RUTCOR
RESEARCH
REPORT

PEBBL 1.4.1 User's Guide

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

RRR 2-2014, Sep 2014,

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

[a]Management Science and Information Systems and RUTCOR, Rutgers University, 100 Rockafeller Road, Piscataway, NJ 08854

[b]Sandia National Laboratories, Mail Stop 1318, P.O. Box 5800, Albuquerque, NM 87185-1318

Rutcor Research Report
RRR 2-2014, Sep 2014,

PEBBL 1.4.1 User's Guide

Jonathan Eckstein Cynthia A. Phillips William E. Hart

Abstract. PEBBL is a C++ framework for implementing general parallel branch-and-bound optimization algorithms, providing a mechanism for the efficient implementation of a wide range of branch-and-bound methods on an large variety of parallel computing platforms. This document describes:

- The history, goals, and general properties of PEBBL
- How to download and compile PEBBL
- PEBBL's special search capabilities, including enumeration, early output, and checkpointing
- PEBBL's ability to enumerate multiple solutions meeting various criteria
- PEBBL's basic architecture, including its serial and parallel layers
- The design of the serial layer and the notion of manipulating subproblem states
- The design and capabilities of the parallel layer
- How to build a simple serial branch-and-bound algorithm using PEBBL
- How to extend a serial implementation into a parallel one
- Many of the numerous parameters that can be used to control PEBBL's behavior.

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

1 Introduction

1.1 What is PEBBL?

PEBBL (*P*arallel *E*numeration and *Branch*-and-*B*ound *L*ibrary) is a C++ class library for constructing serial and parallel branch-and-bound optimization algorithms. It is a *framework*, *shell*, or *skeleton* that handles the generic aspects of branch and bound, allowing the developer to focus primarily on the unique aspects of their particular branch-and-bound algorithm. It is thus similar to other software projects such as PUBB [18, 17], BoB [12], PPBB-Lib [20], and ALPS [15] (PEBBL's development has significantly influenced the architecture of ALPS). PEBBL has a number of unique features, including very flexible parallelization strategies and the ability to enumerate near-optimal solutions.

In principle, one can build an arbitrary branch-and-bound method atop PEBBL by defining a relatively small number of abstract methods. By defining a few more methods, the algorithm can be immediately parallelized. PEBBL contains numerous tuning parameters that can adapt the resulting parallel implementation to any parallel architecture that supports an MPI message passing library [19].

1.2 The Genealogy of PEBBL

Most of the development work on PEBBL has been carried out by

- Jonathan Eckstein, Rutgers University
- William Hart, Sandia National Laboratories
- Cynthia A. Phillips, Sandia National Laboratories.

PEBBL was originally the "core" layer of the PICO (*P*arallel *I*nteger and *C*ombinatorial *O*ptimization) package. PICO was designed to solve mixed integer programming problems, but included a "core" layer supporting implementation of arbitrary branch and bound algorithms. In the Spring of 2006, the development team decided to distribute this core layer as a software package in its own right, changing its name from "the PICO core" to PEBBL. Much of PEBBL's basic design is thus described in preliminary publications concerning PICO [6, 8, 7]. In fact, significant portions of this user guide are derived from Eckstein *et al.* [7, 8].

PEBBL's parallelization strategies are partially patterned after CMMIP [4, 5], a parallel mixed integer programming solver developed for the Thinking Machines CM-5 parallel supercomputer. CMMIP, however, was specifically designed for mixed integer programming, and to take advantage of particular features of the CM-5 architecture. PEBBL is more generic in two senses: it is a framework that one can use to implement any branch-and-bound algorithm, and it is designed to run in a generic message-passing environment. The ABACUS package [10] also influenced some of PEBBL's design.

2 Downloading and Compiling PEBBL

The PEBBL software project is supported by the Acro optimization project. Acro supports the integration of PEBBL builds with auxiliary libraries, like UTILIB. PEBBL in turn supports other Acro projects, such as PICO.

Compiling PEBBL requires a unix-like environment such as Linux or the Cygwin Linux emulator for Microsoft Windows. This guide assumes basic familiarity with the command-line interface to such environments.

The instructions below describe how to download and compile PEBBL and a minimal set of supporting packages, collectively referred to as `acro-pebbl`. You may also download and compile PEBBL as a component of larger packages such as PICO; to do so, see the documentation for those packages.

2.1 System requirements

- A unix-like operating systems (including Linux, Mac OS X, or the Cygwin emulator for windows)
- The subversion (`svn`) version management system
- Python
- GNU autoconf
- GNU libtool
- A C/C++ compiler, such as gcc/g++
- For parallel execution, some form of MPI, such as OpenMPI or MPICH.

These packages should be readily installable in most unix-like systems. In Ubuntu Linux, for example, subversion, Python, autoconf, libtool, gcc, and OpenMPI can all be easily installed from the standard package manager or software center.

Note that Python is needed only because it is used in some of PEBBL's configuration and debugging utilities. Branch-and-bound applications built with PEBBL will contain only C++ code unless the user explicitly includes code written in other languages.

2.2 Downloading PEBBL

To install the latest version of PEBBL, make sure Subversion is already installed on your system and issue the following command:

```
svn checkout https://software.sandia.gov/svn/acro/acro-pebbl/trunk acro-pebbl
```

This command will create a directory called `acro-pebbl`. To download specific release, execute a command of the following form

```
svn checkout \
    https://software.sandia.gov/svn/acro/acro-pebbl/releases/release \
    acro-pebbl-release
```

This command will create a directory called `acro-pebbl-`*`release`*. For example, to download release 1.4.1, you would give the command

```
svn checkout \
    https://software.sandia.gov/svn/acro/acro-pebbl/releases/1.4.1 \
    acro-pebbl-1.4.1
```

This command should produce a directory called `acro-pebbl-1.4.1`.

Throughout this guide, will assume that PEBBL resides in a directory called `acro-pebbl`. Specific release checkout directories of the form `acro-pebbl-`*`release`* may be used interchangeably with `acro-pebbl` in all directions and procedures below.

2.3 Compiling: Configuring and Building

To configure and build PEBBL, enter the `acro-pebbl` directory (using `cd acro-pebbl`, for example). Then issue the following commands:

```
./setup
autoreconf -i -f
./configure [options]
[make clean]
make
```

Note that to configure PEBBL, your system must have `autoconf` installed. The *options* after `./configure` may be omitted; if so, you will get a default configuration that effectively compiles only PEBBL's serial layer (see Section 3). See Section 2.3.1 below for a description of the possible configuration options. The `make clean` step is only required if you already configured PICO in the same directory, but have changed your system configuration or the arguments to `./configure`. If you do not need any options to `./configure`, you may replace the four-command sequence above with the simple command

```
./setup configure make
```

This method is "quiet", redirecting all configuration and compilation output to the `test` subdirectory of `acro-pebbl`, in the files

`config.out`	The output of `autoreconf` and `configure`
`config.xml`	Summary of `config.out` to detect errors
`build.out`	The output of `make`
`build.xml`	Summary of `build.out` to detect errors

2.3.1 Options for `configure`

A full list of the possible configuration options may be be obtained by executing

```
configure --help
```

Some of the more common options are as follows:

`--enable-validating`
: Increase internal error checking, with a slight performance and executable size penalty.

`--with-debugging`
: Compile with additional debug printout capabilities and symbol table information for stack traces and system debugging tools. There may be slight performance and executable size penalty.

`--without-optimization`
: Disable compiler optimization; yields faster compilation at the cost of slower execution and larger executable files.

`--with-mpi-compilers=yes`
: Enable MPI and the parallel layer, using MPI compilers found in the current shell path.

`--with-mpi-compilers=`*pathname*
: Enable MPI and the parallel layer, using MPI compilers installed at *pathname*.

3 Architecture and Features

PEBBL consists of two *layers*, the *serial layer* and the *parallel layer*. The serial layer provides an object-oriented means of describing branch-and-bound algorithms, with very little reference to parallel implementation. If you do not need parallelism, or are simply in the early stages of algorithm development, the serial layer allows branch-and-bound methods to be described, debugged, and run in a familiar, serial programming environment.

The parallel layer contains the core code necessary to create parallel versions of serial applications. To parallelize a branch-and-bound application developed with the serial layer, you simply define new classes derived from *both* the serial application and the parallel layer. A fully-operational parallel application only requires the definition of a few additional methods for these derived classes, principally to tell PEBBL how to pack application-specific problem and subproblem data into MPI message buffers, and later unpack them.

Any parallel PEBBL application constructed in this way inherits the full capabilities of the parallel layer, including a wide range of different parallel work distribution and load balancing strategies, and user-configurable levels of interprocessor communication. You can then add application-specific refinements to the parallelization, but are not required to. Figure 1 shows the conceptual relationship between the two layers, a serial application, and its parallelization. In the figure, the application is one of the examples distributed with PEBBL, for solving binary knapsack problems; the same basic pattern applies to all PEBBL applications. The serial layer class implementing the branch and bound algorithm is called `binaryKnapsack`, and the parallel application is called `parallelBinaryKnapsack`.

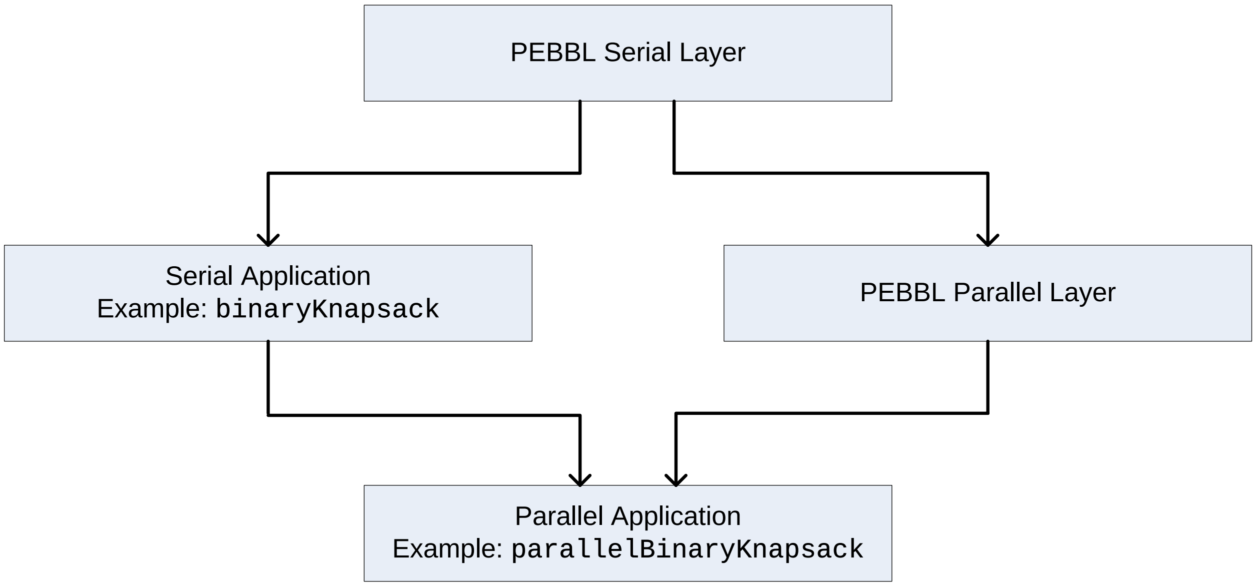

Figure 1: The conceptual relationships of PEBBL's serial layer, the parallel layer, a serial application (in this case, `binaryKnapsack`), and the corresponding parallel application (in this case, `parallelBinaryKnapsack`).

The "diamond" inheritance structure shown in Figure 1 is integral to PEBBL's design — it is a powerful but sometimes problematic use of C++ multiple inheritance.

3.1 Serial layer architecture

3.1.1 The `branching`, `branchSub`, and `solution` classes

To define a serial branch-and-bound algorithm, you must extend the two or three key classes in the PEBBL serial layer, `branching`, `branchSub`, and possibly `solution`. The `branching` class stores global information about a problem instance, and contains methods that implement various kinds of serial branch-and-bound algorithms, as described below. Each `branchSub` object stores data about a subproblem (or node) in the branch-and-bound tree, and the `branchSub` class contains methods that perform generic operations on subproblems. The `solution` class stores a description of a feasible solution to the problem. PEBBL provides some standard `solution`-derived template classes, essentially representing solutions as vectors of standard C++ types such as `int` or `double`; for other solution representations, users should create their own classes derived from `solution`. The basic `branching-branchSub-solution` organization of PEBBL is inspired by ABACUS [10], but is more general, since there is no assumption that cutting planes or linear programming are involved.

By way of example of PEBBL's class structure, our binary knapsack solver defines a class `binaryKnapsack`, derived from `branching`, to describe the capacity of the knapsack and the possible items to be placed in it. We also define a class `binKnapSub`, derived from

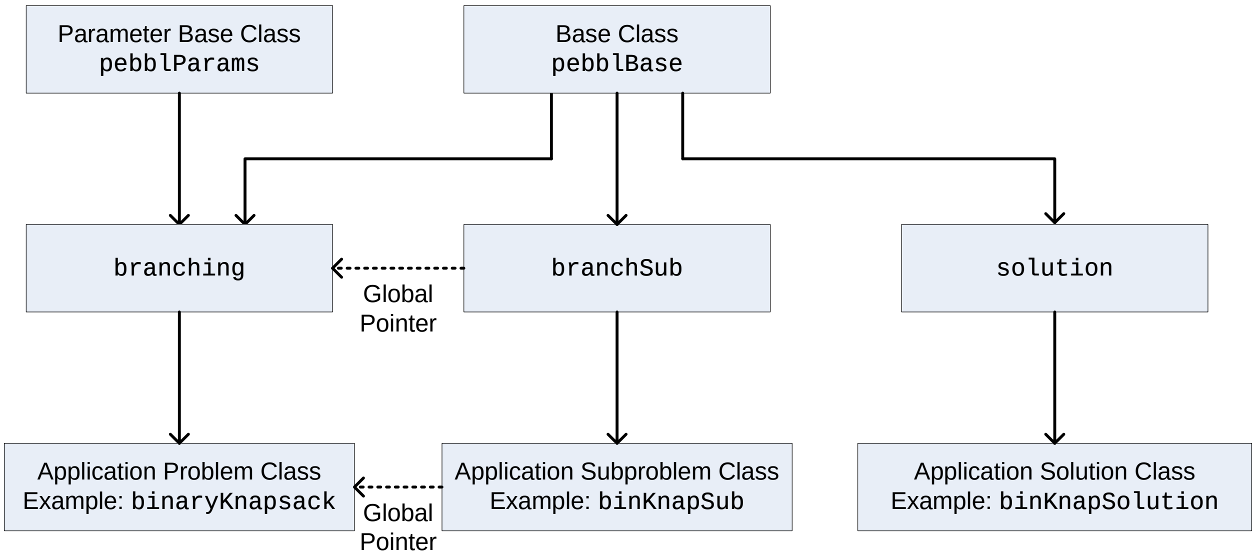

Figure 2: Basic class hierarcy for a serial PEBBL application (in this case, `binaryKnapsack`, with corresponding subproblem class `binKnapSub` and solution class `binKnapSolution`).

`branchSub`, which describes the status of the knapsack items at nodes of the branching tree (*i.e.*, included, excluded, or undecided); this class describes a node of the branch-and-bound tree. Each object in a subproblem class like `binKnapSub` contains a pointer back to the corresponding instance of the "global" problem class, in this case `binaryKnapsack`. Through this pointer, each subproblem object can find global information about the problem instance and the overall branch-and-bound process. The knapsack application also defines a class `binKnapSolution` derived from `solution`, to represent solutions in a compact form that is particularly efficient for large knapsack problems.

Both `branching` and `branchSub` are derived from a common base class, `pebblBase`, which mainly contains common symbol definitions. The `branching` class also derives from `pebblParams`, which holds command-line-specifiable parameter objects implemented using the UTILIB class parameter package. The `solution` class also derives from `pebblBase`. Figure 2 illustrates the basic class hierarchy for a serial PEBBL application.

The class header file for the serial binary knapsack example is in `acro-pebbl/packages/pebbl/src/example/pebbl/serialKnapsack.h`. Note that `binaryKnapsack` is defined via

```
class binaryKnapsack :  virtual public branching { ··· }  ,
```

`binKnapSub` is defined via

```
class binKnapSub :  virtual public branchSub ··· { ··· }  ,
```

and `binKnapSolution` is defined via

```
class binKnapSolution :  public solution ··· { ··· }  .
```

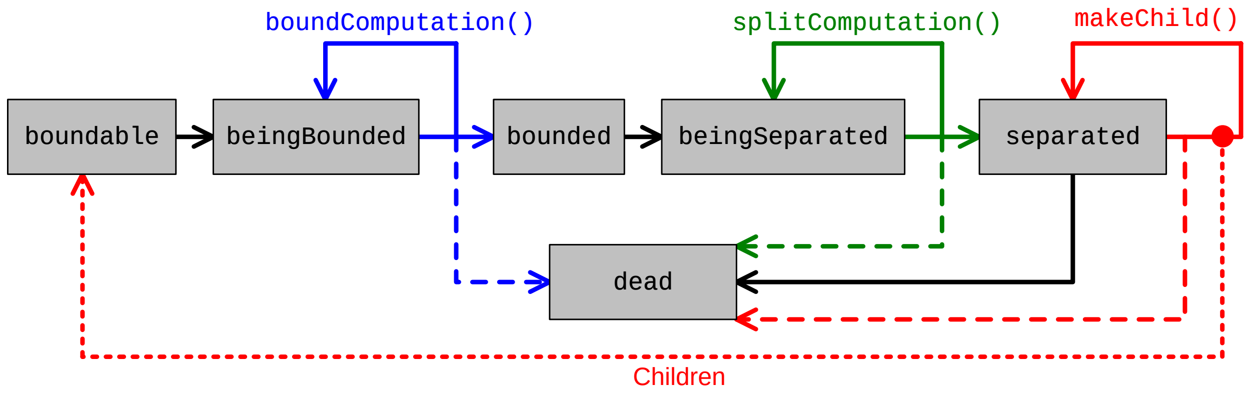

Figure 3: PEBBL's subproblem state transition diagram. It is possible that a single application of `boundComputation` may take a subproblem from the `boundable` state, through `beingBounded`, to `bounded`. Similarly, a single use of `splitComputation` may move a subproblem from `bounded`, through `beingSeparated`, to `separated`.

