ldots, d-1$, and to each receiver a unique number in the range $0, \ldots, r-1$. The balance sweep is a form of parallel prefix operation [5], involving a single round of messages passing up the tree, followed by a single pass down. At the end of the balance sweep, all hub processors also know the values of d and r. After the balance sweep, 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 [28, Sect. 6.3] or [14,16] 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. During a load-balancing cycle, from the start of a successful survey sweep through any corresponding work exchanges between hubs, the number of load-balancing-related messages passing through any given hub processor is bounded above by the constant $2(b + 1) + 2(b + 1) + 6 = 4b + 10$, and the cumulative message latency associated with propagating this information is $O(\log_b H) = O(\log H)$, where H is the number of clusters. This design ensures the scalability of PEBBL's load balancing mechanism.

The frequency of survey and balance 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 Sect. 3.8 for a discussion of the termination check procedure.

Finally, we note that interprocessor load balancing mechanisms are sometimes classified as either "work stealing" initiated by the receiver or "work sharing" initiated by the donor. PEBBL's rendezvous method is neither. Instead, donors and receivers efficiently locate one another on an equal, peer-to-peer basis.

3.5 Thread and scheduler architecture

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. On each processor, PEBBL creates some subset of the following threads:

– Two "compute threads" that handle the fundamental computations: a "worker thread" for processing subproblems, and an optional "incumbent search thread" dedicated to running incumbent heuristics.
– Two threads associated with work distribution and load balancing on hub processors: a "hub thread" that handles the main hub functions and a "load balancing/termination thread" for handling intercluster load balancing (see Sect. 3.4 above) and termination detection (see Sect. 3.8 below).
– Three worker-based threads to handle communication between workers and from hubs to workers
– 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, and two optional threads to handle enumeration of multiple solutions (see Sect. 3.6 below).

Furthermore, applications derived from PEBBL have the ability to incorporate their own additional, application-specific threads into PEBBL's multitasking framework. For example, the PICO MIP solver creates an additional thread to manage communication of "pseudocost" branching quality information between processors.

Except for the compute threads, all of PEBBL's threads are "message triggered": they only run when specific kinds of messages are received. The compute threads run whenever they have work available and no arriving messages need to be serviced. When both kinds of compute threads are active, PEBBL manages compute-time allocations between them through a variant of stride scheduling [34,51], allowing for an intelligent allocation of CPU resources to each thread; see [20,21] for details. The split of compute resources between the two threads is determined by a combination or run-time parameters and the current gap between the value of the incumbment solution and the best currently known global bound on the optimal solution value.

The incumbent broadcast thread manages asynchronous tree-based broadcasts of new incumbent values, with special features to handle "collisions" between trees of potential new incumbent messages spreading from different initiating processors. Essentially, the broadcast tree will automatically "wither" and cease propagating messages wherever it encounters a processor with a better incumbent value, or the same incumbent value but a lower-numbered initiating processor. This scheme ensures that all processors eventually agree on the incumbent value and on which processor the incumbent resides.

3.6 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.

PEBBL can also be configured to enumerate multiple solutions. Such enumeration is controlled by four further parameters, as follows:

`enumAbsTol` = a: retain solutions within a units of optimal.
`enumRelTol` = r: retain solutions within a multiplicative factor r of optimal.
`enumCutoff` = c: retain 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 just the usual single incumbent solution, 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 enumeration parameters is to alter the criteria PEBBL uses to fathom subproblems and the behavior of the internal method that signals a potentially new feasible solution. When using just the `enumAbsTol` or `enumRelTol` parameters, fathoming only occurs when a subproblem 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 the repository already contains `enumCount` solutions and a new solution enters, one of the repository's worst solutions is deleted.

For enumeration to work properly, the application classes need two capabilities: "branch-through" and solution duplicate detection. Branch-through refers to branching on terminal subproblem. Without enumeration, PEBBL will never apply the `splitComputation` method to a terminal subproblem: by definition, one has identified an optimal solution in the subproblem's region of the search space, so ordinarily there would be no need to explore this region further. This may not be the case when enumerating, so the application must be prepared to split terminal subproblems. In some applications, splitting terminal subproblems may require a completely different branching procedure: in integer programming, for example, branching ordinarily uses variables with fractional values in the linear programming relaxation solution, but a terminal subproblem will have no such variables.

The second capability normally needed to support enumeration is duplicate solution detection, which is necessary to prevent applications that generate solutions heuristically from accumulating multiple identical solutions in the repository. To support PEBBL's detection of duplicate solutions, any solution representation used by a PEBBL application must implement a solution comparision method and a solution hash value function. Equivalent solutions must have the same hash value. The PEBBL-supplied solution classes already have these capabilities and PEBBL also supplies tools to simplify the construction of such methods for customized solution representation classes.

PEBBL currently lacks any gradated notion of solution distance or "diversity." It has only a boolean sense of solutions being "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 a non-boolean sense. For an example of this kind of technique for MIP problems, see [11]. Unlike PEBBL's enumeration scheme, however, the technique described in [11] does not guarantee complete enumeration of specific sets of solutions; such full solution enumeration appears to be a unique feature of PEBBL.

When running in parallel, PEBBL promotes scalability by partitioning the repository 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. New feasible solutions are sent to their owning processor, where they are checked for duplication before entering the 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. However, if `enumCount` is set, then the implementation becomes more complicated. For proper pruning, one would like to have an estimate of the `enumCount`th-best solution in the union of the repository segments of all processors, which we call the "cutoff solution". In order for each processor to maintain a valid estimate of the cutoff solution, PEBBL periodically passes messages up and down a balanced tree similar to that used for load balancing, but consisting of all processors, rather than just hubs. These messages contain sorted lists of solution identifiers (objective values, owning processor and serial number), that are merged as messages converge up the tree. The root forms an estimate of the cutoff solution that is then broadcast down the tree.

3.7 Synchronous ramp-up support

PEBBL provides support for specialized, synchronous ramp-up procedures at the beginning of the search process. PEBBL's main approach to parallelism is to evaluate multiple nodes of the search tree simultaneously. Early in the search process, however, the pool of search nodes is still small, and particular applications may have different opportunities for parallelism that provide more concurrency. Therefore, 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 `boundComputation`, `splitComputation`, `makeChild`, or the incumbent heuristic. Because different processors might find different incumbents—for example by using randomized procedures with different random-number seeds—PEBBL provides special methods to sychronize the incumbent value during ramp-up. Without such synchronization, there are various mechanisms by which the ramp-up phase may deadlock.

A virtual method controls the termination of the ramp-up phase: Once this method signals the end of ramp-up, the worker processors partition the leaf nodes of the current search tree as equally as possible on both the intercluster and intracluster level. This partition operation can be performed without any communication, since all processors have copies of the same leaf nodes in local memory; it is simply a matter of each processor deleting the subproblems it does not “own”. After partitioning the active subproblems, PEBBL enters its normal asychronous search mode.

PEBBL’s default implementation of the crossover trigger method senses whether the tree has at least αW nodes, where α is a run-time parameter defaulting to 1, and W is the number of worker processors. Ideally, however, it is best to also consider whether the parallelism available in the active nodes of the search tree appears to exceed any alternative source of parallelism available to the application. Such a test is necessarily application-dependent and must be implemented by the user overriding the default crossover-triggering method.

3.8 Startup and termination

In addition to the ramp-phase and the main asynchronous search phase, a PEBBL run has two additional phases: an initial problem read-in operation at the very beginning, and a solution and statistics output phase at the end.

The read-in stage is straightforward: processor 0 reads the problem instance data and broadcasts it to all other processors. 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 processor 0. 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. Finally, some synchronous MPI reduction operations gather performance statistics for output by the processor 0.

A critical aspect of the implementation is detecting the end of the asynchronous search phase, so that the solution output phase can commence. Detecting termination is simple for parallel programs that are organized on a strictly “master-slave” or hierarchical principle. PEBBL has a more complicated messaging pattern with multiple asychronous processes, introducing 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 quiescence, PEBBL uses a method derived from [38], but adapted and specialized to its particular processor organization and messaging pattern. 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). These counts are aggregated at the hub processors, and (if there is more than one hub) through the message sweeps of 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 “pseudoquiescence”. Pseudoquiescense is necessary for search termination, but it is not sufficient because the measurements making up the various sums involved are typically not taken at exactly the same time. For efficiency reasons relating to PEBBL’s specific messaging pattern, especially if enumeration is active, detection of pseudoquiescence is a two-stage process. The first detection level involves only messages relating to work distribution, but not incumbent or repository management, and it is easily triggered at processor 0 as part of the ordinary process of load balancing. If the first level of pseudoquiescence is detected, an additional “quiescence check” pattern of messages confirms whether the total count of all messages sent and received appears to balance. If this check confirms that pseudoquiescence has indeed occurred, PEBBL proceeds to a second check, the “termination check”.

Suppose that pseudoquiescence has occurred, and the total number of messages sent and received is m. It is shown in [38] that to confirm that true quiescence 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. PEBBL uses an additional message sweep to confirm whether this is the case.

3.9 Checkpointing

PEBBL has a checkpointing feature that allows partial runs of branch-and-bound search to be resumed at a later time. This feature can be useful when a system failure or expiration of allotted time stops a PEBBL run before its natural completion. Note that some MPI implementations, for example Open MPI [26], now provide their own transparent checkpointing features, which could be used instead. However, PEBBL’s application checkpointing feature has some capabilities that transparent checkpointing does not, such as the ability to restart a run with different parameter settings or even a different number of processors.

PEBBL’s checkpointing feature is integrated into its termination-detection mechanism. A run-time parameter can be used to specify that PEBBL writes a checkpoint approximately every t minutes after the synchronous ramp-up phase. Once PEBBL detects that a checkpoint is due to be written, it signals all processors through the same process used to signal termination. Upon receiving the checkpoint signal, workers stop processing subproblems, and PEBBL waits until it has confirmed that there are no messages in transit, using the same procedure it employs for termination detection. Once this condition has been confirmed, each processor writes a binary checkpoint file. This file typically consists of the subproblems and tokens in the processor’s memory. Writing the file reuses the methods for constructing work-distribution messages, so no additional customization of PEBBL is needed to support checkpointing. However, PEBBL also provides virtual methods that can be customized to include additional application-specific internal state information in checkpoints.

PEBBL provides two methods, “restart” and “reconfigure”, that support 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 reads a single checkpoint file, which is 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. In this case, 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 it is potentially much slower because it requires a serial pass through all the checkpoint data.

4 Application to maximum monomial agreement

4.1 The MMA problem and algorithm

To illustrate PEBBL’s performance and capabilities, we now describe its application to the maximum monomial agreement (MMA) problem. Eckstein and Goldberg [18] describe this problem and an efficient serial branch-and-bound method for solving it. An earlier, less efficient algorithm is described in [25], and a slightly more general formulation of the same problem class may be found in [13]. The algorithm described in [18] uses a combinatorial bound not based on linear programming, and it significantly outperforms applying 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 [18] 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 [18].

The goal of MMA is to find a logical pattern that “best fits” a set of weighted, binary-encoded observations divided into two classes, in the sense that it maximizes the difference between the weight of matching observations from one class and the weight of matching observations from the other class. This problem arises as a natural subproblem in various machine learning applications. Each MMA instance consists of a 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 $\Omega^+ \subset \{1, \dots, M\}$ and “negative” observations $\Omega^- = \{1, \dots, M\} \backslash \Omega^+$. 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 $i = 1, \dots, M$.

A “monomial” on $\{0, 1\}^N$ is a logical conjunction of features and their complements, that is, a function $m_{J,C} : \{0, 1\}^N \to \{0, 1\}$ of the form

$$m_{J,C}(x) = \prod_{j \in J} x_j \prod_{c \in C} (1 - x_c), \tag{2}$$

where J and C are (usually disjoint) subsets of $\{1, \dots, N\}$. 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 $m_{J,C}$ as

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

Define the weight of a set of observations $S \subseteq \{1, \dots, M\}$ 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, Ω^+) with 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 $J, C \subseteq \{1, \dots, N\}$ solving

$$\begin{array}{ll} \max & \left| w\left(c_{J,C} \cap \Omega^+\right) - w\left(c_{J,C} \cap \Omega^-\right) \right| \\ \text{s.t.} & J, C \subseteq \{1, \dots, N\}. \end{array} \tag{3}$$

When the problem dimension N is part of the input, [18] proves that this problem formulation is $\mathscr{NP}$-hard, using techniques derived from [33].

The main ingredients in any branch-and-bound algorithm are a subproblem description, a bounding function, and a branching rule. For MMA, each possible subproblem is described by some partition (J, C, E, F) of $\{1, \dots, N\}$. 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. $F = \{1, \dots, N\} \backslash (J \cup C \cup E)$ is the set of "free", undetermined features. The root of the branch-and-bound tree is the subproblem $(J, C, E, F) = (\emptyset, \emptyset, \emptyset, \{1, \dots, N\})$. Subproblems with $F = \emptyset$ correspond to only one possible monomial and cannot be subdivided.

The details of the bound $b(J, C, E, F)$ may be found in [18] and are not necessary for the discussion here. It involves equivalence classes of observations induced by the set E and has complexity $\mathrm{O}(MN)$.

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 [18], 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. A byproduct of this procedure is the following potentially tighter "lookahead" bound on the current subproblem objective:

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

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; we use the straightforward strategy of [18], which is simply to use (J, C) as a trial feasible solution whenever processing a subproblem (J, C, E, F).

4.2 Parallel implementation in PEBBL

We extended the serial PEBBL implementation tested in [18] 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. Beyond this basic task, there are three additional, optional implemention steps which may be taken to improve parallel performance when converting a serial PEBBL application to a parallel one:

1. Implementing a synchronous ramp-up procedure, if applicable.
2. Creating an enhanced incumbent heuristic that can run semi-independently from the search process as part of the incumbent heuristic thread.
3. Implementing interprocessor communication for any application-specific state information beyond what can be included in the initial problem instance broadcast or stored within individual subproblems.

Steps 2 and 3 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 relevant to step 3. However, because of the strong branching procedure, the MMA algorithm does have a secondary source of parallelism that lends itself naturally to PEBBL's synchronous ramp-up phase.

The time to process each subproblem (J, C, E, F), especially near the root of the branch-and-bound tree, tends to be dominated by the strong branching procedure's 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 $j \in F$. These calculations are independent and readily parallelizable. While all other calculations are performed redundantly, the ramp-up-phase version of the separation procedure divides the $|F|$ undecided features as evenly as possible between the available processors. Then each processor computes the triple of potential child bounds for each feature j that it has been allocated. Each processor rounds the elements of these triples, sorts the elements within each triple in descending order, and determines the lexicographically smallest triple. 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).

Overriding PEBBL's default implementation of the method that senses the end of ramp-up, we terminate the synchronous ramp-up phase when the size of the subproblem pool becomes comparable to the number of features, in the sense that $|\mathscr{P}| > \rho N$, where $\mathscr{P}$ 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".

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 2816 compute nodes, each with two quad-core Intel Xeon X5570 processors sharing 48 GB of RAM. Red Sky's compute nodes are connected by an Infiniband interconnect arranged as a torroidal three-dimensional mesh, with a 10 GB/s 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. We compiled 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 allocated ν compute nodes behaves as an MPI job with 8ν parallel "processors". By using shared memory

Table 1 Performance of PEBBL on MMA instance `hung23` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	979.5			71,296		979.5			71,296	
2	492.0	2.0	1.00	71,185	−0.2	491.4	2.0	1.00	71,030	−0.4
3	328.6	3.0	0.99	71,046	−0.4	328.0	3.0	1.00	70,738	−0.8
4	245.0	4.0	1.00	70,872	−0.6	244.8	4.0	1.00	70,933	−0.5
6	168.3	5.8	0.97	70,891	−0.6	168.6	5.8	0.97	71,010	−0.4
8	126.8	7.7	0.97	70,954	−0.5	126.7	7.7	0.97	71,012	−0.4
16	63.2	15.5	0.97	70,591	−1.0	63.0	15.5	0.97	70,778	−0.7
24	42.6	23.0	0.96	70,842	−0.6	42.1	23.3	0.97	70,797	−0.7
32	32.2	30.4	0.95	70,670	−0.9	31.9	30.7	0.96	70,896	−0.6
48	22.0	44.6	0.93	70,892	−0.6	21.5	45.6	0.95	71,262	−0.0
64	17.2	57.0	0.89	71,125	−0.2	16.6	59.1	0.92	71,155	−0.2
96	12.2	80.3	0.84	71,698	+0.6	11.4	86.2	0.90	71,419	+0.2
128	9.7	100.6	0.79	73,110	+2.5	8.9	109.6	0.86	71,548	+0.4
192	8.5	115.8	0.60	77,412	+8.6	6.9	142.8	0.74	71,741	+0.6
256	7.3	133.8	0.52	84,497	+18.5	6.1	160.0	0.63	72,121	+1.2
384	6.8	144.9	0.38	97,460	+36.7	5.4	182.7	0.48	75,277	+5.6
512	6.5	149.8	0.29	99,296	+39.3	4.7	208.4	0.41	77,664	+8.9

areas rather than the Infiniband interconnect, MPI processes on the same node can communicate faster than processes on different nodes. Our only attempt to exploit this property in configuring PEBBL was to choose cluster sizes that were multiples of 8.