In order for subsequent parallization to work properly, you should use `virtual` when deriving classes from `branching` and `branchSub`. It is typically not necessary to use `virtual` when deriving classes from `solution`.

Calculations made for subproblems frequently require data stored in the problem description class, which are accessible via pointers as depicted by the two horizontal dotted arrows in Figure 2. To implement the upper arrow, the definition of the subproblem class must instantiate the abstract method `bGlobal()`; in `binKnapSub`, for example, this capability is implemented via:

```
protected:
   binaryKnapsack* globalPtr;
public:
   inline binaryKnapsack* global() const { return globalPtr; };
   branching* bGlobal() const { return global(); };
```

This pattern should be fairly typical: each object of the `branchSub`-derived class should contain a pointer to the `branching`-derived object for which it represents a subproblem. The `bGlobal()` method should then be implemented by casting this pointer to a `branching*`.

3.1.2 Manipulating subproblem states

A key feature of PEBBL, first published in Eckstein et al. [8], is that subproblems remember their *state*. Each subproblem progresses through as many as six of these states, `boundable`, `beingBounded`, `bounded`, `beingSeparated`, `separated`, and `dead`, as illustrated in Figure 3.

A subproblem always comes into existence in state `boundable`, meaning that little or no bounding work has been done for it, although it still has an associated bound value; typically, this bound value is simply inherited from the parent subproblem. Once PEBBL starts work on bounding a subproblem, its state becomes `beingBounded`, and when the bounding work is complete, the state becomes `bounded`.

Once a problem is in the `bounded` state, PEBBL may elect to split it into smaller subproblems. At this point, the subproblem's state becomes `beingSeparated`. Once separation is complete, the state becomes `separated`, at which point the subproblem's children may be created. Once the last child has been created, the subproblem's state becomes `dead`, and it may be deleted from memory. Subproblems may also become `dead` at earlier points in their existence, because they have been fathomed or represent portions of the search space containing no feasible solutions.

Class `branchSub` has three key abstract virtual methods, namely `boundComputation`, `splitComputation`, and `makeChild`, that are responsible for applying these state transitions to subproblems. PEBBL's search framework interacts with applications primarily through these methods; defining a PEBBL branch-and-bound application essentially consists of providing definitions for these three operators for the application subproblem class (*e.g.* `binKnapSub`).

The `boundComputation` method's job is to move the subproblem to the `bounded` state, updating the data member `bound` to reflect the computed bound value. The `boundComputation` method is allowed to pause an indefinite number of times, leaving the subproblem in the `beingBounded` state. The only requirement is that any subproblem will eventually become `bounded` after some finite number of applications of `boundComputation`. This flexibility allows PEBBL to support branch-and-bound variants where bounding is suspended on a subproblem, the subproblem is set aside, and another task or subproblem is considered. It also allows for bounding procedures that establish progressively stronger bounds through multiple stages of computation. The subproblem's bound, reflected in the data member `bound`, may change at each step of this process. When `boundComputation` decides that there is no more bounding work to be done for subproblem, it should change the subproblem state to bounded by executing `setState(bounded)`. Changes in subproblem state should be implemented via the `setState` rather than by direct assignment to the data member `state` to ensure PEBBL can keep accurate subproblem statistics.

The `splitComputation` method's job is similar to `boundComputation`'s, but it manages the process of splitting subproblems. Eventually it must execute `setState(separated)` to signal that the subproblem is completely separated, and then return the number of child subproblems (`splitComputation` has a return type of `int`). Before that, however, it is allowed to return an indefinite number of times with the problem left in the `beingSeparated` state (in which case the return value is ignored). This feature allows PEBBL to implement branch-and-bound methods where the work in separating a subproblem is substantial and might need to be paused to attend to some other subproblem or task. The subproblem's bound may be updated by `splitComputation` if the separation process yields additional bounding information.

Finally, `makeChild` returns a `branchSub*` pointing to a single child of the subproblem it is applied to. This parent must be in the `separated` state. After its last child has been made, PEBBL automatically puts the subproblem in the `dead` state.

If at any point in `boundComputation`, `splitComputation`, or `makeChild`, it becomes evident that a subproblem does not require further investigation — for example, because it has become evident the subproblem is infeasible — one may mark the subproblem as `dead` by executing `setState(dead)`.

In addition to `boundComputation`, `splitComputation`, and `makeChild`, some additional virtual methods must to be defined to complete the specification of a branch-and-bound application; all these methods are described in Section 4.2.

3.1.3 Pools, handlers, and the search framework

PEBBL's serial layer orchestrates branch-and-bound search through a module called the "search framework", literally, `branching::searchFramework`. The search framework acts as an attachment point for two user-specifiable objects, a *pool* and a *handler*, whose combination determines the exact "flavor" of branch and bound being implemented.

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 pool: a heap sorted by subproblem bound[1], a stack, and a FIFO queue. If you specify the heap pool, then PEBBL will follow a best-first search order; specifying the stack pool results in a depth-first order, and specifying the queue results in a breadth-first order.

Critically, at any instant in time, the subproblems in the pool may in principle represent any mix of states: for example, some might be `boundable`, and others `separated`. This feature gives you flexibility in specifying the *bounding protocol*, which is a different aspect of the algorithm than the search order; each "handler" object implements a particular bounding protocol.

To illustrate what a bounding protocol is, consider the usual branch-and-bound method for mixed integer programming as typically described by operations researchers: one removes a subproblem from the currently active pool, and computes its linear programming relaxation bound. If the bound is strong enough to fathom the subproblem, it is discarded. Otherwise, one selects a branching variable, creates two child subproblems, and inserts them into the pool. This type of procedure is an example of what is often called "lazy" bounding (see for instance [2]), because it views the bounding procedure as something time-consuming (like solving a large linear program) that should be delayed if possible. In the PEBBL framework, lazy bounding is implemented by a handler that tries to keep all subproblems in the active pool in the `boundable` state.

An alternative approach, common in work originating from the computer science community, is usually called "eager" bounding (again, see [2] for an example of this terminology). Here, all subproblems in the pool have already been bounded. One picks a subproblem out of

[1]Objects of type `branchSub` have a member called `integralityMeasure` which may be used by the application to measure how far a subproblem is from being completely feasible (that is, from having `candidateSolution` yield `true`; see Section 4.2). If two subproblems have identical bounds, the one with the lower `integralityMeasure` will be placed higher in the heap, since it presumably is more likely to lead to an improved incumbent solution.

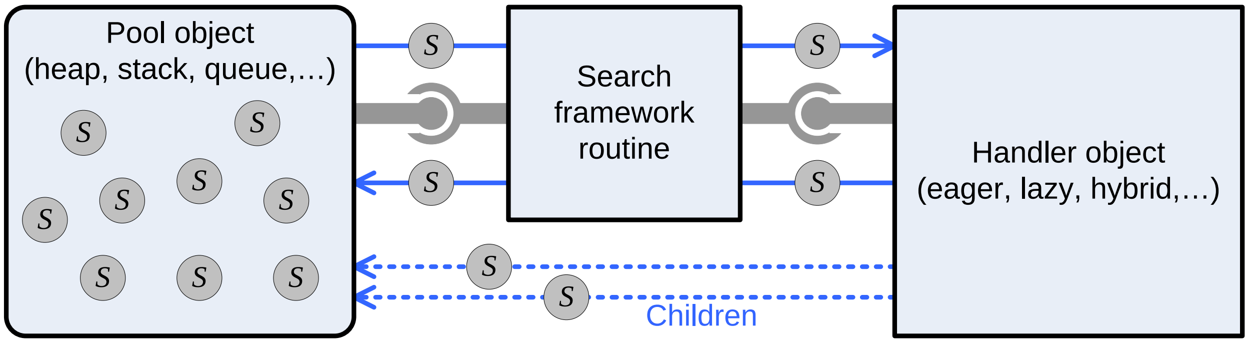

Figure 4: The search framework, pool, and handler. Each S indicates a branch-and-bound subproblem — an object of type derived from `branchSub`.

the pool, immediately separates it, and then forms and bounds each of its children. Children whose bounds do not cause them to be fathomed are returned to the pool.

Lazy and eager bounding each have their own advantages and disadvantages, and the best choice may depend on both the application and the implementation environment. Typically, implementors seek to postpone the most time-consuming operations in the hope that the discovery of a better incumbent solution will make them unnecessary. So, if the bounding operation is much more time-consuming than separation, lazy bounding is most appealing. If the bounding operation is very quick, but separation more difficult, then eager bounding would be more appropriate. Eager bounding may save some memory since leaf nodes of the search tree may be processed without entering the pool, but has a larger task granularity, resulting in lower parallel communication overhead per subproblem, but also somewhat less potential for parallelism.

Because PEBBL's serial layer stores subproblem states and lets the you specify a handler object, it gives you the freedom to specify lazy bounding, eager bounding, or other protocols. The search framework routine simply extracts subproblems from the pool and passes them to the handler until the pool becomes empty. Currently, there are three possible handlers, `eagerHandler`, `lazyHandler`, and `hybridHandler`. The `eagerHandler` and `lazyHandler` objects respectively implement eager and lazy bounding by trying to keep as many subproblems as possible in the `bounded` and `boundable` states, respectively.

The `hybridHandler` object implements a strategy that is somewhere between eager and lazy bounding, and is perhaps the most simple and natural given PEBBL's concept of subproblem states. Given any subproblem, `hybridHandler` performs a single application of either `boundComputation`, `splitComputation`, or `makeChild`, to try to advance the subproblem one transition through the state diagram of Figure 3. If the subproblem's state is `boundable` or `beingBounded`, it applies `boundComputation` once. If the subproblem's state is `bounded` or `beingSeparated`, it applies `splitComputation` once. Finally, if the state is `separated`, the handler performs one call to `makeChild`, and inserts the resulting subproblem into the pool. `Hybridhandler` is currently PEBBL's default choice of handler.

The combination of multiple handlers, multiple pool implementations, and the freedom in implementing `boundComputation` and `splitComputation` create considerable flexibility in the kinds of branch-and-bound methods that the serial layer can implement. Figure 4 depicts the relationship of the search framework, pool, and handler.

You may choose between the existing pools and handlers by setting parameters in the `branching` class object; see Section 5.10. You may also in principle supply your own pools and handlers, but we consider that an advanced topic, and it is not presently covered in this guide.

3.2 Controlling the Search Process

3.2.1 Basic tolerances

In its normal, non-enumeration mode of operation, PEBBL's optimality criteria are controlled by two parameters, `absTolerance` and `relTolerance`, with respective default values 0 and 10^{-7}. PEBBL attempts to locate a single problem solution that is either within an additive distance `absTolerance` or a relative distance `relTolerance` of optimality, whichever turns out to be less restrictive. For example, setting `relTolerance` to 0.05 would specify a solution with 5% of optimality.

3.2.2 Multiple solutions: enumeration criteria and the solution repository

PEBBL can also function in an enumeration mode in which it stores multiple feasible solutions to a problem. Enumeration mode is activated if any of the four parameters `enumAbsTolerance`, `enumRelTolerance`, `enumCutoff` or `enumCount` is set. The meaning of these parameters is as follows:

`enumAbsTolerance`: Find all solutions whose objective value is within `enumAbsTolerance` of the best possible. For example, setting `enumAbsTolerance` = 6 indicates you want all solutions within 6 units of optimal.

`enumRelTolerance`: Find all solutions within a relative distance `enumRelTolerance` of the best possible. For example, setting `enumRelTolerance` = 0.10 specifies that PEBBL should enumerate all solutions within 10% of optimal.

`enumCutoff`: Find all solutions with an objective value better than this value. For a minimization problem, for example, setting `enumCutoff` = 0 instructs PEBBL to find all solutions with a negative objective value.

`enumCount`: Find the best `enumCount` solutions; for example setting `enumCount` = 10 requires PEBBL to find the 10 best solutions.

If you set more than one of these parameters, PEBBL returns only solutions simultaneously meeting all the specified criteria. For example, setting `enumRelTolerance` = 0.05 and `enumCount` = 200 means that PEBBL should return only those solutions that are among the 200 best and are also within 5% of optimal.

The enumeration mechanism keeps a repository of feasible solutions represented as objects of type derived from the `solution` class, maintained both as a hash table and heap in reverse order of solution quality — that is, the solution on the top of the heap has the worst objective value. The hash table representation, along with hashing and comparison and methods of the `solution` class, prevents duplicate solutions from entering the repository. Discovery of a new best incumbent solution can cause solutions to be removed from the repository if either `enumAbsTolerance` or `enumRelTolerance` is set. If `enumCount` is set and the repository already contains `enumCount` solutions, then entry of a new solution into the repository causes the previously worst solution in the repository to be removed.

3.2.3 The `solution` class

PEBBL represents optimization problem solutions by objects derived from the abstract class `solution`. PEBBL provides a template class, `arraySolution<`*T*`>`, which represents solutions as a one-dimensional array or vector of type *T*. This solution representation should be adequate for many applications; if it is not convenient or efficient for a given application, users may create their own `solution`-derived classes. It is even possible for a single application to use several different solution representations.

To properly maintain the hash table used to filter out duplication solutions, PEBBL needs a hash function that may be applied to solutions, and a means of comparing two solutions to see if they are duplicates. The normal means of hashing and comparison are through the *sequence representation* of solutions. The `solution` class contains abstract methods `sequenceLength`, `sequenceData`, and `sequenceReset` whose purpose is to convert a solution to sequence of `doubles`, in such a way that two solutions may be considered duplicates if and only if invoking these methods results in identical behavior. If these methods are implemented, PEBBL automatically supplies a hash function in the method `computeHashValue`, and comparison operator for solutions is the method `duplicateOf`.

It is possible to dispense with the sequence representation methods; in this case, however, users must provide their own explicit implementations of `computeHashValue` and `duplicateOf`. Even if a sequence representation is defined, users might want to override either of these methods in the interest of efficiency.

3.2.4 Early output and checkpointing

Some of the calculations for which PEBBL is intended may be extremely long-running, and thus vulnerable to data loss if a system crashes or a job's time allocation is exhausted. PEBBL has two features designed to mitigate such data loss. The first, *early output*, tries to ensure that if a PEBBL run is interrupted, you can recover the best solution found so far. You enable this feature by setting the parameter `earlyOutputMinutes` to some positive value m. Each time PEBBL finds a new incumbent solution (that is, a solution better than all previously found ones) that exists for at least m minutes, PEBBL writes it to disk so that it is available if the run crashes. Early output is also useful if you wish to voluntarily terminate a run and still have access to the best feasible solution computed so far. This feature may be useful in early testing of an application for which you may not yet know appropriate tolerance values. For example, if you initiate a run with a `relTolerance` of 5%, but after considerable computation, the optimality gap is still 10%, you may decide you are satisfied with the 10% tolerance.

The second data loss mitigation feature is *checkpointing*, which imposes more overhead on PEBBL, but is more powerful. Checkpointing is currently available only for the parallel layer; it is not available in applications built solely on the serial layer. The key parameter controlling checkpointing is `checkPointMinutes`. If this parameter has a positive value m, PEBBL writes its complete internal state to disk approximately every m minutes. Each processor writes a separate file, and the checkpointing feature requires that all processors have access to file I/O, although not necessarily to the same directory.

The computation can be restarted from the time of the checkpoint by specifying either of the command-line parameters `restart` or `reconfigure`.[2] Using `restart` is potentially faster, but `reconfigure` allows one to restart with a different parallel configuration — for example, a different total number of processors. To use `reconfigure`, all checkpoint files must be in (or moved to) the same directory.

Some MPI implementations provide their own checkpointing capabilities. PEBBL's checkpointing feature is independent of any such capabilities and does not require them.

The early-output mechanism and enumeration mechanisms are currently independent. When using enumeration, the early output mechanism simply outputs the best incumbent. Checkpointing, however, is fully compatible with enumerating multiple solutions: each checkpoint will save the entire state of the solution repository.

3.3 Parallel layer architecture

PEBBL's parallel layer attempts to accelerate the branch-and-bound process by using multiple processors, and requires some form of the MPI message passing interface. PEBBL's parallel layer will run on shared-memory (SMP) systems, but only if MPI is configured to emulate a message passing environment (which is the default on multicore systems when packages such as OpenMPI are downloaded with utilities such as the Ubuntu package manager).

PEBBL's primary mode of parallelism, as is standard in parallel branch and bound, is to explore different nodes of the search tree simultaneously on different processors. However, PEBBL has the optional capability to use different modes of parallelism during the early stages of the search; see Section 3.3.5 below.

For the most part, parallel-layer search node processing is carried out by the same `boundComputation`, `splitComputation`, and `makeChild` methods as in serial layer. Thus, once you have created these methods for your application, parallel execution should be available with little additional development effort.

[2] As with all PEBBL command-line parameters, these parameters should be preceeded by two hyphens when used on the command line, as in `--restart`.

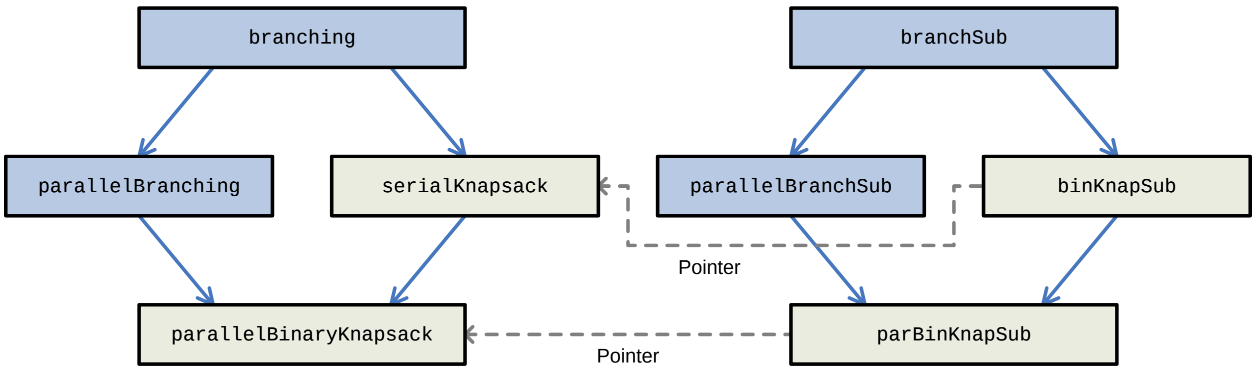

Figure 5: Partial depiction of the inheritance structure of the parallel knapsack application. Other parallel applications are similar.