For runs on fewer 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 [6], and dropping observations with missing fields. Four of the problems are derived from the Hungarian heart disease dataset, the most challenging one tested in [18], 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 4601 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 [12]. This proce-

Table 2 Performance of PEBBL on MMA instance `hung46` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	1836.9			151,265		1836.9			151,265	
2	921.3	2.0	1.00	151,186	−0.1	921.0	2.0	1.00	151,398	+0.1
3	615.3	3.0	1.00	151,263	−0.0	613.5	3.0	1.00	151,286	+0.0
4	458.7	4.0	1.00	151,250	−0.0	458.5	4.0	1.00	151,180	−0.1
6	315.0	5.8	0.97	151,528	+0.2	315.0	5.8	0.97	151,289	+0.0
8	236.8	7.8	0.97	151,516	+0.2	236.8	7.8	0.97	151,495	+0.2
16	118.0	15.6	0.97	151,322	+0.0	117.5	15.6	0.98	151,694	+0.3
24	78.7	23.3	0.97	151,354	+0.1	78.3	23.5	0.98	151,628	+0.2
32	59.1	31.1	0.97	151,459	+0.1	58.8	31.3	0.98	151,693	+0.3
48	39.9	46.0	0.96	151,296	+0.0	39.5	46.5	0.97	151,782	+0.3
64	30.7	59.9	0.94	151,584	+0.2	30.2	60.9	0.95	151,948	+0.5
96	21.0	87.4	0.91	152,023	+0.5	20.4	90.0	0.94	152,208	+0.6
128	16.6	110.9	0.87	152,798	+1.0	15.7	117.1	0.92	152,541	+0.8
192	12.7	144.9	0.75	154,941	+2.4	11.4	161.4	0.84	151,886	+0.4
256	10.6	173.9	0.68	157,209	+3.9	9.3	197.1	0.77	152,699	+0.9
384	8.6	212.6	0.55	164,665	+8.9	7.7	239.8	0.62	153,383	+1.4
512	7.6	241.1	0.47	169,226	+11.9	6.6	277.5	0.54	155,518	+2.8

dure starts by applying its “weak learner”, in this case the MMA solver, with uniform weights. 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 new weights from which the weak learner creates a new term to add to the classification rule, and then the process repeats. As observed in [18], 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. The problems derived from iterations before the 23rd 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 our testing, we left most of PEBBL’s parameters in their default configuration, except those concerned with sizing processor clusters. Processing an MMA subproblem tends to be fairly time-consuming due to the effort involved in computing the equivalence classes used in the bound calculation, which is multiplied by the require-

Table 3 Performance of PEBBL on MMA instance `hung110` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	3623.1			329,489		3623.1			329,489	
2	1806.5	2.0	1.00	328,899	−0.2	1816.6	2.0	1.00	330,541	+0.3
3	1209.5	3.0	1.00	329,026	−0.1	1208.8	3.0	1.00	328,748	−0.2
4	907.8	4.0	1.00	330,675	+0.4	903.8	4.0	1.00	329,142	−0.1
6	618.2	5.9	0.98	329,074	−0.1	618.3	5.9	0.98	329,238	−0.1
8	464.4	7.8	0.98	329,018	−0.1	464.9	7.8	0.97	329,147	−0.1
16	230.9	15.7	0.98	329,600	+0.0	230.4	15.7	0.98	329,160	−0.1
24	153.4	23.6	0.98	329,325	−0.0	153.3	23.6	0.99	329,246	−0.1
32	114.9	31.5	0.99	329,049	−0.1	114.9	31.5	0.99	329,774	+0.1
48	77.0	47.1	0.98	329,207	−0.1	76.9	47.1	0.98	329,774	+0.1
64	59.2	61.2	0.96	330,225	+0.2	58.6	61.8	0.97	329,954	+0.1
96	40.0	90.5	0.94	330,231	+0.2	39.2	92.4	0.96	329,787	+0.1
128	30.6	118.5	0.93	331,207	+0.5	29.8	121.7	0.95	331,066	+0.5
192	22.2	163.2	0.85	332,312	+0.9	21.1	171.5	0.89	330,529	+0.3
256	17.5	206.6	0.81	333,326	+1.2	16.7	217.2	0.85	330,211	+0.2
384	14.1	257.7	0.67	337,697	+2.5	12.5	289.4	0.75	332,331	+0.9
512	11.1	326.4	0.64	340,258	+3.3	10.4	348.4	0.68	333,977	+1.4
768	9.2	394.7	0.51	344,002	+4.4	8.3	434.4	0.57	335,733	+1.9
1024	8.9	408.0	0.40	345,871	+5.0	7.4	489.6	0.48	332,765	+1.0

ments of the strong branching procedure. Because subproblems take relatively long to process—on the order of 0.01–0.02 s for the heart disease problems, and 0.7–0.8 s 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 processor cores. In clusters below 64 processor cores, 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.

Tables 1, 2, 3, 4, 5, 6, 7, 8, and 9 show the results, one table per problem instance. The range of processor cores shown varies by problem instance: four of the spambase

Table 4 Performance of PEBBL on MMA instance `hung253` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	9021.5			847,774		9021.5			847,774	
2	4271.4	2.1	1.06	847,976	+0.0	4270.7	2.1	1.06	848,087	+0.0
3	2842.3	3.2	1.06	848,171	+0.0	2837.0	3.2	1.06	848,144	+0.0
4	2109.1	4.3	1.07	848,004	+0.0	2116.7	4.3	1.07	848,084	+0.0
6	1446.2	6.2	1.04	848,219	+0.1	1443.4	6.2	1.04	848,118	+0.0
8	1084.0	8.3	1.04	848,200	+0.1	1084.6	8.3	1.04	848,385	+0.1
16	537.9	16.8	1.05	848,062	+0.0	538.4	16.8	1.05	848,302	+0.1
24	357.4	25.2	1.05	848,297	+0.1	357.7	25.2	1.05	848,365	+0.1
32	267.8	33.7	1.05	848,302	+0.1	267.5	33.7	1.05	848,606	+0.1
48	178.4	50.6	1.05	848,802	+0.1	178.3	50.6	1.05	848,820	+0.1
64	135.7	66.5	1.04	848,735	+0.1	135.5	66.6	1.04	848,908	+0.1
96	90.0	100.2	1.04	849,005	+0.1	89.7	100.6	1.05	849,388	+0.2
128	68.1	132.4	1.03	849,399	+0.2	67.3	134.0	1.05	849,423	+0.2
192	46.7	193.1	1.01	849,151	+0.2	46.2	195.4	1.02	849,334	+0.2
256	36.0	250.9	0.98	849,904	+0.3	35.1	256.7	1.00	849,569	+0.2
384	25.5	353.5	0.92	850,075	+0.3	24.8	364.4	0.95	849,241	+0.2
512	20.8	434.1	0.85	850,235	+0.3	19.6	461.2	0.90	849,522	+0.2
768	18.7	482.4	0.63	850,625	+0.3	14.5	623.9	0.81	849,569	+0.2
1024	14.6	616.2	0.60	851,105	+0.4	11.9	758.1	0.74	849,433	+0.2
1536	13.8	652.8	0.42	852,023	+0.5	9.3	965.9	0.63	849,771	+0.2
2048	16.3	552.1	0.27	852,799	+0.6	8.7	1041.7	0.51	850,346	+0.3

Table 5 Performance of PEBBL on MMA instance `spam` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	1784.7			2,450		1784.7			2,450	
2	891.1	2.0	1.00	2,451	+0.1	890.7	2.0	1.00	2,449	−0.0
3	596.9	3.0	1.00	2,446	−0.1	596.6	3.0	1.00	2,447	−0.1
4	451.4	4.0	0.99	2,455	+0.2	448.5	4.0	0.99	2,453	+0.1
6	316.2	5.6	0.94	2,469	+0.8	314.4	5.7	0.95	2,474	+1.0
8	244.4	7.3	0.91	2,491	+1.7	240.5	7.4	0.93	2,470	+0.8
16	140.7	12.7	0.79	2,718	+10.9	130.5	13.7	0.86	2,553	+4.2
24	112.9	15.8	0.66	3,071	+25.4	92.7	19.3	0.80	2,614	+6.7
32	101.5	17.6	0.55	3,520	+43.7	67.8	26.3	0.82	2,499	+2.0
48	91.6	19.5	0.41	4,332	+76.8	55.0	32.4	0.68	2,457	+0.3
64	82.0	21.8	0.34	4,863	+98.5	50.7	35.2	0.55	2,463	+0.5
96	75.8	23.6	0.25	5,284	+115.7	47.8	37.3	0.39	2,458	+0.3
128	76.9	23.2	0.18	5,938	+142.4	46.6	38.3	0.30	2,460	+0.4

Table 6 Performance of PEBBL on MMA instance `spam5` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	11799.7			158,904		11642.2			156,181	
16	5863.0	15.9	0.99	158,747	+1.6	5855.3	15.9	0.99	158,598	+1.5
24	3876.1	24.0	1.00	157,975	+1.1	3882.1	24.0	1.00	158,458	+1.5
32	2932.2	31.8	0.99	159,533	+2.1	2905.1	32.1	1.00	158,271	+1.3
48	1937.8	48.1	1.00	158,393	+1.4	1923.9	48.4	1.01	157,531	+0.9
64	1481.6	62.9	0.98	158,086	+1.2	1473.8	63.2	0.99	158,398	+1.4
96	990.9	94.0	0.98	159,513	+2.1	978.0	95.2	0.99	158,911	+1.7
128	747.5	124.6	0.97	159,799	+2.3	738.7	126.1	0.98	159,653	+2.2
192	526.5	176.9	0.92	166,316	+6.5	492.4	189.1	0.99	157,824	+1.1
256	392.6	237.2	0.93	164,004	+5.0	376.2	247.6	0.97	159,796	+2.3
384	281.5	330.9	0.86	172,453	+10.4	257.0	362.4	0.94	160,146	+2.5
512	226.8	410.7	0.80	181,387	+16.1	196.5	473.9	0.93	160,591	+2.8
768	164.7	565.4	0.74	182,631	+16.9	135.8	686.0	0.89	159,639	+2.2
1024	140.9	661.2	0.65	194,038	+24.2	109.5	850.6	0.83	163,256	+4.5
1536	117.0	795.9	0.52	203,513	+30.3	80.6	1155.3	0.75	163,788	+4.9
2048	112.8	825.4	0.40	222,768	+42.6	70.6	1319.6	0.64	165,470	+5.9

Table 7 Performance of PEBBL on MMA instance `spam6` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	22402.1			311,899		22328.3			311,201	
16	11094.1	16.1	1.01	312,018	+0.3	11075.1	16.1	1.01	311,562	+0.1
24	7361.2	24.3	1.01	312,102	+0.3	7349.9	24.3	1.01	311,834	+0.2
32	5499.5	32.5	1.02	311,954	+0.2	5496.2	32.5	1.02	311,787	+0.2
48	3653.0	48.9	1.02	312,008	+0.3	3647.1	49.0	1.02	311,748	+0.2
64	2781.8	64.2	1.00	312,427	+0.4	2776.9	64.3	1.01	311,567	+0.1
96	1844.4	96.9	1.01	312,535	+0.4	1833.8	97.4	1.01	311,674	+0.2
128	1384.5	129.0	1.01	312,862	+0.5	1370.7	130.3	1.02	311,633	+0.1
192	934.1	191.2	1.00	312,814	+0.5	918.1	194.6	1.01	311,658	+0.1
256	704.3	253.6	0.99	312,535	+0.4	689.3	259.1	1.01	311,882	+0.2
384	478.3	373.4	0.97	312,701	+0.5	462.6	386.1	1.01	312,182	+0.3
512	366.9	486.9	0.95	313,810	+0.8	350.9	509.1	0.99	313,428	+0.7
768	254.7	701.2	0.91	312,407	+0.4	238.2	750.0	0.98	314,097	+0.9
1024	204.3	874.5	0.85	313,950	+0.9	182.9	976.8	0.95	314,136	+0.9
1536	153.3	1164.9	0.76	313,509	+0.7	130.0	1373.6	0.89	313,562	+0.8
2048	137.5	1299.1	0.63	313,642	+0.8	105.0	1701.5	0.83	313,415	+0.7
3072	117.2	1523.9	0.50	308,055	−1.0	88.3	2023.4	0.66	314,293	+1.0

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 stopped increasing the number of processor cores after the relative speedup level had departed significantly from linear, or when we reached 8192 processor cores. All CPU times are in seconds, and the "Spdp." and "Eff." columns display relative speedup and relative efficiency, respectively. For problems too difficult to solve on a single processor core, these values are computed by linearly extrapolating the average time of the runs with the fewest cores to estimate the single-core running time. The "Tree growth" columns display the size of the search tree relative to the run(s) with the fewest cores.

Figures 4 and 5 graphically display the run time information for two of the tables. Figure 4 depicts the `hung110` problem, which is of modest difficulty, and Fig. 5 shows `spam26`, the most difficult problem. Each graph uses a log-log scale, with processor cores on the horizontal access and run time (in s) 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 without the synchronous ramp-up phase ($\rho = 0$), and the "$\times$" marks represent the runs with a full synchronous ramp-up phase ($\rho = 1$). 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 $\rho = 0$.

Table 8 Performance of PEBBL on MMA instance `spam12` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	30499.5			420,980		30458.9			420,898	
16	15064.5	16.2	1.01	420,709	−0.0	15052.7	16.2	1.01	420,616	−0.1
24	10000.4	24.4	1.02	421,016	+0.0	9990.1	24.4	1.02	420,782	−0.0
32	7473.9	32.6	1.02	420,829	−0.0	7472.5	32.6	1.02	420,993	+0.0
48	4968.7	49.0	1.02	421,507	+0.1	4957.1	49.2	1.02	420,782	−0.0
64	3780.5	64.5	1.01	421,644	+0.2	3782.6	64.4	1.01	421,103	+0.0
96	2509.3	97.1	1.01	422,114	+0.3	2496.1	97.6	1.02	421,155	+0.1
128	1879.6	129.6	1.01	422,815	+0.5	1864.4	130.7	1.02	421,058	+0.0
192	1260.6	193.3	1.01	424,007	+0.7	1245.8	195.6	1.02	420,920	+0.0
256	951.0	256.2	1.00	425,483	+1.1	934.9	260.6	1.02	421,069	+0.0
384	641.9	379.6	0.99	427,773	+1.6	625.6	389.5	1.01	420,846	−0.0
512	490.8	496.5	0.97	431,180	+2.4	472.3	515.9	1.01	421,498	+0.1
768	337.4	722.3	0.94	434,532	+3.2	320.5	760.4	0.99	421,259	+0.1
1024	264.5	921.1	0.90	442,836	+5.2	242.8	1003.7	0.98	420,520	−0.1
1536	192.5	1266.1	0.82	451,665	+7.3	168.2	1448.7	0.94	418,104	−0.7
2048	159.5	1527.5	0.75	462,696	+9.9	132.7	1836.5	0.90	419,506	−0.3
3072	125.5	1941.6	0.63	468,931	+11.4	96.6	2521.4	0.82	416,004	−1.2
4096	110.0	2216.0	0.54	472,170	+12.2	81.1	3003.1	0.73	418,068	−0.7

The following observations are apparent from the tables and charts:

- Using a synchronous ramp-up phase improves scalability for all problem instances. Its effect is limited for relatively small numbers of processors, but as one increases the processor count it eventually significantly increases efficiency and reduces tree growth.
- 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 the difficulty of the subproblems 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, with an efficiency of 94 %, even at 6144 processor cores. For 8192 cores, efficiency is still 89 %.
- By applying a sufficient number of processors, the solution time for each instance can be reduced to the range of approximately 1–3 min. For example, `spam26` can be solved in less than 3 min on 6144 processor cores, whereas solving it on 8 processor cores takes over 27 h, from which it may be extrapolated that solution on one processor core would require over 9 days.

Table 9 Performance of PEBBL on MMA instance `spam26` on Red Sky, without enumeration

Cores	$\rho = 0$					$\rho = 1$				
	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	114884.1			1,949,533		115029.9			1,952,015	
16	56668.7	16.2	1.01	1,949,472	−0.1	56705.3	16.2	1.01	1,952,580	+0.0
24	37447.6	24.6	1.02	1,949,573	−0.1	37507.7	24.5	1.02	1,952,433	+0.0
32	27930.2	32.9	1.03	1,949,615	−0.1	27973.6	32.9	1.03	1,952,160	+0.0
48	18500.2	49.7	1.04	1,949,517	−0.1	18532.0	49.7	1.03	1,952,282	+0.0
64	14041.1	65.5	1.02	1,949,562	−0.1	14121.9	65.2	1.02	1,955,219	+0.2
96	9273.8	99.2	1.03	1,949,722	−0.1	9284.0	99.1	1.03	1,952,368	+0.0
128	6914.0	133.1	1.04	1,950,024	−0.1	6920.6	133.0	1.04	1,952,336	+0.0
192	4611.3	199.6	1.04	1,950,055	−0.1	4612.7	199.5	1.04	1,952,499	+0.0
256	3448.7	266.8	1.04	1,950,524	−0.1	3447.5	266.9	1.04	1,952,518	+0.0
384	2297.8	400.5	1.04	1,951,492	−0.0	2291.3	401.6	1.05	1,951,256	−0.0
512	1725.9	533.2	1.04	1,952,537	+0.0	1718.1	535.6	1.05	1,952,104	+0.0
768	1155.7	796.2	1.04	1,955,600	+0.2	1145.9	803.0	1.05	1,952,580	+0.0
1024	873.5	1053.6	1.03	1,958,905	+0.4	861.2	1068.6	1.04	1,951,931	−0.0
1536	592.2	1553.9	1.01	1,963,065	+0.6	578.6	1590.3	1.04	1,954,442	+0.1
2048	452.4	2033.9	0.99	1,966,829	+0.8	438.0	2101.0	1.03	1,955,466	+0.2
3072	316.4	2908.7	0.95	1,975,231	+1.2	299.1	3076.7	1.00	1,957,375	+0.3
4096	245.8	3744.5	0.91	1,983,256	+1.6	227.7	4040.7	0.99	1,959,468	+0.4
6144	181.4	5073.5	0.83	1,993,530	+2.1	159.8	5758.7	0.94	1,962,657	+0.5
8192	150.7	6107.2	0.75	2,004,048	+2.7	126.7	7265.4	0.89	1,967,053	+0.8