3.3.1 Inheritance pattern

The parallel layer's capabilities are embodied in the classes `parallelBranching` and `parallelBranchSub`, which have the same function as `branching` and `branchSub`, respectively, except that they perform parallel search of the branch-and-bound tree. Both are derived from a common base class `parallelPebblBase`, whose function is similar to `pebblBase`, containing mainly common symbol definitions. The class `parallelBranching` also derives from `parallelPebblParams`, which contains a large number of parameters for controlling parallel search. Furthermore, each of `parallelBranching` and `parallelBranchSub` is derived from the corresponding class in the serial layer.

To turn a serial application into a parallel application, one must define two new classes. The first is derived from `parallelBranching` and the serial application global class. In the knapsack example, for instance, we defined a new class `parallelBinaryKnapsack` which has both `parallelBranching` and `binaryKnapsack` as `virtual` base classes. We call this class the *global parallel class*. For each problem instance, the information in the global parallel class is replicated once on every processor.

This basic inheritance pattern is repeated for parallel subproblem objects. In the knapsack case, we defined a parallel subproblem class `parBinKnapSub` to have `virtual public` base classes `binKnapSub` and `parallelBranchSub`. As with the serial subproblems, each instance of `parBinKnapSub` has a `parallelBinaryKnapsack` pointer that allows it to locate global problem information. Figure 5 depicts the inheritance structure for the parallel knapsack application. The classes `solution` and `binKnapSolution` are omitted from the figure since their relationship is unchanged from Figure 2, although a few additional `virtual` methods in `solution` may need to be implemented for parallel operation.

The header file `acro-pebbl/packages/pebbl/src/example/pebbl/parKnapsack.h` defines the inheritance structure depicted in Figure 5. It defines `parallelBinaryKnapsack` via

```
class parallelBinaryKnapsack :
      public parallelBranching,
      public binaryKnapsack
{
      ⋮
},
```

and parBinKnapSub via

```
class parBinKnapSub :
      public parallelBranchSub,
      public binKnapSub
{
private:
   parallelBinaryKnapsack* globalPtr;
public:
   parallelBinaryKnapsack* global() const { return globalPtr; };
   parallelBranching* pGlobal() const { return globalPtr; };
      ⋮
}
```

Once this basic inheritance pattern is established, the parallel application automatically combines the description of the application coming from the serial application (in the knapsack case, embodied in `binaryKnapsack` and `binKnapSub`) with the parallel search capabilities of the the parallel layer. For the parallel application to function, however, additional methods must be defined, as summarized in Section 4.3. The most critical of these methods are `pack` and `unpack`, which must be defined for the global parallel class, the subproblems, and solutions. These methods respectively describe how to encode and decode objects into a UTILIB `Packbuffer` or `UnPackBuffer` object (see the UTILIB documentation).

3.3.2 Processor clustering

PEBBL's parallel layer employs a generalized form of the processor organization used by the later versions of CMMIP [4, 5]. Processors are organized into *clusters*, each with one *hub* processor and one or more *worker* processors. The hub processor serves as a "master" in work-allocation decisions, whereas the workers are in some sense "slaves," doing the actual work of bounding and separating subproblems. The degree of control that the hub has over the workers may be varied by a number of run-time parameters, and may not be as tight as a classic "master-slave" system. Further, the hub processor has the option of simultaneously functioning as a worker.

Three run-time parameters, all defined in `parallelPebblParams`, govern the partitioning of processors into clusters: `clusterSize`, `numClusters`, and `hubsDontWorkSize`. First

PEBBL finds the size k of a "typical" cluster via the formula

$$k = \min\left\{\texttt{clusterSize}, \max\left\{\left\lfloor\frac{\overline{p}}{\texttt{numClusters}}\right\rfloor, 1\right\}\right\},$$

where $\overline{p}$ is the total number of processors. Thus, k is the smaller of the cluster sizes that would be dictated by `clusterSize` and `numClusters`. Processors are then gathered into clusters of size k, except that if k does not evenly divide $\overline{p}$, the last cluster will be of size $\overline{p}$ mod k. In clusters whose size is greater than or equal to `hubsDontWorkSize`, the hub processor is "pure," that is, it does not simultaneously function as a worker. In clusters smaller than `hubsDontWorkSize`, the hub processor is also a worker. The rationale for this arrangement is that, in very small clusters, the hub will be lightly loaded, and its spare CPU cycles should be used to help explore the branch-and-bound tree. If a cluster is too big, however, using the hub simultaneously as a worker may unacceptably delay the hub's response to messages, slowing down the entire cluster. In such cases, a "pure" hub is preferable.

3.3.3 Tokens and work distribution within a cluster

Unlike some "master-slave" implementations of branch and bound, each PEBBL worker maintains its own pool of active subproblems. This pool may be any of the kinds of pools described in Section 3.1.3, although all workers must use the same pool type. Depending on various parameter settings, however, the pool might be very small, in the most extreme case never holding more than one subproblem. Each worker processes its pool in the same general manner as the serial layer: it picks subproblems out of the pool and passes them to a search handler until the pool is empty. When running in parallel, handlers have the additional ability to *release* subproblems from the worker to the hub.

For the remainder of this subsection, assume for simplicity that a single cluster spans all available processors; in the next subsection, we will amend our description to cover the case of multiple clusters.

Random release of subproblems: When running in a parallel context, `eagerHandler` decides whether to release a subproblem as soon as it has become `bounded`. In parallel situations, `lazyHandler` and `hybridHandler` make the release decision when they create a subproblem. No matter which handler is used, the decision is a random one, with the probability of release controlled by run-time parameters. Released subproblems do not return to the local pool; instead, the worker cedes control over these subproblems to the hub. Eventually, the hub may send control of the subproblem back to the worker, or to another worker.

If the release probability is 100%, then every subproblem is released, and control of each subproblem is always returned to the hub at a some point in its lifetime (at creation for `lazyHandler` and `hybridHandler`, and upon reaching the `bounded` state for `eagerHandler`). In this case, the hub and its workers function like a standard "master-slave" system. When the probability is lower, the hub and its workers are less tightly coupled. The release probability is controlled by the run-time parameters `minScatterProb`, `targetScatterProb`, and `maxScatterProb`. The use of three different parameters, instead of a single one, allows the release probability to be sensitive to a worker's load. Basically, if the worker appears to have a fraction $1/w(c)$ of the total work in the cluster, $w(c)$ denotes the total number of workers in cluster c, then the worker uses the value `targetScatterProb`. If it appears to have less work, then a smaller value is used, but no smaller than `minScatterProb`; if it appears to have more work, it uses a larger value, but no larger than `maxScatterProb`.

Subproblem tokens: When a subproblem is released, only a small portion of its data, called a *token* [16, 4], is actually sent to the hub. The subproblem itself may move to a secondary pool, called the *server pool*, that resides on the worker. A token consists of only the information needed to identify a subproblem, locate it in the server pool, and schedule it for execution. Since the hub receives only tokens from its workers, as opposed to entire subproblems, these space savings translate into reduced storage requirements and communication load at the hub.

When making tokens to represent new, `boundable` subproblems, the parallel version of `lazyHandler` and `hybidHandler` take an extra shortcut. Instead of creating a new subproblem with `parallelMakeChild` and then making a token that points to it, they simply create a token pointing to the parent subproblem, with a special field, `whichChild`, set to indicate that the token is not for the subproblem itself, but for its children. Optionally, a single token can represent multiple children. If every child of a `separated` subproblem has been released, the subproblem is moved from the worker pool to the server pool.

Hub operation and hub-worker interaction: Workers that are not simultaneously functioning as hubs periodically send messages to their controlling hub processor. These messages contain blocks of released subproblem tokens, along with data about the workload in the worker's subproblem pool, and other miscellaneous status information.

The hub processor maintains a pool of subproblem tokens that it has received from workers. Again, this pool may be any one of the pools described in Section 3.1.3. Each time it learns of a change in workload status from one of its workers, the hub reevaluates the work distribution in the cluster. The hub tries to ensure that each worker has a sufficient quantity of subproblems, and optionally, that they are of sufficient quality (that is, with bounds sufficiently far from the incumbent). Quality balancing is controlled by the boolean parameter `qualityBalance`, which is `true` by default. Workload quantity evaluation is via the parameters `workerSPThreshHub` and `hubLoadFac`. A worker is judged "deserving" of work if the number of subproblems in its local pool is less than

$$\max\left\{\texttt{workerSPThreshHub}, \left(\frac{1-\texttt{hubLoadFac}}{W}\right)Q\right\},$$

where W is the number of workers and Q is the total number of known active subproblems. The logic behind the second term in the above maximum is that `hubLoadFac` specifies the fraction of active subproblems that are under hub control and constitute a "working set" used by the hub to balance workloads between workers. The remaining work should be roughly even distributed among the workers. This calculation is slightly modified when the same processor is serving as both a hub and a worker, since such processors may not be able to process subproblems quite as quickly as ordinary workers. PEBBL adaptively estimates the degree of "handicap" for such colocated worker-hubs as each run progresses.

If quality balancing is activated, a worker is also judged deserving if the best bound in its pool is worse than the best bound in the hub's pool by a factor exceeding the parameter `qualityBalanceFactor`. Of the workers that deserve work, the hub designates the one with fewest subproblems as being most deserving, unless this number exceeds `workerSPThreshHub`; in that case, the workers are ranked in reverse order of the best subproblem bound in their pools.

As long as there is a deserving worker and the hub's token pool is nonempty, the hub picks a subproblem token from its pool and sends it to the most deserving worker. The message sending the subproblem may not go directly to that worker, however; instead, it goes to the worker that originally released the subproblem. When that worker receives the token, it forwards the necessary subproblem information to the target worker, much as in [4, 5, 16]. This process will be described in more detail below.

When a single activation of the hub logic results in multiple dispatch messages to be sent from the hub to the same worker, the hub attempts to pack them, subject to an overall buffer length limit, into a single MPI message, saving system overhead.

If the subproblem release probability is set to 100%, and `workerSPThreshHub` is set to 1, the cluster will function like a classic master-slave system. The hub will control essentially all the active subproblems, and send them to workers whenever those workers become idle. Less extreme parameter settings will reduce the communication load substantially, however, at the cost of possibly greater deviation from the search order that would have been followed by a serial implementation. Also, setting `workerSPThreshHub` larger than 1 helps to reduce worker idleness by giving each worker a "buffer" of subproblems to keep it busy while messages are in transit or the hub is attending to other workers.

The best setting of the parameters controlling the degree of hub-worker communication depends on both the application and the hardware, and may require some tuning, but the scheme has the advantage of being highly flexible without any need for reimplementation or recompilation.

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

Rebalancing: If the probability of workers releasing their subproblems is set too low, or the search process is nearing completion, workers in a cluster may have workloads that are seriously out of balance, yet the hub's token pool may be empty. In this case, the hub has no work to send to underloaded workers. To prevent such difficulties, there is a secondary mechanism, called "rebalancing," by which workers can send subproblem tokens to the hub even if the original decision had been to keep the subproblems on the worker rather than release them. If a worker detects that it has a number of subproblems exceeding its target load in the cluster, it selects a block of subproblems in its local pool and releases them to the hub. The hub can then redistribute these subproblems to other workers.

3.3.4 Work distribution between clusters

With any system-application combination, there will be a limit to the cluster size that can operate efficiently, even if its hub does not have any worker responsibilities. To be able to use all the available processors, it may then be necessary to partition the system into multiple clusters.

PEBBL's method for distributing work between clusters resembles CMMIP's [4, 5], with some additional generality: there are two mechanisms for transferring work between clusters, *scattering* and *load balancing*. Scattering comes into play when subproblems are released by workers. If there are multiple clusters, the worker makes a supplementary random decision as to whether the subproblems should be released to the worker's own hub or to a cluster chosen at random. This random decision is controlled by the apparent workload of the cluster relative to the entire system, and the parameters `minNonLocalScatterProb`, `targetNonLocalScatterProb`, and `maxNonLocalScatterProb`. When choosing the cluster to scatter to, the probability of picking any particular cluster is proportional to the number of workers it contains (the worker's own cluster is not excluded).

To supplement scattering, PEBBL also uses a form of "rendezvous" load balancing that resembles CMMIP's [5]; [13] and [11] also contain earlier, synchronous applications of the same basic idea. This procedure also has the important side effect of gathering and distributing global information on the amount of work in the system, which in turn facilitates control of the scattering process, and is also critical to termination detection in the multi-hub case.

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

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

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 incumbent 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 exponent $\rho \in \{0,1,2,3\}$ is set by the parameter `loadMeasureDegree`. If $\rho = 0$, only the number of subproblems in the cluster matters. Higher values of ρ give progressively higher "weight" to subproblems farther from the incumbent. The default value of ρ is 1.

PEBBL redistributes work between clusters using a "rendezvous" scheme that organizes all the cluster hub processors into a balanced tree whose radix (branching factor) is determined by the parameter `loadBalTreeRadix`, with a default value of 2. The purpose of the tree is to organize messages between cluster hubs in a way that limits the communication load on any one processor, and the tree is not related to any hierarchy of control. Note that unlike ALPS [15], PEBBL does not employ a “master of masters” or “hub of hubs” processor, and its parallelization scheme is in principle indefinitely scalable because all cluster hubs interact in a peer-to-peer manner, but one that has a bounded communication load per-sweep, independent of the number of clusters.

Load-balancing messages pass up and down the tree of cluster hubs in a pattern consisting of a *survey sweeps* followed by *balance sweeps*. The frequency of these sweeps is controlled by a timer, with the minimum spacing between survey sweeps being set by a run-time parameter. If the total workload on the system appears to be zero, then this minimum spacing is not observed and sweeps are performed as rapidly as possible, to facilitate rapid termination detection. Under certain conditions, including at least once at the end of every run, a *termination check* sweep, which checks for completion of the computation, substitutes for the 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 this calculation was based on incumbent values that differed from processor to processor, it immediately repeats itself, a situation we call a *survey restart*.

At the end of a successful survey sweep, each hub determines whether its cluster should be a potential *donor* of work, a potential *receiver* of work, or (typically) neither. Donors are clusters whose workload exceeds the average by a factor of at least `loadBalDonorFac`, while receivers must be below the average by at least `loadBalReceiverFac`. Next, the balance sweep begins. A form of parallel prefix operation [1], this single up-and-down message sweep of the tree counts the total number of donors d and receivers r, assigns each donor a unique number in the range $0, \dots, d-1$, and assigns each receiver a unique number in the range $0, \dots, r-1$. The first $y = \min\{d, r\}$ donors and receivers then “pair up” via a rendezvous procedure involving $3y$ point-to-point messages. Specifically, donor i and receiver i each send a message to the hub for cluster i, for $i = 0, \dots, y-1$. Hub i then sends a message to donor i, telling it the processor number and load information for receiver i. See [9, Section 6.3] or [3, 5] for a more detailed description of this process. Within each pair, the donor sends a single message containing subproblem tokens to the receiver. Thus, the sweep messages are followed by up to $4y$ additional point-to-point messages, with at most 6 messages being sent or received by any single processor — this worst case occurs when a hub is both a donor and a rendezvous point. Both the survey and balancing sweeps involve at most $2(b+1)$ messages being sent or received at any given hub processor, where b is the branching factor of the load-balancing tree. Thus, the total number of messages per processor per round of load balancing is bounded above by the constant $2(b+1) + 2(b+1) + 6 = 4b + 10$. This kind of constant upper bound on the number of messages per processor required to perform a global operation is instrumental in designing scalable parallel algorithms.

Peer-to-peer load balancing mechanisms are frequently classified as either “work stealing,” that is, initiated by the receiver, or “work sharing,” that is, initiated by the donor. The rendezvous method is neither; instead, donors and receivers efficiently locate one another on an equal basis, possibly across a large collection of processors.

The load balancing scheme has an important secondary function of detecting termination, which can in general be challenging in highly asynchronous parallel programs. The general approach is a varient of the "four counters" technique proposed in [14], although only three counters are actually necessary, and the pattern of messages is adapted to PEBBL's clustered processor organization. In essence, PEBBL's survey sweeps can detect the situation that no cluster in the system appears to have any active subproblems, and the total count of messages sent matches the total count of messages received. While this situation is necessary for the calculation to be truly complete, it is not sufficient, since all the constituent processors cannot typically be sampled at the exact same moment in time. Therefore, and additional check is necessary to confirm termination. When there is only one hub, the termination procedure, while conceptually similar, is somewhat simpler.

3.3.5 Ramp-up: starting the parallel search

Ramp-up refers to the initial phase of a parallel search algorithm when the number of active search nodes is of a smaller order than the available processors. In some branch-and-bound applications, particularly when the number of processors is large, poor handling of ramp-up can have significantly reduce parallel efficiency.

If only one processor at a time can work on a given search node, the vast majority of processors will be idle during the initial development of the search tree. Often, this idleness is not a major issue, because the search tree grows quickly. However, in some applications, the root node of the tree, and possibly nodes near it, may take much longer to bound or separate than "typical" nodes later in the search. In such situations, the ramp-up phase prolonged and it may be hard to make efficient use of all available processors.

To help improve ramp-up performance, PEBBL supports a special ramp-up phase in which the application may exploit parallelism *within* each subproblem, if it is available. During the ramp-up phase, all processors synchronously explore exactly the same search nodes near the root of the branch-and-bound tree. That is, all processors will collectively process the root node, then all processors will collectively process the same child of that root node, and so forth. Each processor will thus redundantly have in its local memory exactly the same pool of active branch-and-bound nodes as every other processor. Instead of exploring parallelism inherent in exploration of the tree, the application attempts to exploit some source of parallelism within each search node, for example in calculating the bound or deciding how to separate a subproblem. This alternate mode of parallelism, however, is entirely the responsibility of the application.