– Since there is no significant departure from ideal linear speedup 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 (in the runs with 128 or fewer cores) to multiple processor clusters. For example, the `spam26` instance shows linear relative speedup between 128 and 4096 processor cores, even though the 128-processor configuration has a single cluster and the 4096-processor configuration requires the coordination of 32 such clusters.
– Tree growth 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. This form of inefficiency results from a com-

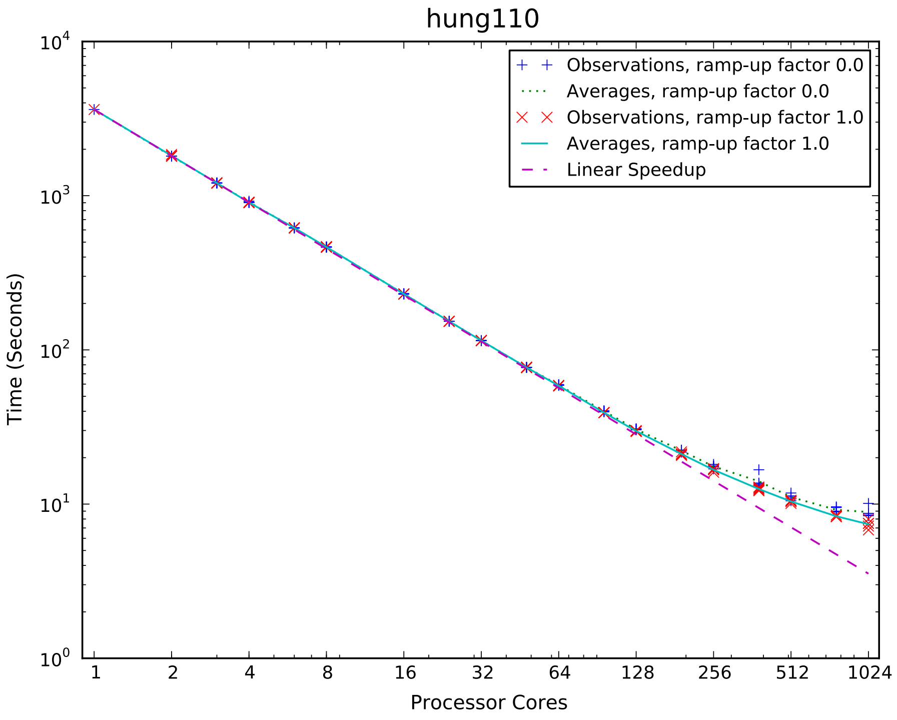

Fig. 4 Speedup behavior on problem instance `hung110`

bination of (1) the application not having a strong incumbent-generating heuristic and (2) it being increasingly difficult to approximate a best-first search order as the number of processors increases. In applications where heuristics can generate high-quality initial incumbents early in the solution process (e.g. the quadratic assignment problem), the main impediments to scalability would instead be processor idleness caused by a lack of available subproblems and inefficiencies in moving subproblems between processors.

– Modestly superlinear speedups, corresponding to efficiencies slightly higher than 1.00, are sometimes observed. We believe that the main reason for this phenomenon is that more cache memory becomes available to the computation as the number of processor cores increases. PEBBL's parallelization has sufficiently low overhead that such speedup effects are observable. In Sect. 5 below, we investigate a more dramatic version of this cache-related speedup effect using a simple knapsack algorithm.

Our second set of tests used PEBBL's enumeration feature. One principle application of the MMA is in a column-generation setting, and column generation algorithms typically try to add several dozen columns to the master problem at a time. Hence, enumeration is a realistic application for MMA.

We used the same set of problems as above, but with `enumCount` $= 25$, instructing PEBBL to find a best possible set of 25 solutions. The `enumCount` enumeration mode

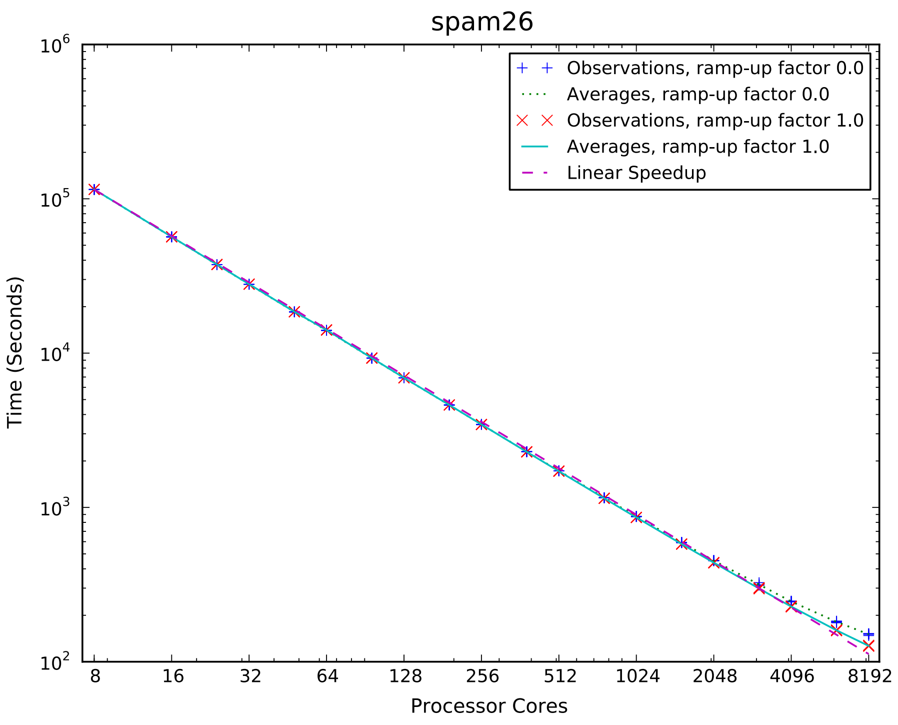

Fig. 5 Speedup behavior on problem instance `spam26`

Table 10 Performance of PEBBL on MMA instance `hung23` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	1179.4			88,492	
2	589.2	2.0	1.00	88,641	+0.2
3	395.3	3.0	0.99	88,584	+0.1
4	294.7	4.0	1.00	88,622	+0.1
6	202.7	5.8	0.97	88,882	+0.4
8	152.1	7.8	0.97	88,703	+0.2
16	77.4	15.2	0.95	89,564	+1.2
24	52.3	22.6	0.94	89,662	+1.3
32	39.7	29.7	0.93	90,116	+1.8
48	28.4	41.5	0.87	90,920	+2.7
64	22.3	52.8	0.83	91,761	+3.7
96	16.2	72.9	0.76	93,062	+5.2
128	14.2	82.9	0.65	96,832	+9.4
192	13.2	89.3	0.47	106,585	+20.4
256	11.8	99.6	0.39	112,266	+26.9
384	10.8	109.4	0.28	122,887	+38.9

Table 11 Performance of PEBBL on MMA instance `hung46` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	1869.7			157,819	
2	932.9	2.0	1.00	156,680	−0.7
3	622.7	3.0	1.00	156,130	−1.1
4	465.8	4.0	1.00	156,785	−0.7
6	322.4	5.8	0.97	158,085	+0.2
8	240.8	7.8	0.97	157,302	−0.3
16	120.3	15.5	0.97	157,071	−0.5
24	80.8	23.2	0.96	157,697	−0.1
32	61.5	30.4	0.95	158,797	+0.6
48	44.0	42.5	0.89	162,240	+2.8
64	35.0	53.4	0.83	165,089	+4.6
96	25.2	74.1	0.77	170,697	+8.2
128	21.3	87.6	0.68	170,332	+7.9
192	16.8	111.3	0.58	176,815	+12.0
256	15.0	124.3	0.49	188,492	+19.4
384	12.9	144.9	0.38	198,787	+26.0
512	12.0	156.3	0.31	204,802	+29.8
768	11.3	165.5	0.22	218,509	+38.5
1024	11.4	164.0	0.16	269,745	+70.9

Table 12 Performance of PEBBL on MMA instance `hung110` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	3999.4			363,786	
2	1972.5	2.0	1.01	364,391	+0.2
3	1317.3	3.0	1.01	364,070	+0.1
4	984.0	4.1	1.02	364,592	+0.2
6	674.0	5.9	0.99	364,646	+0.2
8	505.2	7.9	0.99	364,662	+0.2
16	252.6	15.8	0.99	364,964	+0.3
24	168.9	23.7	0.99	366,077	+0.6
32	127.4	31.4	0.98	366,363	+0.7
48	85.7	46.7	0.97	367,832	+1.1
64	67.1	59.6	0.93	368,094	+1.2
96	46.6	85.8	0.89	369,307	+1.5
128	36.8	108.6	0.85	369,207	+1.5
192	27.9	143.5	0.75	373,400	+2.6
256	22.5	177.4	0.69	373,381	+2.6
384	18.4	217.1	0.57	386,690	+6.3
512	16.2	247.2	0.48	393,499	+8.2
768	13.5	297.1	0.39	405,187	+11.4
1024	12.7	315.4	0.31	417,996	+14.9

Table 13 Performance of PEBBL on MMA instance `hung253` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	9005.3			856,159	
2	4307.4	2.1	1.05	858,106	+0.2
3	2865.3	3.1	1.05	858,233	+0.2
4	2136.2	4.2	1.05	858,863	+0.3
6	1458.8	6.2	1.03	859,347	+0.4
8	1096.1	8.2	1.03	858,963	+0.3
16	545.2	16.5	1.03	861,324	+0.6
24	363.1	24.8	1.03	862,063	+0.7
32	271.9	33.1	1.04	862,801	+0.8
48	182.3	49.4	1.03	864,311	+1.0
64	140.2	64.2	1.00	865,715	+1.1
96	95.0	94.8	0.99	866,883	+1.3
128	72.4	124.5	0.97	866,986	+1.3
192	50.6	178.0	0.93	867,546	+1.3
256	40.7	221.5	0.87	867,817	+1.4
384	29.8	302.4	0.79	869,918	+1.6
512	24.3	370.0	0.72	870,155	+1.6
768	19.1	472.0	0.61	874,577	+2.2
1024	16.2	556.6	0.54	878,796	+2.6
1536	13.9	649.7	0.42	911,618	+6.5
2048	12.8	701.3	0.34	930,150	+8.6

Table 14 Performance of PEBBL on MMA instance `spam` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
1	2163.9			3,422	
2	1092.9	2.0	0.99	3,437	+0.4
3	726.6	3.0	0.99	3,422	+0.0
4	543.7	4.0	1.00	3,419	−0.1
6	379.1	5.7	0.95	3,432	+0.3
8	292.1	7.4	0.93	3,455	+1.0
16	155.5	13.9	0.87	3,511	+2.6
24	107.6	20.1	0.84	3,531	+3.2
32	85.1	25.4	0.79	3,602	+5.3
48	62.9	34.4	0.72	3,600	+5.2
64	53.2	40.7	0.64	3,648	+6.6
96	48.3	44.8	0.47	3,931	+14.9
128	48.6	44.5	0.35	4,339	+26.8
192	48.9	44.3	0.23	5,265	+53.8
256	47.9	45.2	0.18	5,226	+52.7

Table 15 Performance of PEBBL on MMA instance `spam5` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` = 25

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	12,685.6			175,478	
16	6208.1	16.3	1.02	172,599	−1.6
24	4113.8	24.7	1.03	171,991	−2.0
32	3069.8	33.1	1.03	171,688	−2.2
48	2045.4	49.6	1.03	171,850	−2.1
64	1579.7	64.2	1.00	174,476	−0.6
96	1032.2	98.3	1.02	172,014	−2.0
128	775.1	130.9	1.02	172,589	−1.6
192	525.9	193.0	1.01	174,422	−0.6
256	403.9	251.2	0.98	178,658	+1.8
384	274.5	369.7	0.96	179,610	+2.4
512	210.7	481.7	0.94	180,629	+2.9
768	150.0	676.6	0.88	186,893	+6.5
1024	124.8	812.9	0.79	203,088	+15.7

Table 16 Performance of PEBBL on MMA instance `spam6` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` = 25

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth
8	23,029.4			325,033	
16	11,399.5	16.2	1.01	325,104	+0.0
24	7563.6	24.4	1.01	325,211	+0.1
32	5655.5	32.6	1.02	325,242	+0.1
48	3755.5	49.1	1.02	325,357	+0.1
64	2860.9	64.4	1.01	325,322	+0.1
96	1892.6	97.3	1.01	325,730	+0.2
128	1418.2	129.9	1.01	326,302	+0.4
192	954.0	193.1	1.01	327,790	+0.8
256	716.3	257.2	1.00	328,460	+1.1
384	482.8	381.6	0.99	330,368	+1.6
512	368.4	500.1	0.98	333,884	+2.7
768	252.0	731.1	0.95	338,536	+4.2
1024	194.0	949.9	0.93	340,697	+4.8
1536	141.2	1304.6	0.85	347,682	+7.0
2048	118.4	1555.8	0.76	359,656	+10.7
3072	98.6	1867.8	0.61	377,464	+16.1

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 base tree size (for the smallest number of cores tested) changed little for the heart disease problems, but increased by roughly 25 % for the spam problems.

Table 17 Performance of PEBBL on MMA instance `spam12` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	37,125.0			523,287	
16	18,317.7	16.2	1.01	523,199	−0.0
24	12,149.0	24.4	1.02	523,181	−0.0
32	9074.8	32.7	1.02	523,228	−0.0
48	6024.5	49.3	1.03	523,205	−0.0
64	4593.3	64.7	1.01	523,571	+0.1
96	3025.2	98.2	1.02	523,329	+0.0
128	2261.4	131.3	1.03	523,549	+0.1
192	1512.5	196.4	1.02	524,134	+0.2
256	1132.1	262.3	1.02	524,497	+0.2
384	758.0	391.8	1.02	525,668	+0.5
512	571.2	519.9	1.02	526,929	+0.7
768	384.9	771.7	1.00	529,491	+1.2
1024	293.5	1011.8	0.99	530,916	+1.5
1536	202.7	1465.2	0.95	535,375	+2.3
2048	159.7	1859.3	0.91	537,927	+2.8
3072	119.5	2484.9	0.81	546,267	+4.4
4096	100.7	2950.5	0.72	550,352	+5.2

Tables 10, 11, 12, 13, 14, 15, 16, 17, and 18 show results of the experiments using enumeration. 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, scalability is slightly worse in the heart disease problems, since the base search trees are of similar size but the implementation has more communication overhead (especially during the synchronous ramp-up phase, because synchronizing the repository requires significant communication). The results for the spam-detection problems are quite similar to those without enumeration; in some cases scalability improves slightly due to the larger base 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.

5 Cache-related superlinear speedups

Some of the data tables contain a curious phenomenon: numerous relative efficiencies are greater than 1.00, indicating that speedups are slightly better than linear. One mechanism through which such a phenomenon can occur, depending on the subproblem pool ordering and the means of generating incumbent solutions, is that using more processors may allow an incumbent solution to be found at a relatively earlier point in the search process, which shrinks the size of the search tree. However, that is not the case here, because tree sizes gradually increase with the number of processors.

Table 18 Performance of PEBBL on MMA instance `spam26` on Red Sky, $\rho = 1$ only, enumerating multiple solutions with `enumCount` $= 25$

Cores	Time	Spdp.	Eff.	Tree nodes	Tree growth (%)
8	132,996.0			2,338,175	
16	65,612.9	16.2	1.01	2,338,043	−0.0
24	43,430.7	24.5	1.02	2,339,840	+0.1
32	32,282.8	33.0	1.03	2,334,058	−0.2
48	21,405.2	49.7	1.04	2,337,107	−0.0
64	16,302.0	65.3	1.02	2,341,934	+0.2
96	10,687.4	99.6	1.04	2,334,356	−0.2
128	7977.1	133.4	1.04	2,337,007	−0.0
192	5309.1	200.4	1.04	2,332,914	−0.2
256	3979.6	267.4	1.04	2,340,722	+0.1
384	2651.7	401.2	1.04	2,345,555	+0.3
512	1980.4	537.2	1.05	2,336,781	−0.1
768	1324.4	803.4	1.05	2,342,848	+0.2
1024	997.0	1067.1	1.04	2,343,364	+0.2
1536	670.8	1586.0	1.03	2,350,458	+0.5
2048	509.1	2090.1	1.02	2,363,376	+1.1
3072	341.8	3113.0	1.01	2,327,884	−0.4
3384	315.4	3373.6	1.00	2,355,573	+0.7
4096	264.9	4016.5	0.98	2,354,345	+0.7
6144	190.5	5585.7	0.91	2,384,924	+2.0
8192	155.0	6862.5	0.84	2,331,413	−0.3

Instead, the slightly superlinear results are due to processor memory cache behavior. We demonstrate this by studying an instance of a different problem class in which the effect is far more pronounced. Figure 6 shows the speedup behavior on Red Sky of a PEBBL knapsack application on a 3000-object binary knapsack problem in which the object weights and values are strongly correlated. This knapsack application is distributed with PEBBL, but we did not use 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 Fig. 6 all evaluated approximately 200 million subproblems. To avoid congestion at the hubs, we used a much smaller 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" (i.e. hubs do not also function as workers). Otherwise, we used PEBBL's default configuration settings (see Sect. 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 reproducible superlinear speedup occurs. To explain this phenomenon, we first checked the size of the search tree