During ramp-up, the method `rampingUp()`, available in both `parallelBranching` and `parallelBranchSub`, returns `true`; otherwise, it returns `false`. In response to the value returned by `rampingUp()`, `boundComputation`, `splitComputation`, and even `makeChild` may then attempt to explore some form of synchronous parallelism within the processing of individual search tree nodes. These methods are free to conduct MPI communication, but should be sure to leave all processors in a uniform state upon exit.

Ramp-up execution is controlled by two virtual methods, `continueRampup()` and `force`

`ContinueRampUp()`. When both these methods return `false`, PEBBL will terminate the ramp-up phase. PEBBL then automatically partitions the active search nodes, leaving each worker processor with an approximately equal number of active subproblems. PEBBL then begins its standard, asynchronous cluster/worker/hub search phase.

The default implementation of `continueRampup()` is controlled by two parameters, `rampUpPoolLimit` and `rampUpPoolLimitFac`. It returns `true` as long as the number of active subproblems does not exceed $\max\{\texttt{rampUpPoolLimit}, \overline{p} \cdot \texttt{rampUpPoolLimitFac}\}$, where $\overline{p}$ is the total number of processors. The default implementation of `forceContinueRampUp()` is to return `true` whenever the total number of subproblems created is does not exceed the parameter `minRampUpSubprobsCreated`. You may override these rudimentary implementations with implementations more specific to your application.

When the ramp-up phase ends, PEBBL evenly distributes the current active subproblems among the worker processors. Since all processors have the same tree in local memory at the end of the ramp-up phase, this distribution can be efficiently performed without any interprocessor communication. This process is called *crossover*.

3.3.6 Enumeration in parallel

PEBBL's parallel layer now supports the same multiple-solution enumeration criteria as its serial layer. For scalability, storage of the repository is partitioned approximately equally among all processors through a mapping based on the solution hash value — every solution s has a unique "owning" processor based on its hash value. Each processor has a local repository segment using the same hash-table-and-heap representation as the serial layer's single repository.

When enumeration is active and a solution s is passed to `foundSolution` on processor p_0, `foundSolution` immediately uses s's hash value to compute its owning processor $p(s)$. Unless $p(s) = p_0$, `foundSolution` immediately sends the s to processor $p(s)$. Using the hash table representation of the local portion of the repository and the `duplicateOf` method, processor $p(s)$ then checks if s is a duplicate of a solution already in its local segment of the repository. If so, it simply discards s. Otherwise, it enters s into the local repository. At the end of the run, PEBBL uses a synchronous parallel sort/merge algorithm to output a sorted list of all solutions found, ordered from best objective value to worst.

Unless `enumCount` is in use, this logic is essentially all that is needed. If `enumCount` is set, however, the implementation becomes more complicated. In that case, for proper pruning, PEBBL maintains an estimate of the $\texttt{enumCount}^{\text{th}}$-best solution in the union of the repository segments of all processors, called the *cutoff solution*. To keep track of the cutoff solution, PEBBL organizes all processors into a balanced tree. By periodically sending sorted arrays of solution values up this three, PEBBL computes the current cutoff solution value, which it then broadcasts down the tree. This technique is not totally scalable with respect to the value of `enumCount`, in that each processor potentially needs sufficient storage to hold `enumCount` solution values and identifier codes (but not entire solutions).

3.3.7 On-processor multitasking: threads and the scheduler

Once the ramp-up phase is over, the PEBBL parallel layer requires each processor to perform a certain degree of multitasking. PEBBL handles the multitasking through what it calls "threads", although they are not true threads, for example, in the POSIX sense: such true multithreading would be incompatible with some older MPI implementations and some specialized supercomputer node operating systems. Instead, PEBBL uses *non-preemptive* threads, essentially coroutines that voluntarily and periodically return control to a central scheduler module.

The PEBBL scheduler module recognizes two main types of threads: *message-triggered* and *compute* threads. Each message-triggered thread effectively "listens" for MPI messages with a specific tag value. If the scheduler detects a complete received message with the specified tag, it activates the thread to process the message (which may involve sending messages to other processors). The thread then returns to a dormant state until the scheduler detects the next message with its specified tag.

If there are no messages pending processing, the scheduler instead tries to activate the compute threads. Each compute thread has a *bias* which may be considered as a kind of priority. Among all compute threads that have declared themselves "ready", the scheduler tries to allocate CPU resources in proportion to the threads' bias values. Threads can adjust their bias values over time.

Figure 6 depicts PEBBL's standard threads. You may add additional threads of your own, but that is an advanced topic not currently covered in this guide. The standard threads are as follows:

Incumbent broadcast thread: This message-triggered thread is active on all processors. Its job is to make sure that all processors become aware, as soon as possible, of the objective value used to prune the search tree. It implements a form of asynchronous broadcast using a tree with an adjustable branching factor.

Early output thread: This message-triggered thread is active on all processors if the parameter `earlyOutputMinutes` is positive; otherwise it is absent. This thread coordinates the process of writing solution files, making sure the file is written by a processor that is priviledged to do I/O (the MPI standard specifies that not all processors, and perhaps only one processor, must have file and console output capabilities).

Repository receiver thread. This thread is active on all processors if PEBBL is enumerating multiple solutions. It handles the receipt of most messages involved in coordinating the individual repository segments on each processor into a global asynchronous repository.

Repository merger thread. This thread active is on all processors whenever the parameter `enumCount` is greater than 1. When `enumCount` is in use, certain messages required to maintain a global estimate of the cutoff solution may be sent on a time-delayed basis

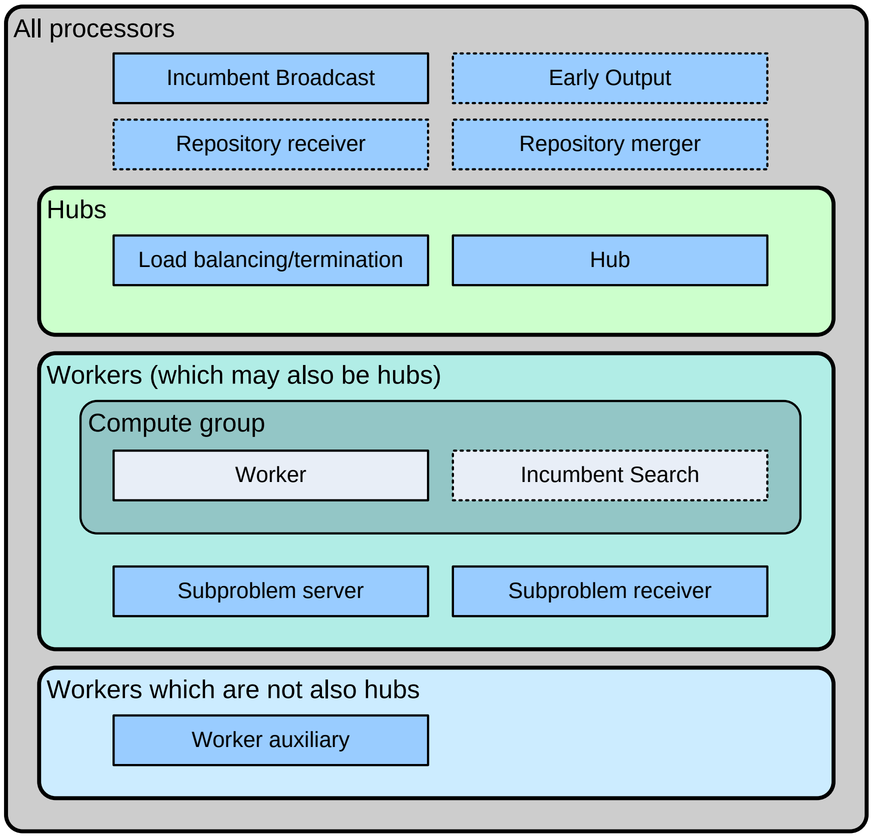

Figure 6: PEBBL's standard threads. A dashed outline indicates that the thread may not exist in some cases.

to limit peak message volumes; the repository merger thread handles this "throttled" message sending process.

Hub thread: This thread is active on all hub processors, and responds to messages from workers; these messages contain acknowledgements of received subproblems, tokens for released subproblem, and worker load information. Note that a hub processor's work distribution functions may also be activated in other situations besides receipt of one of these messages, via the `parallelBranching::activateHub()` method.

Load balancing/termination thread: This thread is active on all hub processors, and manages termination detection. It also controls checkpointing if `checkPointMinutes` is positive. When there is more than one cluster, it also manages the balancing of workload between clusters. It is generally message-triggered, but in multi-cluster situations it may also self-activate on some processors via the `ready()` predicate called by the scheduler.

Worker thread: This compute thread is active on all worker processors, and manages the processing of subproblems.

Incumbent heuristic thread: This optional compute thread may be active only on workers, and is controlled by the parameter `useIncumbentThread`. Its purpose is to heuristically search for improved incumbent solutions. Packaging this function into a compute thread allows PEBBL to directly control the fraction of CPU resources being dedicated to heuristic incumbent construction; the bias of this thread automatically adjusts based on the *relative gap*, the relative difference between the incumbent value and the best known bound among active search tree nodes. However, if the worker thread becomes idle (because it has no subproblems to process), the incumbent search thread will attempt to use all available CPU resources on the worker. This behavior can be useful near the beginning of a run if the number of subproblems at the end of the ramp-up phase is less than the total number of worker processors: instead of being totally idle, worker processors attempt to heuristically find incumbents until their worker threads receive subproblems.

Subproblem server thread: This message-triggered thread is active on all worker processors. It receives messages from the hub processors, and may in response send subproblems to other workers for processing.

Subproblem receiver thread: This message-triggered thread is active on all worker processors, receiving subproblems for processing.

Worker auxiliary thread: This message-triggered thread is active on workers that are not also functioning as hubs. The hubs send this thread various instructions, such as to write a checkpoint, check for termination, or terminate the search process. The worker auxiliary thread also periodially receives information relevant to the load balancing algorithms.

4 Creating PEBBL Applications

When using PEBBL, you should generally first create a serial PEBBL application and debug it using whatever C++ development tools you are most comfortable with. Then you should extend it to a parallel application using MPI.

4.1 Compiling and linking your application

In developing a new PEBBL application, you may either keep your source files outside the `acro-pebbl` directory tree, or incorporate them into the tree. If you keep your files outside, you should inspect the results of the PEBBL `make` command or the contents of `acro-pebbl/test/build.out`, and use compatible commands to compile and link your application source files. In particular, your compile commands' `#include` path should contain the directory `acro-pebbl/include`, and your link commands should include the libraries `acro-pebbl/lib/libpebbl.a` and `acro-pebbl/lib/libutilib.a`. In standard unix/Linux environments, for example, you could use the compiler/linker directives

```
-I[path/]acro-pebbl/include -L[path/]acro-pebbl/lib -lpebbl -lutilib
```

An alternative approach is to place your application source and header files within the PEBBL directory tree and use PEBBL's own compilation tools to compile your code. The recommended location for both your source and header files is `acro-pebbl/src/example/pebbl`. To incorporate your files into PEBBL's build process, you should proceed as follows:

1. Modify the file `acro-pebbl/src/Makefile.am` to include your new source files in a pattern similar to those for the `knapsack` application.
2. Incorporate any new header files you have created into the list contained in the file `acro-pebbl/src/headers/Makefile.am`.
3. Rerun the configuration and compilation procedure of Section 2.3. The `configure` step, guided by your altered `Makefile.am` files, will incorporate your application into PEBBL's build process.

4.2 Defining a serial application

To create a basic serial PEBBL application, you need to:

- Define a class extending `branching`
- Define a class extending `branchSub`
- If necessary, create a class extending `solution`
- Create a "driver" program that runs your algorithm.

You may concentrate your code for all these purposes in a single C++ file. Conventionally, however, you would use a header (`.h` or `.hpp`) file to define your new classes, one or more additional C++ source files containing code to implement methods for those classes, and a separate C++ source file containing the driver.

To load the basic definitions of `branching`, `branchSub`, and `solution` into the compiler, you should `#include` the file `<pebbl/branching.h>`. Supposing your classes are called *`myBranching`* and *`myBranchSub`*, and you are using a single header file, it should take the general form shown in Figure 7.

We now describe how to construct these classes; in the course of this description, you may also refer to the files `serialCore.h`, and `serialKnapsack.{h,cpp}` in the directory `acro-pebbl/packages/pebbl/src/example/pebbl`.

4.2.1 Methods you should create — `branching`-derived class

In your `branching`-derived derived class, for example *`myBranching`*, you should define the following methods:

Constructor

Classes derived from `branching` should have a constructor with no arguments; the diamond inheritance pattern used by PEBBL means that constructors with arguments are in general not advisable for `branching`- and `branchSub`-derived classes. The constructor should contain a call to `branching::branchingInit(···)`. If you want to maximize the objective function, you should supply `branchingInit` an argument of `maximization`; if you want to minimize, you can leave the argument list blank, or supply the argument `minimization`. Thus, one might have something like

```
myBranching()
{
   branchingInit(minimization);
      ⋮
};
```

The direction of minimization is stored in the data member `branching::sense`. The values `minimization` and `maximization` are C++ `enum` values, equal to +1 and −1, respectively.

Destructor

Naturally, you should also have a destructor for your `branching`-derived class, for example: `~myBranching() { ··· };`.

`branchSub* blankSub()`

You must provide a `blankSub()` method of return type `branchSub*` that returns an empty subproblem specific to your application. To avoid circularity, the class declaration for *`myBranching`* should declare this method, but not include its code, as in:

```
#include <pebbl/branching.h>

using namespace pebbl;

class myBranchSub; // Forward declaration

class myBranching :  virtual public branching
{
    friend class myBranchSub;

public:
      ⋮
    myBranching methods and data here
      ⋮
}

class myBranchSub :  virtual public branchSub
{
    friend class myBranching;

protected:
    // A pointer to the global branching object
    myBranching* globalPtr;
  ⋮
public:
    // Return a pointer to the global branching object
    myBranching* global() const { return globalPtr; }

    // Return a pointer to the base class of the global branching object
    branching* bGlobal() const { return global(); }
      ⋮
    myBranchSub methods and data here
      ⋮
}
```

Figure 7: Standard code pattern for creating a serial PEBBL application.

```
branchSub* blankSub();
```

Once *myBranching* and *myBranchSub* are both declared, you should have the actual code, for example:

```
branchSub* myBranching::blankSub()
{
   myBranchSub* newSP = new myBranchSub;
   newSP->setGlobalInfo(this);
   return newSP;
};
```

Here, *myBranchSub*`::setGlobalInfo` is a method that is required to be a member of the subproblem class (see Section 4.2.2 below), and copies any necessary information from the *myBranching* object into a *myBranchSub* object. At a minimum, it should set the subproblem's `globalPtr` member.

`bool setupProblem(int& argc,char**& argv)`

This method is responsible for reading in any input data describing the problem instance. Its arguments `argc` and `argv` are standard unix-style command line descriptors. However, by the time `setupProblem` is called, these arguments will be "cleaned" of any MPI-related information and parameter settings recognized by PEBBL. Typically, `setupProblem` should simply read a problem description from the file whose name is pointed to by `argv[1]`. The `setupProblem` method should return `true` is problem setup was successful.

`void reset(bool resetVB = true)`

Perform any initializations needed after `setupProblem` and before solving the actual problem. For some applications, the default implementation `branching::reset` may suffice. The optional argument `resetVB` indicates whether virtual base classes should be reset as well; it is present to allow one to avoid multiple redundant initializations of such bases classes. Typically, it can be ignored, although there may be a minor performance penalty. If you override the `reset` method, make sure your implementation invokes `branching::reset` in addition to any application-specific initializations it performs.

4.2.2 Methods you should create — `branchSub`-derived class

$\boxed{\text{Constructor}}$

As with the `branching`-derived class, you should have an empty-argument constructor for your `branchSub`-derived class, for example:

```
myBranchSub() { ··· };
```

$\boxed{\text{Destructor}}$

A destructor is naturally required, for example:

```
~myBranchSub() { ··· };
```

`branching* bGlobal() const`

This method is required for methods in `branchSub` to locate the corresponding problem-wide information in the corresponding `branching` object. Presuming you have declared a data member *myBranching* `* globalPtr` as in Figure 7, then Figure 7's definition should suffice:

```
branching* bGlobal() const { return global(); };
```

Your implementations of methods such as `boundComputation()` will almost certainly require access to problem-wide but application-specific information. In the implementation above, this may be obtained directly through `globalPtr` or slightly more elegantly via

```
myBranching* global() const { return globalPtr; };
```

`void setGlobalInfo(myBranching* global_)`

The purpose of this routine should be to "bind" a particular subproblem object to the problem description embodied in the object `*global_`. At a minimum, this requires setting `globalPtr`. For example:

```
void setGlobalInfo(myBranching* global_)
{
   globalPtr = global_;
   (Copy any other desired information from *global_ )
}
```

`void setRootComputation()`

A call to this method indicates that a *myBranchSub* object should be made into a root subproblem. Information on the problem instance should be obtained, for example, via `global()` or `globalPtr`.

`void boundComputation()`

This method should attempt to bound the subproblem. When the bound computation is complete, it should execute `setState(bounded)`. If the method exits without calling `setState(bounded)` to declare the subproblem `bounded`, PEBBL will assume that the bounding operation is incomplete: it will call `boundComputation()` repeatedly (but perhaps not immediately) until the subproblem is fathomed or declared `bounded`. To indicate the value of the bound, you set the `double` data member `bound`; you may update `bound` as many times as you like, and PEBBL will use the information immediately, even before the problem state becomes `bounded` — for example, a sequence of calls to `boundComputation()` may gradually improve the bound. To indicate infeasibility of a subproblem, or that it cannot possibly improve upon the current solution stored in `bGlobal()->incumbentValue`, you may execute `setState(dead)`.

`bool candidateSolution()`

PEBBL will only invoke this method for subproblems in the `bounded` state. Returning `true` means that the just-computed bound is *exact*: not only is it a bound on all solutions in the region of the search space corresponding to the subproblem, but that bound is also attained by at least one feasible solution in that region. Unless one is enumerating multiple solutions, such a subproblem is considered a *terminal* node of the branch-and-bound tree, and will not be separated. In integer programming, for example, you would want `candidateSolution()` to return `true` whenever the linear programming relaxation produces a solution that has all integer values. Returning `true` also indicates to PEBBL that you have computed a feasible solution to the overall optimization problem, as opposed to just a bound. PEBBL will most likely invoke `extractSolution()` (see below) to retrieve this solution.