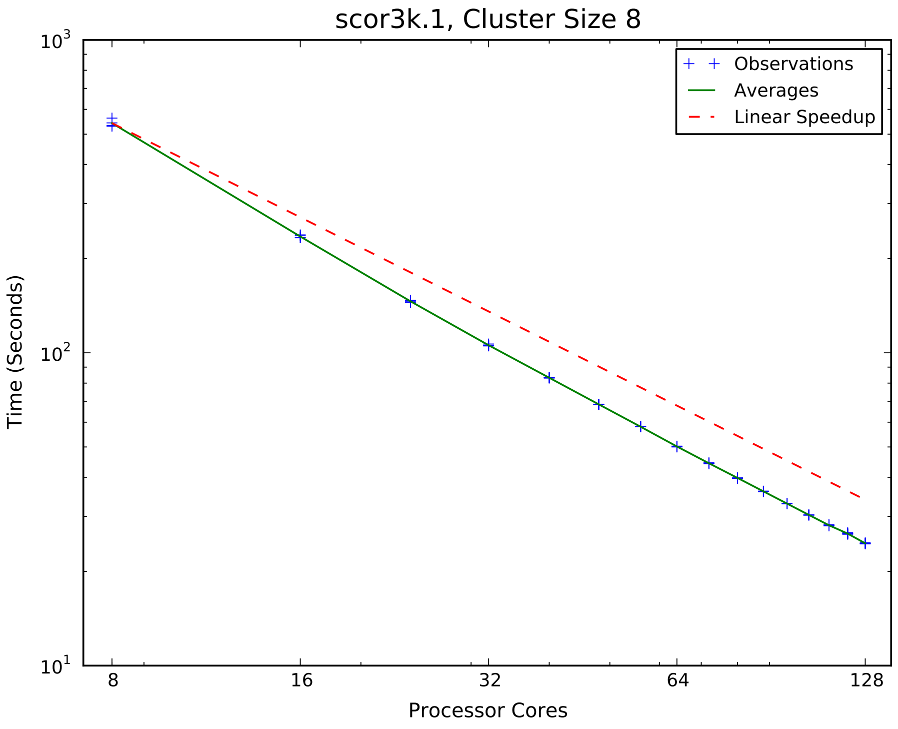

Fig. 6 PEBBL speedups on a 3000-object strongly correlated knapsack problem

explored, but it was virtually constant. We then instrumented the executable code using Open|SpeedShop [44], 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 possible 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 cache memory. Figure 7 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 speed. In practice, as the number of processors increases, the memory resources available to an application often increases as well, including the total amount of each level of cache memory. If the total memory needed to store the active search pool is of a similar magnitude to the total amount of cache memory across all processors, adding cache may increase efficiency more than interprocessor communication overhead decreases it. Despite the apparent complexity of

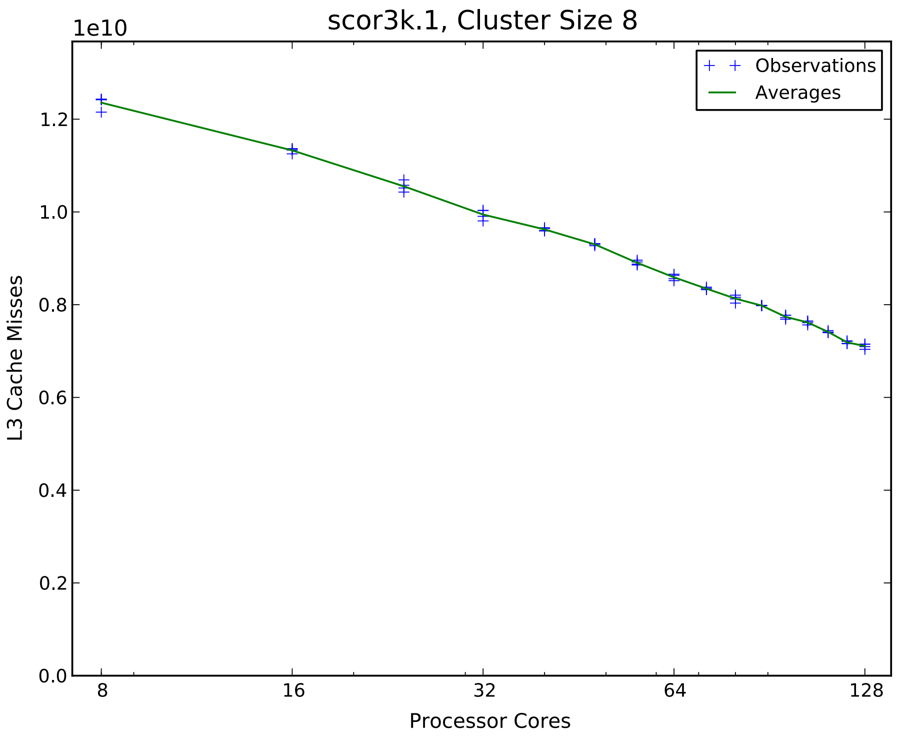

Fig. 7 Number of level-3 cache misses, summed across all processors, for the runs shown in Fig. 6

PEBBL's communication, its overhead is low enough that this situation can indeed occur.

For various kinds of algorithms, superlinear speedups attributed to cache effects have been observed since at least the 1990s [7,53]. More recent examples of such effects are remarked upon in [52], which studies matrix multiplication methods, in [8], which describes matrix factorization methods specifically designed to elicit such effects for particular cache architectures, and in [29], which describes a robot motion planning system. We are not aware of any previous claims of superlinear speedup for branch-and-bound algorithms.

6 Concluding remarks

A clear conclusion from our results is that branch-and-bound algorithms can be highly scalable for applications with large search trees. For the hardest problem instance we considered, `spam26`, we obtained 99 % relative efficiency on 4096 processor cores, 94 % on 6144 cores, and 89 % on 8192 cores. As opposed to some earlier published massively parallel results, we calculated our relative efficiencies relative to base runs with minimal numbers of processors, only 8 cores in the case of difficult instances like `spam26`, and only a single core for the easier ones. We thus demonstrate good scaling more definitively and over a wider range of processor counts than in prior published branch-and-bound work. Despite the complexity of PEBBL's parallelization strategy, it is apparent from our results that it does not add appreciable overhead to the search process for the MMA and knapsack applications. For harder problem instances, furthermore, PEBBL can maintain nearly the same scalability when enumerating multiple MMA solutions (for easier problem instances, enumeration takes a greater toll on scalability, but it remains good).

Our computational results do not include any direct comparisons with other parallel branch-and-bound frameworks. To this end, we attempted to empirically compare PEBBL to ALPS, because it is the only similar generic branch-and-bound framework with published scaling results on thousands of processors. Furthermore, these results were for a simple knapsack algorithm mathematically identical to the knapsack example already present in PEBBL. Unfortunately, ALPS' implementation of this algorithm is much slower and more memory-intensive than PEBBL's, making "head-to-head" comparisons difficult. Furthermore, we could not get the ALPS knapsack application to run reliably on the Red Sky platform we used for computational testing. Therefore, we were forced to abandon direct comparison with ALPS.

Our results may have implications regarding parallel computer architectures, in particular the current trend toward large, cache-coherent global memories. The results we have obtained with PEBBL, especially in Sect. 5, suggest that fast local memory, including local cache, may be more critical than global memory 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 domain, 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. Although it might be more scalable, a MIP solver based on PEBBL would differ from the parallel branch-and-cut solvers in current commercial MIP packages in that PEBBL makes no effort to enforce determinism: two runs of the same problem instance on the same processor configuration could easily explore different numbers of search nodes and find different optimal solutions (but with the same objective value within the specified termination tolerance). Depending on the application, such nondetermism may be either desirable or undesirable.

Acknowledgments 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.

References

1. Asuncion, A., Newman, D.J.: UCI machine learning repository (2007). http://www.ics.uci.edu/~mlearn/MLRepository.html
2. Bader, D.A., Hart, W.E., Phillips, C.A.: Parallel algorithm design for branch and bound. In: Greenberg, H.J. (ed.) Tutorials on Emerging Methodologies and Applications in Operations Research, pp. 5-1–5-44. Kluwer Academic, Dordrecht (2004)
3. Benaïchouche, M., Cung, V.D., Dowaji, S., Cun, B.L., Mautor, T., Roucairol, C.: Building a parallel branch and bound library. In: Solving Combinatorial Optimization Problems in Parallel—Methods and Techniques, Lecture Notes in Computer Science, vol. 1054, pp. 201–231. Springer-Verlag, London (1996)
4. Bendjoudi, A., Melab, N., Talbi, E.G.: 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), pp. 1806–1814. Anchorage, Alaska (20011)
5. Blelloch, G.E.: Scans as primitive parallel operations. IEEE Trans. Comput. **38**(11), 1526–1538 (1989)
6. Boros, E., Hammer, P.L., Ibaraki, T., Kogan, A.: Logical analysis of numerical data. Math. Program. **79**(1–3), 163–190 (1997)
7. Brewer, T.: A highly scalable system utilizing up to 128 PA-RISC processors. In: COMPCON, Technologies for the Information Superhighway, Digest of Papers, pp. 133–140 (1995). http://dblp.uni-trier.de/db/conf/compcon/compcon95.html#Brewer95
8. Cataldo, A.M., Whaley, R.C.: Scaling LAPACK panel operations using parallel cache assignment. In: ACM Principles and Practice of Parallel Programming. ACM, New York (2010)
9. Clausen, J.: Branch and bound algorithms—principles and examples. In: Migdalas, A., Pardalos, P., Storøy, S. (eds.) Parallel Computing in Optimization, Applied Optimization, vol. 7, pp. 239–267. Kluwer Academic, Dordrecht (1997)
10. Clausen, J., Perregaard, M.: 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)
11. Danna, E., Fenelon, M., Gu, Z., Wunderling, R.: Generating multiple solutions for mixed integer programming problems. In: Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science, vol. 4513, pp. 280–294. Springer, Berlin (2007)
12. Demiriz, A., Bennett, K.P., Shawe-Taylor, J.: Linear programming boosting via column generation. Mach. Learn. **46**, 225–254 (2002)
13. Dobkin, D.P., Gunopulos, D., Maass, W.: Computing the maximum bichromatic discrepancy, with applications to computer graphics and machine learning. J. Comp. Syst. Sci. **52**(3), 453–470 (1996)
14. Eckstein, J.: Control strategies for parallel mixed integer branch and bound. In: Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, pp. 41–48. ACM, New York (1994)
15. Eckstein, J.: Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. SIAM J. Optim. **4**(4), 794–814 (1994)
16. Eckstein, J.: 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)
17. Eckstein, J.: How much communication does parallel branch and bound need? INFORMS J. Comput. **9**(1), 15–29 (1997)
18. Eckstein, J., Goldberg, N.: An improved branch-and-bound method for maximum monomial agreement. INFORMS J. Comput. **24**(2), 328–341 (2012)
19. Eckstein, J., Hart, W.E., Phillips, C.: Massively parallel mixed-integer programming: algorithms and applications. In: Heroux, M.A., Raghavan, P., Simon, H. (eds.) Parallel Processing for Scientific Computing, chapter 17, pp. 323–340. Based on the 11th SIAM Conference on Parallel Processing for Scientific Computing. SIAM (2006)
20. Eckstein, J., Hart, W.E., Phillips, C.A.: Resource management in a parallel mixed integer programming package. In: Proceedings of the Intel Supercomputer Users Group (1997). Online proceedings
21. Eckstein, J., Phillips, C.A., Hart, W.E.: PICO: An object-oriented framework for parallel branch and bound. In: Inherently parallel algorithms in feasibility and optimization and their applications (Haifa, 2000), Stud. Comput. Math., vol. 8, pp. 219–265. North-Holland, Amsterdam (2001)
22. Eckstein, J., Phillips, C.A., Hart, W.E.: PEBBL: An object-oriented framework for scalable parallel branch and bound. RUTCOR Research Report RRR 9-2013, Rutgers University (2013)

23. Eckstein, J., Phillips, C.A., Hart, W.E.: PEBBL 1.4.1 user's guide. RUTCOR Research Report RRR 2-2014, Rutgers University (2014)
24. Elf, M., Gutwenger, C., Jünger, M., Rinaldi, G.: Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS. In: Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions, pp. 157–222. Springer-Verlag, London (2001)
25. Goldberg, N., Shan, C.: Boosting optimal logical patterns. In: Proceedings of the Seventh SIAM International Conference on Data Mining, pp. 228–236. SIAM, Minneapolis (2007)
26. Graham, R.L., Woodall, T.S., Squyres, J.M.: Open MPI: A flexible high performance MPI. In: Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics. Poznan, Poland (2005)
27. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: High-performance, portable implementation of the MPI Message Passing Interface Standard. Parallel Comput. **22**(6), 789–828 (1996)
28. Hillis, W.D.: The Connection Machine. MIT Press, Cambridge (1985)
29. Ichnowski, J., Alterovitz, R.: Parallel sampling-based motion planning with superlinear speedup. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1206–1212 (2012)
30. Jünger, M., Thienel, S.: Introduction to ABACUS–a branch-and-cut system. Oper. Res. Lett. **22**(2–3), 83–95 (1998)
31. Jünger, M., Thienel, S.: The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. Software Pract. Expert. **30**, 1325–1352 (2000)
32. Karypis, G., Kumar, V.: Unstructured tree search on SIMD parallel computers: a summary of results. In: Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, pp. 453–462. IEEE Computer Society Press, Los Alamitos (1992)
33. Kearns, M.J., Schapire, R.E., Sellie, L.M.: Toward efficient agnostic learning. Mach. Learn. **17**(2–3), 115–141 (1994)
34. Kim, M., Lee, H., Lee;, J.: A proportional-share scheduler for multimedia applications. In: IEEE International Conference on Multimedia computing and systems '97, pp. 484–491. IEEE Computer Society Press, Los Alamitos (1997)
35. Koch, T., Ralphs, T., Shinano, Y.: Could we use a millon cores to solve an integer program? Math. Methods Oper. Res. **76**, 67–93 (2012)
36. Ladányi, L., Ralphs, T.K., Trotter Jr., L.E.: Branch, cut, and price: Sequential and parallel. In: Computational combinatorial optimization (Schloß Dagstuhl, 2000), Lecture Notes in Computer Science, vol. 2241, pp. 223–260. Springer, Berlin (2001)
37. Mahanti, A., Daniel, C.J.: A SIMD approach to parallel heuristic search. Artif. Intel. **60**, 243–282 (1993)
38. Mattern, F.: Algorithms for distributed termination detection. Distrib. Comput. **2**, 161–175 (1987)
39. Menouer, T., LeCun, B., Vander-Swalmen, P.: Partitioning methods to parallelize constraint programming solver using the parallel framework BobPP. In: Proceedings of the First International Conference on Computer Science, Applied Mathematics and Applications (ICCSAMA), pp. 117–127. Springer International Publishing, Switzerland (2013)
40. Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.C.: MINTO, a mixed INTeger optimizer. Oper. Res. Lett. **15**(1), 47–58 (1994)
41. Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Parallel branch, cut, and price for large-scale discrete optimization. Math. Program. Ser. B **98**(1–3), 253–280 (2003)
42. Ralphs, T.K., Ládanyi, L., Saltzman, M.J.: A library hierarchy for implementing scalable parallel search algorithms. J. Supercomput. **28**(2), 215–234 (2004)
43. Rayward-Smith, V.J., Rush, S.A., McKeown, G.P.: Efficiency considerations in the implementation of parallel branch-and-bound. Ann. Oper. Res. **43**(1–4), 123–145 (1993)
44. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W., Montoya, D., Cranford, S.: Open|Speedshop: an open source infrastructure for parallel performance analysis. Sci. Program. **16**(2–3), 105–121 (2008)
45. Shinano, Y.: ParaSCIP and fiberSCIP libraries to parallelize a customized SCIP solver (2014). http://scip.zib.de/workshop/parascip_libraries.pdf. SCIP Workshop 2014
46. Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T.: paraSCIP: a parallel extension of SCIP. In: Competence in High Performance Computing 2010, pp. 135–148. Springer-Verlag, Belin Heidelberg (2012)
47. Shinano, Y., Harada, K., Hirabayashi, R.: Control schemes in a generalized utility for parallel branch-and-bound algorithms. In: IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing, p. 621. IEEE Computer Society, Washington, DC (1997)

48. Shinano, Y., Higaki, M., Hirabayashi, R.: A generalized utility for parallel branch and bound algorithms. In: SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, p. 392. IEEE Computer Society, Washington, DC (1995)
49. Snir, M., Otto, S.W., Walker, D.W., Dongarra, J., Huss-Lederman, S.: MPI: The Complete Reference. MIT Press, Cambridge (1995)
50. Tschöke, S., Polzer, T.: Portable parallel branch-and-bound library: PPBB-Lib user manual version 2.0. Tech. rep., Department of Computer Science, University of Paderborn (1996)
51. Waldspurger, C.A.: Lottery and stride scheduling: flexibile proportional-share resource management. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge (1996)
52. Wu, M.S., Aluru, S., Kendall, R.A.: Mixed mode matrix multiplication. In: Proceedings of the IEEE International Conference on Cluster Computing, pp. 195–203 (2002)
53. Wu, X. (ed.): Performance Evaluation, Prediction and Visualization of Parallel Systems, vol. 4 of The Kluwer International Series on Asian Studies in Computer and Information Science. Kluwer Academic, Springer (1999)
54. Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Computational experience with a software framework for parallel integer programming. INFORMS J. Comput. **21**, 383–397 (2009). http://coral.ie.lehigh.edu/~ted/files/papers/CHiPPS-Rev.pdf