`solution* extractSolution()`

PEBBL will only call this method for `bounded` subproblems for which `candidateSolution` has returned `true`. The method should return a pointer to a `solution`-derived object containing a solution corresponding to the subproblem bound. See Section 4.2.3 for more information on `solution` objects. Ownership of the returned object is ceded to PEBBL.

`int splitComputation()`

This method should attempt to separate the subproblem. Similarly to `boundComputation()`, it will be invoked repeatedly until it calls `setState(separated)`. The value returned is the number k of child problems generated (simply 2 for many applications). The number of children can vary between subproblems. If you return without declaring the problem `separated`, PEBBL will ignore the return value and invoke `splitComputation()` again later. In the course of separating the problem, you may update `bound` at any time to reflect any further information gained in the course of separation; PEBBL will use this information immediately. This routine may also execute `setState(dead)`, in which case PEBBL will immediately prune the subproblem.

If PEBBL is not in enumeration mode, it will call `splitComputation()` only for nodes for which `candidateSolution` has returned `false`. On the other hand, if PEBBL is enumerating multiple solutions, it is possible that `splitComputation()` may be called for nodes for which `candidateSolution` has returned `true`. If you need your application to be compatible with enumeration, it should be able to handle this case of separating an otherwise "terminal" subproblem. In some applications, such nodes may require a completely different separation technique than other nodes, and possibly a different number of children. Ideally, when separating a subproblem P for which `candidateSolution` has returned `true`, you should create one or more children corresponding to search-space regions which are disjoint and contain all solutions corresponding to P *except* the one just returned by P`.extractSolution()`. This latter solution will already be known to PEBBL and need not be examined again. If a node is "truly terminal" in that it corresponds to a subset of the search space containing only a single distinct feasible solution, then `splitComputation()` should call `setState(dead)` and return 0.

`branchSub* makeChild(int whichChild)`

This method should create a child problem and return a pointer to it. The argument `whichChild` may take any value between 0 and $k-1$, where k was the value returned by `splitComputation()` for this subproblem. A value of 0 indicates you should create the first child, a value of 1 indicates you should create the second child, and so forth.

When you create a child problem, you should make sure that its `globalPtr` is set as in the parent problem, and all local data are initialized correctly.

4.2.3 The `solution` class and related methods

PEBBL manipulates feasible problem solutions via objects whose type is derived from `solution`. For example, the `extractSolution()` method should return a `solution*`, as should the `initialGuess` method; see Section 4.2.5 below. The `solution` class itself contains (among others) the following members:

`double value`: the solution's objective value.

`int serial`: a serial number (assigned automatically by PEBBL).

`size_type hashValue`: a value for use in hash tables

It is possible to represent solutions via the `solution` base class; however, such a representation will essentially contain only the solution's objective value, and no other information. To construct such an object, use the constructor `solution(branching*)`, and then set the `value` of the resulting object. For example, for the code in the ***myBranchSub*** class of Figure 7, one would use

```
s = new solution(global());
s.value = objective value;
```

Far more likely, however, you will want a true solution representation that contains more information than just the objective value. In this case, you should use another class derived from `solution`. Here, there are two choices: use the template class `arraySolution<`*`T`*`>`, or derive your own `solution`-based class.

Note that each instance of the `solution` class contains its own `sense` member which must be set to `maximization` or `minimization` consistently with `branching`-derived object describing the problem instance (for example, *`myBranching`*). Typically, this is handled by passing a `branching*` pointer to the `solution` constructor.

The template `arraySolution<`*`T`*`>` is a ready-made PEBBL-supplied class that represents a solution as a one-dimensional array (that is, a vector) of some type *`T`* which may be implicitly converted to a `double` without loss of information. To create such a solution object, invoke a constructor (supplied by PEBBL) of the form `arraySolution<`*`T`*`>(`z`,`v`,`b`,`t`,`ν`)`, where:

z denotes the objective value of the solution

v is either a UTILIB array object of type `BasicArray<`*`T`*`>`, an STL vector of type `vector<`*`T`*`>`, or some object derived from either of those possibilities (for example, the UTILIB `IntVector` type derives from `BasicArray<int>`, and may thus be used to create an `arraySolution<int>`). This information is copied into the new `arraySolution` object.

b is a pointer to the `branching`-derived object describing the problem instance; within the class *`myBranchSub`* of Figure 7, you would set b = `global()`, and within the *`myBranching`* of Figure 7, you would set b = `this`. One function of this argument is to make sure that the maximization/minimization `sense` of the `arraySolution` is set correctly.

t is optional, and is simply a `const char*` pointing to a description of the type of solution, for example, `"Vector of integers"`.

ν is also optional, and is a pointer to a `BasicArray<CharString>` object containing names of the decision variables. It should be at least as long as v. These names are used when printing the solution contents.

For descriptions of the types `BasicArray` and `CharString`, see the UTILIB documentation.

If `arraySolution` does not apply to your problem, is inconvenient, or will be inefficient, you should derive your own class of objects from `solution`, as in:

`class `*`mySolution`*` : virtual public solution {` $\cdots$ `}` .

You may use more than one kind of solution class in a single application.

To facilitate hashing and comparison, solutions normally have a “sequence representation”, as described in Section 3.2.3. This representation is essentially a one-dimensonal array of `double`s, although it need not be (and probably should not be) explicitly stored. By default, PEBBL uses this sequence representation to compute a solution’s `hashValue` and to compare two solutions to detect duplicates. The solutions are considered duplicates if and only if their sequence representations are identical in both length and contents.

In addition to application-specific data members describing a solution, a customized solution class like ***mySolution*** should typically define the following `virtual` methods:

Constructor

To create the underlying `solution` object, your solution constructor should invoke the `solution(branching*)` constructor. It should also set the `value` member to reflect the solution's objective value. For the ***myBranching***, ***myBranchSub***, classes, for example, the standard constructor pattern would be

```
mySolution(myBranching* global_,
           [, other arguments]) :
   solution(global_),
   ...
{
   value = ···;
   [Set internal representation of solution]
}
```

Note that in the invocation of `solution(global_)`, the ***myBranching**** pointer `global_` is automatically cast to a base class pointer of type `branching*`.

Destructor

A destructor should be provided to avoid memory leaks.

`const char* typeDescription() const`

Returns a `const char*` describing the type of solution, for example `"Knapsack solution"`.

`virtual void printContents(std::ostream& s)`

Write the solution contents (for example the decision variable values) to the stream `s`. You do not need to write the objective value; it is printed by another method. The default implementation is a stub.

`virtual size_type sequenceLength()`

Return the number of elements in the sequence representation of the solution.

`virtual double sequenceData()`

Return the next element of the sequence representation.

The following method is related to `sequenceData()` and `sequenceLength()`, but may not have to be overridden:

`virtual void sequenceReset()`

Cause the next call to `sequenceData()` to return the first element in the sequence. The default implementation is simply to set the data member `sequenceCursor` (built into the `solution` class) to zero.

Note that PEBBL will call `sequenceData()` at most `sequenceLength()` times between calls to `sequenceReset()`. Typically, the default implementation of `sequenceReset()` will suffice, and each call to `sequenceData()` should return the `sequenceCursor`th element of the sequence and then increment `sequenceCursor`.

It is not absolutely necessary that a solution have a sequence representation, and thus not absolutely required that you define `sequenceLength()` and `sequenceData()`. If you do not define those methods, however, you must define two other methods:

`size_type computeHashValue()`

Return the hash value of the solution.

`bool duplicateOf(solution& other)`

Return `true` if `other` is an identical solution, and otherwise `false`.

Normally, it is recommended that you simply retain the default implementations of these methods, and define application-specific versions of `sequenceLength()`, `sequenceData()`, and possibly `sequenceReset()`. The default implementations of `computeHashValue()` and `duplicateOf` are based on the sequence representation. Even if you have defined a sequence representation, you may elect to override `computeHashValue()` and/or `duplicateOf` to improve efficiency.

4.2.4 The `foundSolution` method

So far, the only means we have discussed for providing solutions to PEBBL is the method `extractSolution` (see Section 4.2.2), which PEBBL calls between `boundComputation` and `splitComputation` for subproblems whose bounds are known to be exact. Many branch-and-bound algorithms encounter feasible solutions at other points in the search process. When this occurs, you may call the method `branching::foundSolution(solution* sol)`, where `sol` is a pointer to a `solution`-derived object. You need not first ascertain whether `*sol` is a better solution than the incumbent, or, when enumerating multiple solutions, whether it is eligible to enter the repository. PEBBL will determine whether the solution is useful, and otherwise delete it. Note that calling `foundSolution(sol)` cedes memory ownership of `*sol` to PEBBL.

It is safe to call `foundSolution` at any point within the branch-and-bound tree exploration, for example, from within the methods `boundComputation`, `splitComputation` and `makeChild`. For example, from the `boundComputation` method of the hypothetical *`myBranchSub`* class, you could invoke `foundSolution` via `global()->foundSolution(sol)`.

You should not call `foundSolution` at points before the branch-and-bound search has begun (for example, in the `setupProblem` method or the `preprocess` method described immediately below), or after the search has concluded. To provide a rough initial solution before the tree exploration process has started, use the `initialGuess` method described immediately below.

4.2.5 Selected additional methods

We now present an inexhaustive list of additional methods that you may wish to override for particular applications.

`void branching::preprocess()`

This method is intended for "preprocessing" of the problem prior to commencing the branch-and-bound search; the default implementation is an empty stub. In principle, its functions could be combined with `setupProblem(···)`, but it is provided for clarity and to ease migration to the parallel layer.

`solution* branching::initialGuess()`

PEBBL calls this method between preprocessing and the start of the branch and bound search. Its intention is allow computation of a "quick and dirty" initial guess at the solution. In a knapsack problem, for example, this routine could run a simple greedy heuristic. Your implementation need not be guaranteed to find a usable solution; if it does not, it should just return a `NULL` pointer. The default implementation is a stub that simply returns `NULL`.

`bool branching::haveIncumbentHeuristic()`

The default implementation of this method just returns `false`. You should change it to return `true` if your application has a way of trying to obtain a feasible solution from a non-terminal `bounded` subproblem.

`void branchSub::incumbentHeuristic()`

This method applies only to the serial layer, and PEBBL will only call it directly if the method `haveIncumbentHeuristic()` returns `true`. PEBBL calls it at two points, once a solution has become `bounded`, and once it has become `separated`. The intention is to allow the heuristic calculation of feasible solutions. Whenever your implementation finds a feasible solution, it should communicate it to PEBBL by calling `foundSolution`. The default implementation is a stub.

You may wish to check the subproblem state at the beginning of your implementation of `incumbentHeuristic`. For example, if you wish your heuristic to be invoked only when subproblems become `bounded`, and not when then become `separated`, you could begin your implementation with

```
if (state != bounded) return;
```

If you wish to construct heuristic solutions at other points besides when subproblems become `bounded` or `separated`, you can create your own heuristic method and invoke it at other times, having it call `foundSolution` to send new solutions to PEBBL.

Note that the parallel layer treats incumbent heuristics somewhat differently, and parallel-layer runs will not explicitly call `incumbentHeuristic`; see Section 4.3 below.

`void branchSub::makeCurrentEffect()`

PEBBL has the notion of a "current subproblem" – the one presently being bounded or separated. The method `makeCurrentEffect()` provides a "hook" that is called whenever a subproblem is made "current". In a branch-and-cut algorithm, for example, this method could load the subproblem's cuts into the linear programming solver. Its default implementation is a stub.

`void branchSub::noLongerCurrentEffect()`

PEBBL calls this "hook" whenever a subproblem ceases to be "current". Again, its default implemetation is a stub.

`bool branchSub::forceStayCurrent()`

PEBL calls this method when it is considering replacing the current subproblem with a different one. Returning `true` indicates that the current subproblem should be kept current if at all possible. The default implementation always returns `false`, meaning that PEBBL is free to "unload" the current subproblem, and replace it with a different one.

`bool branching::checkParameters()`

PEBBL calls this method to make sure that its runtime parameters are set correctly and consistently. You may override this method to perform additional checks specific to your application, and then call `branching::checkParameters()` to make sure PEBBL's internal checks are invoked. You are most likely to need this routine if you have defined your own application-specific runtime parameters; see the section immediately below.

4.2.6 Creating your own parameters

In addition to PEBBL's existing parameters, you may create your own command-line parameters. The simplest place to do so is in the constructor of your `branching`-derived class. Alternately, you can define a underlying base class to hold your parameters: you could create a class *`myParams`* to hold your parameters, and then derive *`myBranching`* from both `branching` and *`myParams`*. Parameter creation statements should usually reside in constructors and have the general form

```
create_categorized_parameter("commandLineName",
                             internalName,
                             "<datatype>",
                             "defaultValue",
                             "text description",
                             "category"[,]
                             [error check object])
```

Here,

commandLineName is the name to be recognized on the command line. For example, `fooBar` will cause the option `--fooBar=`*value* to be recognized on the command line.

internalName is a reference to a class data member in which to store the value of the parameter; usually, *internalName* and *commandLineName* should be identical for clarity.

datatype is the C++ datatype of *internalName*. Common choices are `bool`, `int`, `double`, and `string`.

defaultValue is a text representation of the default value for the parameter. It should match the initial value stored in the data member *internalName*. Note that the specification of *defaultValue* does not automatically initialize the value of *internalName*; its initial value should be set by the class constructor and `reset` methods.

text description is a parameter description printed in response to `--help`. For descriptions longer than one line, embed newline-tab character sequences ("`\n\t`") in this text.

category The category header to be used when printing the parameter description in response to `--help`. You may invent new categories.

error check object is optional and specifies simple constraints on the value of the parameter. For example,

```
ParameterNonnegative<int>()
```

specifies that an integer parameter must be nonnegative, while

```
ParameterBounds<double>(0.0,1.0)
```

specifies that a `double`-datatype parameter must have values between 0 and 1, inclusive. Other common forms for error-check objects are

```
   ParameterLowerBound<datatype>(minimum value)
or ParameterUpperBound<datatype>(maximum value)
```

For more details, refer to the UTILIB `Parameter` and related class documentation.

```
InitializeTiming();
myBranching instance;
bool flag = instance.setup(argc,argv);
if (flag)
   {
      instance.reset();
      instance.solve();
   }
return !flag;
```

Figure 8: Code equivalent to the serial `driver<`*`myBranching`*`>(argc,argv)` template.

For error checks that are more elaborate, or involve interactions between multiple parameters, you may specify parameter error checks by overloading the `checkParameters` boolean method in the `branching` class, for example with *`myBranching`*`::checkParameters`; if this routine detects an error, it should print a diagnostic message and return `false`. If it does not, it should execute `return branching::checkParameters()` to invoke the standard checks on PEBBL's built-in parameters.

Note that for parameters of type `bool`, you may omit `=`*value* on the command line, in which case the value is set to `true`. That is, `--fooBar` is equivalent to `--fooBar=true`

4.2.7 Serial main programs: drivers

PEBBL is structured as a callable library, and can thus be incorporated directly into other C++ applications. If you want your PEBBL application to function as a simple "solver" executable that simply reads in a problem instance data file and solves it, you need to create a "driver" main program. A simple template function called `driver` is available for this purpose. Using this template, you can construct the driver for the hypothetical example application *`myBranching`* as follows:

```
Header file #includes, including myBranching class and <pebbl/branching.h>

int main(int argc, char** argv)
{
    return driver<myBranching>(argc,argv);
}
```

Invoking `driver<`*`myBranching`*`>(argc,argv)` is equivalent to the code shown in Figure 8. Note that the main PEBBL header file `<pebbl/branching.h>` automatically includes the necessary UTILIB header files to define the method `InitializeTiming()`. Note also that `setup(···)`, `reset()`, and `solve()` are all methods of the `branching` class that you typically do not override. Calling `setup(argc,argv)` parses the command line, extracting and processing all command line arguments recognizable as PEBBL parameters, including any application-specific parameters created as described in Section 4.2.6. It then calls your implementation of `setupProblem`, with `argc` and `argv` adjusted so that parameter-setting arguments are removed; `setup`'s returning `true` indicates both that all command line parameters were processed correctly, and that your `setupProblem` implementation returned `true`, indicating it was successful. The method `solve()` invokes the branch-and-bound search engine, and prints subproblem count and timing statistics upon termination. It also uses your `solution`-derived class' `printContents` implementation to write the final solution(s) to a file whose name is derived from the first command line argument not recognizable as a parameter setting, typically by appending `sol.txt`; if no such parameter can be found, the file is called `sol.txt`). If you instead wish to specify your own choice for the solution file, use the `--output=`*filename* command-line option. Here, *filename* may contain path information, and thus need not be in the current directory.

The `setup` method manages parsing of the command line including option specifications and a problem instance data file name. Supposing your driver were called *myDriver*, the command line *myDriver datafile* would call *myBranching*`::setupProblem(argc,argv)` with `argv[1]` containing the null-terminated string *datafile* — with a typical implementation of `setupProblem`, this would cause your application to read in the problem instance description in *datafile*. In parsing the command line, `setup` will also recognize all PEBBL command-line parameters (see Section 5 for a partial catalog) along with any parameters defined for *myBranching*, as described in Section 4.2.6. For example, in the command line

```
myDriver --relTolerance=0.05 --earlyOutputMinutes=2 datafile
```

the PEBBL parameters `relTolerance` and `earlyOutputMinutes` would be set to 0.05 and 2, respectively. In this example, *myBranching*`::setupProblem(argc,argv)` would still be called with `argv[1]` containing the null-terminated string *datafile*, presumably reading the problem instance in *datafile*. The `setup` method also automatically recognizes several special command-line arguments:

`--help` Causes `setup` to print a usage line followed by a description of all available parameters, and then return `false` (so that the driver above would not attempt to solve a problem). If you want to control the form of the usage line, override the method `void branching::write_usage_info(char* progName,std::ostream& os)`. When PEBBL calls this method, the argument `progName` is set equal to `argv[0]`.

`--version` must be the first argument if present. It causes `setup` to print the value of the `static std::string` data member `branching::version_info` and return `false` (so that the driver above would then immediately exit). You may alter the contents of `version_info`, for example, in the constructor for your `branching`-derived class.

`--param-file=`*file* allows multiple parameter settings to be read from the file *file* — see the UTILIB Parameter class documentation for a description of the file format.

The `reset()` method invoked by the driver template performs all necessary initializations prior to the solution process, which is performed by `solve()`. The `solve()` method in turn invokes several lower-level methods, principally:

`search()` to perform the actual branch-and-bound search.

`printAllStatistics()` to print information about the number of search nodes and run time.

`solutionToFile()` to write the solution file.

4.2.8 "Quiet" operation

If PEBBL is embedded within another application, it may be desirable to have it run "quietly", without any configuration, status, or performance-statistics printouts. In this case, you should use the `search()` routine instead of `solve()` and make sure that solution status printouts are disabled by setting both `branching::statusPrintCount` and `statusPrintSeconds` to zero (see Section 5.6 for discussion of the corresponding runtime parameters).

4.2.9 Retrieving solutions to C++ code

Note that PEBBL applications can be invoked in ways other than the simple driver above; for example, they could be embedded in more complicated C++ programs, in which case you would probably use the `search()` method to perform the branch-and-bound search, and avoid command-line-oriented methods like `driver`, `setupProblem`, and `solve()`.

Various approaches are available to retrieve solutions computed by running the `search()` method; note that all the methods described in this subsection are members of the `branching` class. If you are not enumerating multiple solutions, you should simply use the method `getSolution()`, which returns a `solution*` pointer to the incumbent solution. This method returns `NULL` if no solution was found. Within your application, the `solution*` pointer returned by `getSolution()` will most likely need to be cast "up" the inheritance hierarchy so that its contents may be accessed, preferably using the C++ `dynamic_cast` operator.

For retrieving multiple solutions from a run enumerating multiple solutions, the simplest technique is to define an object *solArray* of type `BasicArray<solution*>` and then call the method `getAllSolutions(`*solArray*`)`. This method will redimension *solArray* to the size of the repository and fill it with `solution*` pointers to the repository members, in objective-value order, starting with the best value.

For compatibility with the parallel layer, a second, iterator-like technique is also possible. In this approach, one first calls `startRepositoryScan()`, which returns the size n of the repository. One may then call `nextRepositoryMember()` up to n times: each time, it will return a different member of the repository. Again, solutions are returned in objective value order, starting with the best values.

4.2.10 Debugging outputs

In the process of debugging a PEBBL application, it may be necessary to print diagnostic outputs. Through the companion UTILIB package that it automatically includes, PEBBL provides special mechanisms for such output. The most frequently used is the `DEBUGPR` macro, which takes the form

```
DEBUGPR(level,action);
```

Inside a class derived from `branching`, such a statement indicates that the code given by *`action`* be executed if the PEBBL debug output level specified by the run-time parameter `debug` in Section 5.2 has a value of *`level`* or higher. Typically, *`action`* should involve writing text to the specialized stream `ucout`. For example, the statement

```
DEBUGPR(10,ucout << "Value of n is " << n << endl);
```

will print the value of the variable `n` whenever the debug output level is at 10 or higher. While it makes not difference in the serial layer, using the special stream `ucout` is preferable to using the standard stream `cout` because `ucout` has special features that make it more convenient in a parallel setting. In particular, when used in parallel, each line of text sent to `ucout` will be prefixed by a tag indicating which processor wrote the information.

A slightly different form of debug printing is used in subproblem classes derived from `branchSub`, as opposed to problem classes derived from `branching`. In this setting, the diagnostic output above would instead be expressed as

```
DEBUGPRX(10,global(),ucout << "Value of n is " << n << endl);
```

The reason for this difference is that, for efficiency reasons, `branchSub` does not derive from the same UTILIB `CommonIO` utility class that `branching` does.

The `CommonIO` class from UTILIB makes several other variations on the `DEBUGPR` available for specialized situations. Consult the UTILIB documentation for more information.

An advantage of using the `DEBUGPR` and related facilities instead of *ad hoc* print statements is that the diagnostic output may be left in the code indefinitely and triggered whenever needed by using the `debug` runtime parameter. For example, running a PEBBL application with the command-line option `--debug=10` would set the debugging level to 10.

4.3 Defining a parallel application

Now suppose that your serial layer applications runs acceptably, and you wish to parallelize it. If you want to use parallelism, be sure that you have configured PEBBL with MPI; see Section 2.3, and especially Section 2.3.1. In particular, you should run `configure` with an option such as `--with-mpi-compilers=yes` or `--with-mpi-compilers=`*pathname*.

When Acro is compiled with MPI, the preprocessor symbol `ACRO_HAVE_MPI` is defined; otherwise it is not defined. When `ACRO_HAVE_MPI` is undefined, only the serial portions of

PEBBL are compiled; the parallel portions are omitted or become stubs. You may wish to follow this same pattern within your own application source code.

Figure 9 outlines the recommended inheritance and pointer pattern for creating parallel layer classes from a serial application. Here, the serial layer classes are *myBranching* and *myBranchSub*, and the corresponding respective parallel layer classes are *myParBranching* and *myParSub*. Note that the header file `<pebbl/parBranching.h>` defines all the classes in both the serial and parallel layers.

Note that PEBBL uses `parallelBranchSub::pGlobal()` to find the `parBranching` object associated with a subproblem object. The method `global()` in Figure 9 would be for your own use and might not be necessary if your `parBranching`-derived class does not encapsulate significantly more data than your `branching`-derived class.

Your `parBranching`-derived class may also contain additional parameters not present in your `branching`-derived class. The method for the creating such additional parameters is identical to that of Section 4.2.6, but the corresponding data members should be in your `parBranching`-derived class, for example *myParBranching*, and the necessary matching calls to `create_categorized_parameter` should be in that class' constructor.

4.3.1 Methods you should create — `parallelBranching`-derived class

Constructor

A constructor with no arguments is advised; it will automatically call the no-argument constructor for your serial application class, such as *myBranching*.

myParBranching() { ··· };

Destructor

You should also have a destructor:

~*myParBranching*() { ··· };

`void reset(bool VBflag = true)`

Perform all needed initializations subsequent to reading in the problem instance. This process consists of the following steps (not necessarily in the given order):

- Invoke the underlying serial problem-instance class `reset` method, for example by *myBranching*`::reset`.
- Register at least one reference solution; see Section 4.3.4.
- Invoke the `parBranching::reset` method.

```
#include <pebbl/parBranching.h>

using namespace pebbl;

class myParSub;     // Forward declaration

class myParBranching :
   virtual public parallelBranching,
   virtual public myBranching
{
 public:
      ⋮
   myParBranching methods and data here
      ⋮
};

class myParSub :
   virtual public parallelBranchSub,
   virtual public myBranchSub
{
 protected:
   // A pointer to the global parallel branching object
   myParBranching* globalPtr;
 public:
   // Return a pointer to the global branching object
   myParBranching* global() const { return globalPtr; }

   // Return a pointer to the parallel global base class object
   parallelBranching* pGlobal() const { return global(); }
      ⋮
   myParSub methods and data here
      ⋮
};
```

Figure 9: Standard code pattern for creating a parallel PEBBL application.

- Perform any additional application-specific initializations that are needed only by parallel runs.

The `VBflag` argument has the same purpose as the `resetVB` argument in serial `reset` methods. See Section 4.3.4 below for an example implementation of the parallel `reset` method.

`parallelBranchSub* blankParallelSub()`

This method is similar to `blankSub`, but should return a parallel subproblem with correctly initialized global pointers, as in the following code; see Section 4.3.2 below for a possible implementation of *myParSub*`::setGlobalInfo`.

```
parallelBranchSub* myBranching::blankParallelSub()
{
   myParSub* newSP = new myParSub;
   newSP->setGlobalInfo(this);
   return newSP;
};
```

`void pack(utilib::PackBuffer& outBuffer)`

PEBBL uses this method when broadcasting the problem description. It should write all the the information describing a problem instance (*i.e.* everything read by `setupProblem(···)`) into the UTILIB `PackBuffer` supplied in the argument `outBuffer`. Writing to a `PackBuffer` is very similar to writing to an unformatted stream: for most native C++ datatypes, the `<<` operator, that is, "`outBuffer <<` *data*" will write scalar data. Entire UTILIB arrays, for example of datatype `BasicArray<`*T*`>`, `IntVector` or `DoubleVector`, may also be written with `<<`, as can STL `sets` or `vectors` of standard C++ base types. See the UTILIB documentation for the details of `PackBuffers`.

`void unpack(utilib::UnPackBuffer& inBuffer)`

Again, PEBBL uses this method when broadcasting the problem description. Its job is to read from `inBuffer` the information written by `pack`. Most native C++ datatypes, along with many UTILIB-supplied containers and STL `sets` or `vectors` of simple types, may be read via the the `>>` operator, applied in exactly the same order as you used `<<` in `pack`. For example, data written by

```
outBuffer << time << money;
```

in `pack` might be read via

```
inBuffer >> time >> money;
```

in `unpack`.

`int spPackSize()`

This method should return an upper bound on the number of bytes needed to pack all the information to describe *a subproblem* — that is, the maximum amount of space needed by the method *myParSub*`::pack(···)` described below. It does *not* refer to the amount of space needed by *myParBranching*`::pack`, which PEBBL is able to detect automatically. In `PackBuffers` and `UnPackBuffers`, most C++ native datatypes require the same amount of space as their machine representations: *i.e* a data member of type *x* requires `sizeof(`*x*`)` bytes. You should use `sizeof(·)` in your implementation to make sure it is portable between 32- and 64-bit architectures and varying compilers. For standard container datatypes such as UTILIB arrays, STL `sets`, or STL `vectors` of type *T*, the space required is a single `sizeof(size_t)` to hold the size of the container, plus m · `sizeof(`*T*`)`, where m is the maximum container size possible for the given problem instance.

The `spPackSize()` method is called after the problem has been read and broadcast, so all information set by *myBranching*`::setupProblem` and *myParBranching*`::unpack` should be available to it in all processors.

Instead of direct calculation, an alternative approach to implementing `spPackSize()` is to create a "dummy" subproblem that is as large as possible given the current problem instance description, invoke your *myParSub*`::pack(···)` method to write it to a dummy buffer *dumBuf*, and then measure the size of that buffer with *dumBuf*`.size()`.

4.3.2 Methods you should create — `parallelBranchSub`-derived class

Constructor

Again, you need an empty-argument constructor, for example:

myParSub `() { ··· };`

Destructor

You also need a destructor:

~*myParSub* `() { ··· };`

`parallelBranching* pGlobal() const`

This method is similar to `bGlobal`, but returns a pointer of type `parallelBranching*`, implementing the third dashed arrow in Figure 5. Given `globalPtr` as defined in Figure 9, it could be implemented via

```
myParBranching* global() const { return globalPtr; }
parallelBranching* pGlobal() const { return global(); }
```

`void setGlobalInfo(myParBranching* global_)`

As in the serial, case, the purpose of this routine should be to "bind" a particular subproblem object to the problem instance description embodied in the object `global_`. It should also make sure that the corresponding serial-layer binding is also performed. For example:

```
void setGlobalInfo(myParBranching* global_)
{
   globalPtr = global_;
   myBranching::setGlobalInfo(global_); // Sets serial layer pointer etc.
   ⋮
}
```

`void pack(utilib::PackBuffer& outBuffer)`

This method should pack the application-specific portion of the description of the subproblem into the `PackBuffer` object `outBuffer`, typically using the `<<` operator. For more details about writing to `UnPackBuffer` objects, see the description of `branching::unpack` above.

`void unpack(utilib::UnPackBuffer& inBuffer)`

This method should unpack the application-specific portion of the description of the subproblem from the `UnPackBuffer` object `inBuffer`, typically using the `>>` operator. For more details about reading from `PackBuffer` objects, see the description of `branching::pack` above.

`virtual parallelBranchSub* makeParallelChild(int whichChild)`

This method is similar to `makeChild`, but returns a `parallelBranchSub*`. It should create the `whichChild`'th child of the present subproblem, counting from 0 to $k-1$, where k is the number of children. PEBBL only calls this method for subproblems in the `separated` state.

Note: `p->makeParallelChild(whichChild)` should *not* have the side-effect of changing the bound of `*p`, or memory errors may occur. This restriction may be removed in subsequent PEBBL releases.

4.3.3 Standard disambiguations

While the "diamond" inheritance pattern shown in Figure 1 is powerful, it may also lead to some ambiguities. When an inherited method is defined in several different base classes, the C++ compiler may be unsure which implementation to use. If such ambiguity occurs when compiling a parallel PEBBL application, the general rule is to use the implementation in either `parallelBranching` or `parallelBranchSub`, which will in turn automatically call the appropriate serial layer routine. In a class such as *myParBranching*, for example, it may be necessary to define

```
bool setup(int& argc,char**& argv)
   {
      return parallelBranching::setup(argc,argv);
   }
```

and similarly for a few other routines.

4.3.4 Parallel solution management and parallel `reset`

PEBBL does not need distinct serial and parallel versions of the class representing problem solutions; the same `solution` class fulfills this role in both serial and parallel settings. To function in a parallel settings, however, user-defined `solution`-derived classes require a few additional methods beyond those mentioned in Section 4.2.3. Specifically, you should define implementations of the following `virtual` methods:

$\boxed{\texttt{solution* blankClone()}}$

This method should return a new "blank" version of the current solution, that is, a (derived) solution object of the same type and dimensions. The blank copy should contain all information obtained or copied from your `branching`-derived class (for example, from the *myBranching* class). This information should include any pointers to the problem instance class, and the correct setting of `sense`. For all such information embedded in the `solution` base class, the constructor `solution(solution*)` can be useful; a standard implementation pattern might be

```
solution* blankClone() { return new mySolution(this); };

mySolution(mySolution* toCopy) :
   solution(toCopy),
   ··· other initializations ···
{   ···   };
```

It is not necessary for the object returned by `blankClone()` to contain any valid information about a particular solution to your problem instance.

$\boxed{\texttt{void packContents(PackBuffer\& outBuf)}}$

Pack the application-specific information in the current solution into the `PackBuffer` object `outBuf`, typically using the `>>` operator. For more details about writing to `PackBuffer` objects, see the description of `branching::pack` above.

$\boxed{\texttt{void unpackContents(UnPackBuffer\& inBuf)}}$

Unpack application-specific solution information from the `UnPackBuffer` object `inBuf`, typically using the `<<` operator. The order of unpacking should be identical to the order of packing in `packContents`. For more details about reading from `UnPackBuffer` objects, see the description of `branching::unpack` above.

`int maxContentsBufSize()`

This method should return an upper bound on the maximum number of bytes that calls to `packContents(outBuf)` might write to the buffer `outBuf` for a solution to the current problem instance. For more information about how to calculate this buffer size, see the discussion of the `parallelBranching::spPackSize()` method above.

PEBBL has its own methods for packing and unpacking information contained in the `solution` base class; `packContents` `unpackContents`, and `maxContentsBufSize()` do not need to take such information into account.

The interprocessor communication mechanism for solutions requires that you "register" a pointer to a "reference" object for each `solution`-derived class used by your application. It is recommended that you perform this operation in your parallel `reset` method, for example, *`myParBranching`*`::reset`. Reference solutions have three purposes:

- To calculate the maximum buffer size needed to receive a solution object: for each reference solution, PEBBL will call the method `maxContentsBufferSize`, and incorporate the results into its buffer sizing calculations.
- To create new solutions from messages: when a processor receives a message containing a solution, PEBBL calls the `blankClone` method for one of the reference solutions producing a new object s. PEBBL then reads the solution message into s by invoking s`.unpack`.
- To keep track of multiple solutions types: each kind of solution is assigned a `typeId` value, which is stored in the reference solution. When a processor receives a message containing a solution, this `typeId` value allows PEBBL to determine what kind of solution object it encodes.

Usually, a given application uses only one kind of solution, in which case the registration process is simple. You need only create a single solution object and pass it to the method `parallelBranching::registerFirstSolution(solution*)`. This object should be configured so that its `maxContentsBufSize()` method will return the correct value for the current problem instance. For example, if your application uses the `arraySolution<int>` solution representation and the maximum length of a solution is n, your *`myParBranching`*`::reset` would contain

`registerFirstSolution(new arraySolution<int>(`n`,this));`

The constructor `arraySolution<int>(`n`,this)` creates an array solution object of length n (of indeterminant contents); it can also accept optional arguments for a type name (*e.g.* `"My application solution"`), and an array of variable names. Constructing the reference solution with the maximum possible length n ensures that PEBBL's internal message buffers will be long enough for any message encountered.

If you use an application-specific `solution`-derived class, make sure your reference solution's `maxContentsBufferSize` method will return the maximum necessary space necessary to pack the solution's contents for the current problem instance, and its `blankClone` method will create properly formed solutions ready to have their `unpack` methods called on the results of any message `pack` operation. Assuming the *`mySolution`*`(`*`myBranching`*`*)` constructor of Section 4.2.3 behaves in the manner, a typical form for the parallel `reset` method is

```
void myParBranching::reset(bool vbFlag=true)
{
   myBranching::reset();
   registerFirstSolution(new mySolution(this));
   parallelBranching::reset();
       ⋮
   Any additional application-specific parallel-only initializations
       ⋮
}
```

If your application uses more than one type of solution representation, you must register a reference solution for each type. You should register the first with `registerFirstSolution`, and the rest with `registerSolution`, which also takes a single `solution*` argument. Equivalently, you can call `clearRegisteredSolutions()`, and then invoke `registerSolution` for each reference solution. Each solution type will be assigned an identifier of C++ type `size_type`, which is returned from the `registerFirstSolution` or `registerSolution` call, and is also stored in the corresponding reference solution's `typeId` member. Whenever you create a new solution, make sure its `typeId` member is set to the identifier corresponding to its type (this should happen automatically, for instance, if you create all new solutions by applying `blankClone` to the corresponding reference solution).

Note that calling `registerFirstSolution` or `registerSolution` cedes ownership of the argument to PEBBL.

In the parallel setting, you still make PEBBL aware of new solutions using the same `initialGuess`, `extractSolution`, and `foundSolution` methods as in serial applications. However, these methods have more complex behavior in the parallel layer and may now trigger various forms of interprocessor communication. If you are not using enumeration, the communication is typically a simple broadcast of a possible new incumbent value. If enumeration is active, then new solutions may be "hashed" to other processor to check for duplication and for incorporation into the global solution repository. The `foundSolution` method also has special behavior during the ramp-up phase, see Section 4.3.6 below.

4.3.5 Incumbent heuristics in parallel

PEBBL's parallel layer provides facilities allowing careful control over how much CPU time per processor is spent on incumbent heuristics. In the parallel setting, there are two possible levels of incumbent heuristic: a "quick" incumbent heuristic that is run for each bounded or separated subproblem, much as in the serial layer, and a separate incumbent thread whose CPU usage is controlled by the thread scheduler. We now consider the methods that implement this functionality. As in the serial layer, incumbent heuristics communicate solutions to PEBBL by calling `foundSolution`.

`void parallelBranchSub::quickIncumbentHeuristic()`

The heuristic to be run for every `bounded` or `separated` subproblem, similarly to the method `branching::incumbentHeuristic` in the serial layer. The default implementation is a stub. You need not attempt to truly run your heuristic every time `quickIncumbentHeuristic()` is called. For example, your implementation could immediately return if the subproblem has the wrong state, or does not look particularly "attractive". If you do run your heuristic and find an improved incumbent, you should call `updateIncumbent()` or perform a similar sequence of operations. The process of calling `quickIncumbentHeuristic()` is separate from the heuristic thread and does not require existence of the heuristic thread. If you wish the incumbent heuristic calculations to be identical in the serial and parallel layers, make sure `hasParallelIncumbentHeuristic()` (see immediately below) returns `false`, and implement `quickIncumbentHeuristic` as follows:

```
void myParSub::quickIncumbentHeuristic()
    { myBranchSub::incumbentHeuristic(); } ;
```

`bool parallelBranching::hasParallelIncumbentHeuristic()`

Returns `false` by default. Return `true` if your implementation has the capability to run a heuristic thread. PEBBL will create a heuristic thread if this method returns `true` and the parameter `useIncumbentThread` is `true`.

`void parallelBranchSub::feedToIncumbentThread()`

This method is a "hook" called for each bounded subproblem. It is intended to examine a subproblem, and if it seems sufficiently attractive, copy some representation of it to the data structures used by the routine `parallelIncumbentHeuristic`. In creating these data structures, you may want to make them able to store representations of more than one subproblem.

`void parallelIncumbentHeuristic(double* controlParam)`

This is the method invoked by the incumbent thread. The argument `controlParam` is set by the scheduler to try to control the amount of CPU time each call uses. If you wish, you may ignore the value of this argument, and simply set `*controlParam = 1` upon exit. If you wish to be more responsive to the scheduler, try to do an amount of work roughly proportional (in some sense of your own choosing) to `*controlParam`, and then set `*controlParam` equal to the amount of work performed.

`ThreadObj::ThreadState incumbentHeuristicState()`

This method indicates whether the incumbent thread is ready to run. The default implementation is to always return `ThreadObj::ThreadBlocked`, indicating the thread is unable to run. As soon as your thread has some data upon which to operate, you should return `ThreadObj::ThreadReady` instead.

`double incumbentThreadBias()`

This method indicates the importance of running the incumbent heuristic relative to the regular branch-and-bound worker process. The default implementation uses a formula involving various standard parameters and the current relative gap between the best known search node and the incumbent; see Section 5.8. You are free to override this method with something more specific to your application, but the details are omitted here.

4.3.6 The ramp-up phase and parallel preprocessing

Section 3.3.5 describes how PEBBL can take advantage of non-tree parallelism during the early growth of the search tree. During the ramp-up phase, the methods `boundComputation`, `splitComputation`, and `makeChild` are called synchronously on identical subproblems for all processors. In your ***myParSub*** class, you may further override the implementations of these methods in your ***myBranchSub*** class, so they can exploit synchronous parallelism during ramp-up. The method `parallelBranchSub::rampingUp()` will return `true` during the ramp-up phase, and `false` otherwise.

Just before the ramp-up phase starts, but after the problem description has been broadcast, all processors invoke `preprocess()` and `initialGuess()`. The default behavior of `preprocess` is to just have all processors redundantly call the serial `preprocess()` method for the application. If you override `preprocess()` in your parallel-layer classes, however, you may substitute a different computation, perhaps parallelizing your preprocessing calculations.

A common source of errors during the ramp-up phase is processors not being sufficiently synchronized. If, for example, two processors do not agree on the value of the incumbent, they may start following different computational paths, eventually resulting in an MPI deadlock. At any point in your implementation of ramp-up that you could change the incumbent in a way that might not be identical for all processors, you should call

```
parallelBranchSub::rampUpIncumbentSync()
```

before taking any action that could lead to a different pattern of search tree exploration or any variations in MPI routine calls. This method forcibly synchronizes the incumbent and (if enumeration is active) the solution repository. In parallel settings, PEBBL automatically calls `rampUpIncumbentSync()` immediately after `initialGuess`, so your `initialGuess` routine may return different solutions in different processors without any special subsequent action. PEBBL will select one of the best solutions as the incumbent; if enumeration is active, other solutions may be retained in the repository.

The `foundSolution` method has a second, optional argument `sync` which is relevant only during the ramp-up phase, and only if enumeration is active. Its default value is the `enum` constant `notSynchronous`. Instead supplying the value `synchronous` indicates to PEBBL that you are guaranteeing that all processors are calling the routine at the same time, with the first argument set to identical solutions. This information allows PEBBL to reduce the overhead associated with maintaining the solution repository.

The following methods are useful in controlling the ramp-up phase.

`void parallelBranchSub::rampUpIncumbentHeuristic()`

This method substitutes for the usual `quickIncumbentHeuristic()` during the ramp-up phase. The default implementation of `rampUpIncumbentHeuristic()` is:

```
if (bGlobal()->haveIncumbentHeuristic())
   {
      incumbentHeuristic();
      pGlobal()->rampUpIncumbentSync();
   }
```

The effect of this code is to run the *serial* incumbent heuristic and then synchronize across processors. If you like, you may have your implementation of `rampUpIncumbentHeuristic()` call `feedToIncumbentThread()`, even though the incumbent heuristic thread will not be running yet. This practice allows the incumbent thread on each processor to be “primed” with information found during the ramp-up phase, so it may immediately begin doing useful work once ramp up ends.

`bool parallelBranching::continueRampUp()`

Ramp up will continue as long as either this method or `forceContinueRampUp()` return `true`. The default implementation is described in Section 3.3.5, but you are free to override it.

`bool forceContinueRampUp()`

Ramp up will continue as long as either this method or `continueRampUp()` return `true`. The default implementation is described in Section 3.3.5, but you are free to override it.

`void rampUpCleanUp()`

PEBBL calls this method on all processors when the ramp-up phase is over. The default implementation is a stub.

4.3.7 Providing for checkpoints

If your application maintains data structures not written and read by your implementations of the methods `*myParBranching*::pack`, `*myParBranching*::unpack`, `*myParSub*::pack`, and `*myParSub*::unpack`, then PEBBL's checkpointing feature will neither save nor restore them. To make sure any such information is saved with each checkpoint and restored whenever restarting from a checkpoint, you need to define a few extra methods:

`void *myParBranching*::appCheckpointWrite(PackBuffer& outBuf)`

Write application-specific data to the `PackBuffer outBuf`, typically using the `<<` operator.

This method will be called separately for each processor and for each checkpoint. The default implementation is a stub.

`void` *`myParBranching`*`::appCheckpointRead(UnPackBuffer& inBuf)`

Read the information written by `appCheckpointWrite` from `inBuf`, typically using the `>>` operator. This routine will be called on each processor when a checkpoint is read using the `restart` option. The default implementation is a stub.

`void` *`myParBranching`*`::appMergeGlobalData(UnPackBuffer& inBuf)`

This method is similar to the `appCheckpointRead` routine, but invoked when restarting with `--reconfigure`. It will be called on every processor, but multiple times — once for each dataset written by `appCheckpointWrite` when the checkpoint was created. When restarting with `--reconfigure`, the numbers of worker and hub processors may be different from when the checkpoint was written.

4.3.8 The main program: serial/parallel drivers

PEBBL provides a template that constructs a driver program similar to that of Section 4.2.7, but able to sense and use available parallelism. If PEBBL senses that more than one MPI processor is available or if the command-line option `--forceParallel` is present, it invokes the parallel-layer version of your application; otherwise, it invokes the serial-layer version. If it invokes the serial-layer version, only the command-line parameters defined for that version will be recognized; if you have defined additional parameters in your parallel-layer classes, they will only be recognized if that layer is invoked.

The serial/parallel driver template is also called `driver`, and its syntax is nearly identical to the serial `driver` template, but with two template arguments `<`*`B`*`,`*`PB`*`>`. *`B`* refers to your serial `branching`-derived problem-instance class, and *`PB`* refers to your `parallelBranching`-derived class. For example, you could the template invoke via

`driver<`*`myBranching`*`,`*`myParBranching`*`>(argc,argv)`

If you have configured PEBBL without MPI, then the two-argument form of `driver` is still defined, but the second template argument *`PB`* is ignored. Thus, you can use a single driver program for your application whether or not you have configured PEBBL to use MPI. Once it has detected whether multiple MPI processors are available and whether `--forceParallel` was used, the `driver` template invokes, in both the serial and parallel cases, essentially the same `setup-reset-solve` code pattern as described in Section 4.2.7.

The standard code pattern for creating a serial/parallel driver is shown in Figure 10. The `#ifndef` block is needed if you have enclosed the definition of *`myParBranching`* within an `#ifdef ACRO_HAVE_MPI` block. In this case, even though the second template argument *`myParBranching`* will be ignored in the instantiation of `driver`, one must make *`myParBranching`* a defined type to guard against possible compiler errors; with some compilers, this safeguard may not be needed.

```
Include application-specific header files, for instance myParBranching.h
Include <pebbl/parBranching.h>, unless already included in preceding headers

#ifndef ACRO_HAVE_MPI
typedef void myParBranching;
#endif

int main(int argc, char* argv[])
{
   return driver<myBranching,myParBranching>(argc,argv);
}
```

Figure 10: Standard code pattern for creating a serial/parallel driver program.

Depending on your MPI environment, your serial/parallel driver may have to be invoked in a particular way in order to detect the availability of parallelism. The most common such invocation procedure involves the `mpirun` command, as specified in the MPI standard. For example,

```
mpirun -np 4 myParDriver --useIncumbentThread=false datafile
```

would run *myParDriver* on four processors, without an incumbent heuristic thread, and with the input file *datafile*. See Section 5.8 for a description of the `useIncumbentThread` option.

Note, however, that some installations may use different, nonstandard commands for running MPI programs.

4.3.9 Retrieving solutions to C++ code

As in the serial case, you may invoke PEBBL search as a subroutine from your own parallel C++ applications. The methods available for retrieving solutions are identical to the serial layer methods described is Section 4.2.9 — that is, `getSolution`, `getAllSolutions`, `startRepositoryScan`, and `nextRepositoryMember` — but the parallel layer redefines them to work properly in parallel. It is critical that these routines be called synchronously on all processors, or the PEBBL application will probably hang.

The solution-retrieval methods `getSolution`, `getAllSolutions`, `startRepositoryScan`, and `nextRepositoryMember` have an optional argument that is ignored in the serial layer, specifying which processor(s) the result(s) should be returned to. The default for this argument is the special value `pebblBase::allProcessors`, specifying that results be returned to all processors through a broadcast operation. Otherwise, the argument should be set to an integer k in the range $0 \leq k <$ `uMPI::rank` specifying the particular processor that is

```
        int nSols = instance.startRepositoryScan();
        for(int i=0; i<nSols; i++)
           {
              solution* solPtr = instance.nextRepositoryMember(0);
              if (uMPI::rank == 0)
                 {
                    ucout << solPtr->value << endl;
                    solPtr->deleteIfNotLocal();
                 }
           }
```

Figure 11: Example of scanning the solution repository in the parallel layer.

to receive the solution(s). On other processors, `getAllSolutions` will leave its argument unmodified, and `getSolution` and `nextRepositoryMember` will return `NULL`.

If the repository is very large, it is possible that it will fit in the memory of the entire processor ensemble, but not in the memory of a single processor. In this case, using `getAllSolutions` will cause out-of-memory errors, and it is preferable to use the iterator-like methods `startRepositoryScan` and `nextRepositoryMember`, so that the entire repository need not reside simultaneously within a single processor.

To avoid memory leaks and unnecessary duplication of solutions, any code that loops over repository members should call the `solution` class method `deleteIfNotLocal` as soon as it is finished processing each particular solution. This method will delete solutions that are "owned" by other processors, but preserve the integrity of the repository by not deleting solutions whose "master" repository member resides on the current processor.

As an example, assuming that *`instance`* is an object wose type is derived from the `parallelBranching` class, and that the `search()` method has already run to completion, the code in Figure 11 sends every solution in the repository to processor 0, which prints their objective values:

5 Parameters

This section describes various command line parameters that PEBBL recognizes. The listing is not exhaustive, but contains the parameters you should typically find most useful. A complete listing is produced by specifying the `help` parameter to driver programs constructed in the manner described in this guide. You may add your own application parameters as described in Section 4.2.5.

5.1 Checkpointing

`abortCheckpointCount`

Layer:	Parallel only
Datatype:	`int`
Default value:	0
Constraints:	Nonnegative

Primarily for debugging purposes. Causes an abort after writing this many checkpoints. A zero value, which is the default, disables this feature.

`checkpointDir`

Layer:	Parallel only
Datatype:	`string`
Default value:	Current directory, or from environment variable

Directory to place checkpoint files. The environment variable `PEBBL_CHECKPOINT_DIR`, if defined, provides a default value. If this variable is undefined, the default is the process current directory.

`checkpointMinInterval`

Layer:	Parallel only
Datatype:	`double`
Default value:	0
Constraints:	Nonnegative

Minimum minutes of CPU time per processor between writing checkpoints.

`checkpointMinutes`

Layer:	Parallel only
Datatype:	`double`
Default value:	0
Constraints:	Nonnegative

Desired minutes between starting to write successive checkpoints; the default value of 0 disables checkpointing.

`reconfigure`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Resume from a previously written checkpoint, reading the checkpoint files serially. The configuration of worker and hub processors need not be identical to the run that wrote the checkpoint.

`restart`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Restart from a previously saved checkpoint, attempting to read the checkpoint files in parallel. The configuration of worker and hub processors must be identical to the run that wrote the checkpoint.

5.2 Debugging and performance tuning aids

`debug`

Layer:	Serial and parallel
Datatype:	`int`
Default value:	0
Constraints	Nonnegative

Debugging diagnostic output level. To add diagnostic output controlled by this parameter to your application, you may use the `DEBUGPR` and related macros.

`debug-solver-params`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

If `true`, print the value of all parameters.

`debugSeqDigits`

Layer:	Parallel only
Datatype:	`int`
Default value:	0
Constraints:	Lower bound: 0, Upper bound: 10

Number of sequence digits prepended to output lines. This feature allows you to run debug output through the unix `sort` utility to obtain output grouped by processor. Another way to separate debugging output by processor is to use the `dumpSplit` utility described below in Section 6.1.

`forceParallel`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Force the use of a parallel PEBBL solver, even if there is only one processor.

`hubDebug`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Supplies a specific debug level for hub operations.

`loadLogSeconds`

Layer:	Parallel only
Datatype:	`double`
Default value:	0

If positive, triggers output of load logging information suitable for visualization with the `pebblLoadGraph` utility described in Section 6.2. A positive value indicates the approximate number of seconds that should elapse between load information samples on each processor. For example, a value of 0.1 indicates that each processor should record load information 10 times per second.

`printDepth`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Include subproblem depth in debugging output.

`printIntMeasure`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Include subproblem “integrality measures” in debugging output.

`workerDebug`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Supplies a specific debug level for the worker thread and related operations.

5.3 Enumeration

$\boxed{\texttt{enumAbsTolerance}}$

Layer:	Serial and parallel
Datatype:	`double`
Default value:	-1.0
Constraints	Lower bound: -1

Absolute tolerance for enumeration. Find solutions that are within this additive distance of optimality. The default value means the feature should not be used.

$\boxed{\texttt{enumCutoff}}$

Layer:	Serial and parallel
Datatype:	`double`
Default value:	(none)

If set, specifies an absolute threshold on enumerated solutions. Only solutions with values strictly better than `enumCutoff` will be retained.

$\boxed{\texttt{enumCount}}$

Layer:	Serial and parallel
Datatype:	`int`
Default value:	0
Constraints	Nonnegative

If positive, indicates that only the best `enumCount` solutions (that meet other enumeration criteria, if any) should be retained.

$\boxed{\texttt{enumRelTolerance}}$

Layer:	Serial and parallel
Datatype:	`double`
Default value:	-1.0
Constraints	Lower bound: -1

Relative tolerance for enumeration. Find solutions that are within this multiplicative factor of being optimal. For example, a value of 0.1 requests solutions within 10% of optimality. The default value means the feature is disabled.

5.4 General

$\boxed{\texttt{help}}$

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

If `true`, print usage information and parameter definitions, and then exit.

`randomSeed`

Layer:	Serial and parallel
Datatype:	`int`
Default value:	1
Constraints	nonnegative

Global seed for random number generation.

`version`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

If `true`, print version information and exit. Should be used on the command line only, and as the first parameter specified.

5.5 Incumbent

`startIncumbent`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	(none)

Value of some known feasible solution.

5.6 Output

`earlyOutputMinutes`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	0
Constraints	Nonnegative

If this many minutes have elapsed since its creation, output the current incumbent to a file in case of a crash or timeout. The default value disables this feature.

`output`

Layer:	Serial and parallel
Datatype:	`CharString`
Default value:	See Section 4.2.7

File in which to write the final solution(s). If omitted, output is written in the current directory, using a filename constructed as described in Section 4.2.7.

`printFullSolution`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Print full solution(s) to standard output as well as writing it to a file.

`printSolutionSynch`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`true`

Indicates that only MPI's designated I/O processor (typically processor 0) is allowed to write the solution. If `false`, some communication overhead may be saved in the solution output process.

`statusPrintCount`

Layer:	Serial and parallel
Datatype:	`int`
Default value:	100,000
Constraints	Nonnegative

The maximum number of subproblems bounded between status printouts. Status printouts are triggered by thresholds on wall clock time and the total number of subproblems bounded since the last status printout. Since the default value is large, the default behavior will typically be to trigger status printouts based on wall clock time.

`statusPrintSeconds`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	10.0
Constraints	Nonnegative

The maximum number of seconds elapsing between status printouts.

`suppressWarnings`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Suppress warning messages.

`trackIncumbent`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`false`

Print a message whenever there is a new incumbent.

5.7 Parallel work distribution

`clusterSize`

Layer:	Parallel only
Datatype:	`int`
Default value:	64
Constraints:	Lower bound: 1

Maximum number of processors controlled by a single hub (including the hub itself). Unless `numClusters` is set, this will be the size of all but the last cluster.

`hubsDontWorkSize`

Layer:	Parallel only
Datatype:	`int`
Default value:	10
Constraints:	Lower bound: 2

Size of cluster at or above which hubs do not also function as workers.

`hubLoadFac`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.10
Constraints:	Lower bound: 0 , Upper bound: 1

The target fraction of subproblems to be controlled by hub processors at any given time. See Section 3.3.3 for a detailed discussion of the funcion of this parameter.

`loadMeasureDegree`

Layer:	Serial and parallel
Datatype:	`int`
Default value:	1
Constraints	Must be 0, 1, 2, or 3

Used to measure the "weight" of a subproblem used to calculate worker and cluster workloads. Specifically, if a subproblem has bound b and the current incumbent value is z, its weight is $|z - b|^{\texttt{loadMeasureDegree}}$. Note that subproblem weight is used primarily by the load balancing algorithms in the parallel layer, but `LoadMeasureDegree` also exists in the serial layer for technical reasons. Its value should not materially affect the operation of the serial layer. In the parallel layer, larger values put more load balancing stress on the quality of subproblems, and lower values more stress on quantity of subproblems; a value of zero sets a processor's workload equal to the number of subproblems it controls. The parameter `qualityBalance` specifies additional attention to subproblem quality in load balancing, beyond that specified in `loadMeasureDegree`.

`numClusters`

Layer:	Parallel only
Datatype:	`int`
Default value:	1
Constraints:	Lower bound: 1

Forces a minimum number of processor clusters, even if all are smaller than `clusterSize`.

`qualityBalance`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`true`

If `true`, hubs shift work between workers based on each worker's best subproblem bound, as well as the total workload as measured by (1) in Section 3.3.4. Note that this workload metric may already take some measure of quality into account if `loadMeasureDegree` is positive.

`minScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.05
Constraints:	Lower bound: 0 , Upper bound: 1

`targetScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.25
Constraints:	Lower bound: 0, Upper bound: 1

`maxScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.90
Constraints:	Lower bound: 0, Upper bound: 1

These parameters control how frequently subproblems are released from workers. Generally, you should observe the ordering

$$\texttt{minScatterProb} \leq \texttt{targetScatterProb} \leq \texttt{maxScatterProb}.$$

The release decision scheme uses the notion of a worker having its "fair share" of work, as described in Section 3.3.3. If a worker has exactly its fair share of work, then it releases subproblems with probability `targetScatterProb`. If it has more than its fair share, it uses a higher probability, linearly increasing up to `maxScatterProb` if it has 100% of the work in the system. Similarly, if it has less than its fair share, it uses a lower probability, linearly decreasing down to `minScatterProb` if appears to have no work.

`minNonLocalScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.0
Constraints:	Lower bound: 0, Upper bound: 1

`targetNonLocalScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.33
Constraints:	Lower bound: 0, Upper bound: 1

`maxNonLocalScatterProb`

Layer:	Parallel only
Datatype:	`double`
Default value:	default: 0.9
Constraints:	Lower bound: 0, Upper bound: 1

When there is more than one cluster, these parameters control the decision, once a worker has decided to release a subproblem, of whether it should be released to the worker controlling hub, or to a randomly chosen hub (which could also be the worker's own hub). This decision is based on whether the worker's cluster has its "fair share" of the total workload: a fraction $w(c)/W$ of the total work, where $w(c)$ is the number of workers in the worker's cluster, and W is the total number of workers. When the worker's cluster has its fair share, random scattering is performed with probability `targetNonLocalScatterProb`. If the cluster has more than its fair share, a larger probability is used, increasing linearly to `maxNonLocalScatterProb` if the cluster has all the work in the system. If the cluster has less work, a smaller probability is used, decreasing linearly to `minNonLocalScatterProb` if the cluster has no work.

5.8 Parallel thread control

`incThreadBiasFactor`

Layer:	Parallel only
Datatype:	`double`
Default value:	100.0
Constraints:	Nonnegative

`incThreadBiasPower`

Layer:	Parallel only
Datatype:	`double`
Default value:	1.0
Constraints:	Nonnegative

`incThreadMaxBias`

Layer:	Parallel only
Datatype:	`double`
Default value:	20.0
Constraints:	Nonnegative

`incThreadMinBias`

Layer:	Parallel only
Datatype:	`double`
Default value:	1.0
Constraints:	Nonnegative

These parameters are used in computing the bias (priority) of the incumbent heuristic thread, if it is present. If we represent the above parameters by ϕ, π, $\bar{b}$, and $\underline{b}$, respectively, the formula for the incumbent thread bias b is

$$b = \max\left\{\underline{b}, \min\left\{\bar{b}, \phi r^{\pi}\right\}\right\},$$

where r is the current relative gap, that is, the relative difference between the incumbent value and the best known bound in the pool of active subproblems.

`timeSlice`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.01
Constraints:	Lower bound: 10^{-7}

Target thread timeslice in seconds. This is the typical run time or "granularity" that the scheduler tries to acheive for each invocation of a compute thread.

`useIncumbentThread`

Layer:	Parallel only
Datatype:	`bool`
Default value:	`true`

Controls whether each worker dedicates a thread to incumbent search. If `false`, the only source of new incumbents will be terminal subproblems and `quickIncumbentHeuristic()`. This parameter is ignored if `haveParallelIncumbentHeuristic()` returns `false`.

`workerThreadBias`

Layer:	Parallel only
Datatype:	`double`
Default value:	100.0
Constraints:	Nonnegative

Scheduling priority for main worker thread.

5.9 Ramp-up (standard implementation)

The following parameters control the default implementation of the ramp-up crossover mechanism, that is, the transition from ramp-up to "regular" parallel execution. Your application may override `parallelBranching`'s default implementations of `continueRampUp()` or `forceContinueRampUp()` to ignore these parameters or interpret them differently.

`minRampUpSubprobsCreated`

Layer:	Parallel only
Datatype:	`int`
Default value:	0
Constraints:	Nonnegative

Force this many subproblem creations before ramp up ends.

`rampUpPoolLimit`

Layer:	Parallel only
Datatype:	`int`
Default value:	0
Constraints:	Nonnegative

Total subproblem pool size beyond which the ramp-up phase may end.

`rampUpPoolLimitFac`

Layer:	Parallel only
Datatype:	`double`
Default value:	1
Constraints:	Nonnegative

Desired average number of subproblems per worker processor immediately after ramp-up.

5.10 Search order and protocol

`breadthFirst`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

In serial, use breadth-first search; in parallel, use an approximation of breadth-first search based on treating all subproblem pools as FIFO queues. Ignored if `depthFirst` is also specified.

`depthFirst`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

In serial, use depth-first search; in parallel, use an approximation based on treating all subproblem pools as stacks. Overrides `breadthFirst` if both are specified.

Note that if neither `breadthFirst` nor `depthFirst` is specified, then PEBBL uses best-first search. That is, it tries to select the subproblem with the lowest possible bound for minimization problems, and the highest possible bound for maximization problems. In parallel, conformance to the best-first search order is only approximate.

`initialDive`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

`integralityDive`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`true`

These options are useful for applications that do not have a good incumbent heuristic. Setting `initialDive` is incompatible with the `breadthFirst` and `depthFirst` options, and `integralityDive` is only meaningful if `initialDive` is `true`. `InitialDive` specifies that the best-first search order should only be followed after the first incumbent is found. Beforehand, PEBBL should "dive" in the tree to try to identify a feasible solution. If `integralityDive` is `true`, then "diving" means giving priority to processing subproblems with the *lowest* value of the data member `branchSub::integralityMeasure` (a value of zero is interpreted as meaning a subproblem is "integer feasible"). If `integralityDive` is `false`, it means selecting the subproblem with the highest possible depth. Both these techniques tend to produce initial search trees similar to classical depth-first search. Once an incumbent is found, PEBBL reverts to best-first search.

`eagerBounding`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Specifies the search protocol implemented by the "eager" handler, as described in Section 3.1.3. This handler tries to bound subproblems as soon as they are created and keep all subproblems in the pool in either the `bounded` or possibly `beingBounded` or `beingSeparated` states. This handler is recommended for applications in which subproblem bounds are typically computed very quickly. This parameter is ignored if `lazyBounding` is also specified.

`lazyBounding`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Specifies the search protocol implemented by the "lazy" handler, as described in Section 3.1.3. This handler attempts to delay bounding subproblems as long as possible, and fill all subproblem pools with `boundable` and possibly `beingBounded` or `beingSeparated` problems. Once a problem is `separated`, its children are created as soon as possible. This option takes precedence over `eagerBounding`.

If both `eagerBounding` and `lazyBounding` are `false`, which is the default, PEBBL uses the "hybrid" handler described in Section 3.1.3. This handler simply chooses problems from the subproblem pool, attempts to advance them one state in Figure 3, and replaces them. With this handler, the subproblem pools may contain a mix of all the possible states except `dead`.

5.11 Termination

`absTolerance`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	0
Constraints	Nonnegative

When enumeration is not active, the absolute tolerance for optimal objective value. Subproblems are fathomed when their bounds are within `absTolerance` of the incumbent value. The final solution reported should be within this distance of the true optimum. This parameter is used to set the `branching` class member `absTol`. However, applications may adjust the value of `absTol`. For example, in a linear pure integer program with all objective function coefficients integer, you may want to set `absTol` to at least 1. In this situation, you could alternatively design your `branching`-derived class to round up the value of `branchSub::bound` to the next integer.

`relTolerance`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	10^{-7}
Constraints	Nonnegative

When enumeration is not in use, the relative tolerance for optimal objective value. If the incumbent value is z and the subproblem bound is b, a subproblem can be fathomed if $|(z-b)/z| \leq$ `relTolerance`. This parameter essentially controls the number of digits of precision of the final solution. A value of zero is possible, but not recommended unless your bound calculations use only integer arithmetic, and there is no possibility of round-off error. This parameter is used to set the data member `relTol` in the `branching` class.

`integerTolerance`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	10^{-5}
Constraints	Lower bound: 0, Upper bound: 1

Tolerance for determining whether numbers are integers. Note that this parameter does not affect PEBBL itself, but only applications that use the methods `isInteger(double)` or `isZero(double)` provided for convenience in class `pebblBase`, from which `branching` and `branchSub` are both derived.

`maxCPUMinutes`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	0
Constraints	Nonnegative

`maxSPBounds`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	0
Constraints	Nonnegative

`maxWallMinutes`

Layer:	Serial and parallel
Datatype:	`double`
Default value:	0
Constraints	Nonnegative

These parameters control PEBBL's built-in abort function. A PEBBL run will abort if the CPU time (per processor) exceeds `maxCPUMinutes`, the total number of subproblems bounded exceeds `maxSPBounds`, or the total wall clock time spent in the search exceeds `maxWallMinutes`. In each case, a zero value, which is the default, means there is no limit.

`printAbortMessage`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`true`

Instructs PEBBL to print an explanatory message when aborting due to the `maxCPUMinutes`, `maxSPBounds`, or `maxWallMinutes` limits.

`rampUpOnly`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

Forces PEBBL runs to terminate immediately after ramp-up. This parameter is provided primarily for debugging or evaluating the performance of your application's ramp-up phase.

`useAbort`

Layer:	Serial and parallel
Datatype:	`bool`
Default value:	`false`

If `true`, force an abort when an error occurs.

6 Python-based Utilities

Like many ACRO packages, PEBBL provides some untility programs written in Python. These utilities are strictly serial, and reside in the directories

```
        acro-pebbl/packages/pebbl/python/scripts
    and acro-pebbl/python/bin
```

6.1 The `dumpSplit` utility

The `dumpSplit` utility is helpful for separating logs produced by setting the `debug` command-line parameter. For parallel runs, PEBBL prefixes each line of output with the MPI process number enclosed in square brackets. For example, output lines from processor 0 start with `[0]`, output lines from processor 1 start with `[1]`, and so forth. These lines typically appear intermixed in the output. The `dumpSplit` utility is meant to assist in forming a clearer picture of what is happening on each individual processor. Its usage is through a command line of the form

```
path/dumpSplit logFile
```

Here, ***path*** is the location of the utility (such as `acro-pebbl/python/bin` above) and ***logFile*** is the pathname of the file containing the PEBBL debug output. For each processor p that `dumpSplit` detects in ***logFile***, it writes a file of the form ***logFile***.p containing just the output flagged as coming from processor p. For example if `dumpSplit` were to be run on a file called `dump.log` containing debug log output from three processors, its output would be the files `dump.log.0`, `dump.log.1`, and `dump.log.2`.

6.2 The `pebblLoadGraph` utility

The `pebblLoadGraph` allows you to visualize parallel performance-tracking data produced by invoking the `loadLogSeconds` parameter described in Section 5.2. To use this utility, your Python installation must contain the `matplotlib` package available at `http://matplotlib.org`. When PEBBL is run with `loadLogSeconds` set to a positive value, performance data are placed in a file called ***problemName***`.loadLog` in the current directory, where ***problemName*** is the name of the current problem instance, typically derived from the problem instance filename. The `pebblLoadGraph` utility processes such files to produce either screen or PDF-file graphics. You invoke this utility by a command of the form

```
path/pebblLoadGraph [options] file
```

where ***file*** denotes the load-log file produced by the PEBBL runs. By default, graphical output is placed in a file whose name is derived from ***file*** but ends in `.pdf`. The options available in the optional clauses ***options*** are extensive and may be viewed in their entirety by the command

```
path/pebblLoadGraph --help
```

The most common options are

`--noworkers` Do not graph subproblem loads at worker processors. The opposite of this option, `--workers`, is the default.

`--nohubs` Do not graph subproblem loads at hub processors. The opposite of this option, `--hubs`, is the default.

`--proc=`p Only graph loads for processor p. The argument p may also take the value `hubs`, `workers` or `all`, which is the default. This parameter may be specified multiple times: for example, `--proc=0 --proc=2` will show only information for processors 0 and 2.

`--bounds=`p Show the known solution bounds (lower bound for a minimization problem, upper bound for a maximization problem) known at processor p. The p argument is interpreted the same way as for the `--proc` option.

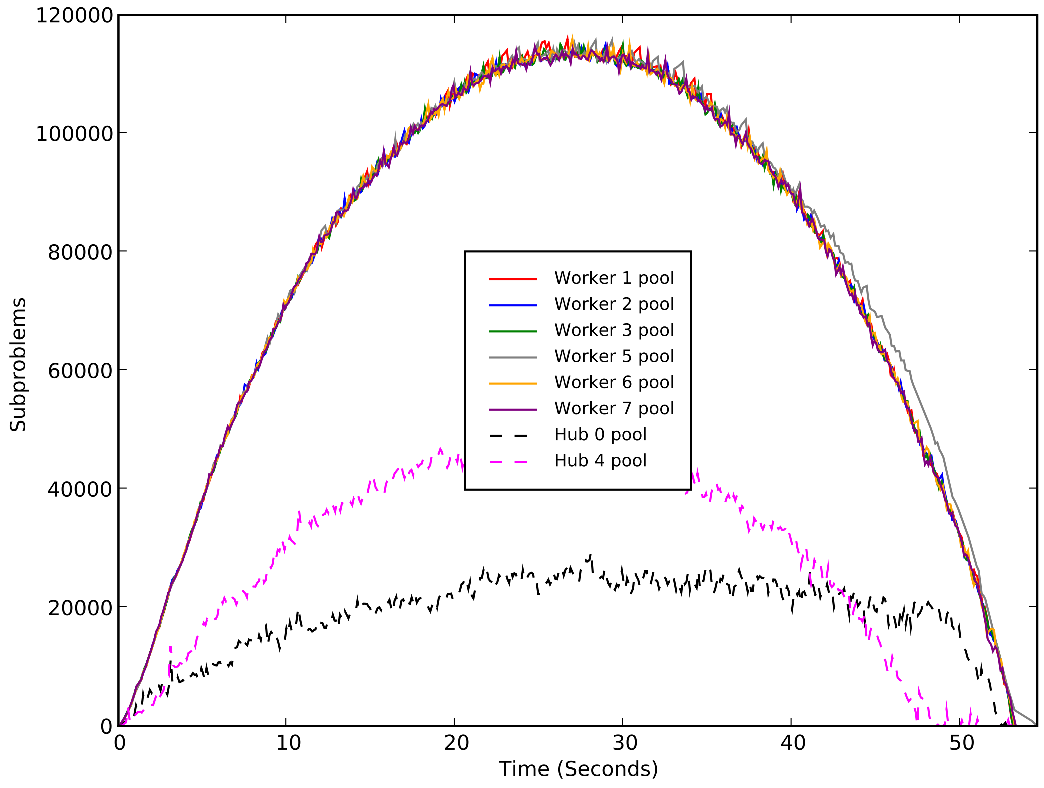

Figure 12: Sample graphical output from `pebblLoadGraph`

.

`--incumbent=`p Show the incumbent values known at processor p. The p argument is interpreted the same way as for the `--proc` option.

`--show` Show the graphical output on-screen in addition to producing the default PDF file. Requires an active X display connection.

`--startseconds=`s_1 Only show information after s_1 seconds of PEBBL runtime.

`--endseconds=`s_2 Only show information before s_2 seconds of PEBBL runtime elapsed.

Note that separate loads may be displayed for a single processor if it is both a worker and a hub — one display indicates subproblems under control of the worker process and the other indicates subproblems under control of the hub process.

Figure 12 shows the result of running `pebblLoadGraph` with its default settings on a logfile produced by running PEBBL on a difficult knapsack problem using 8 MPI processors, two of which were desiganed as combined worker-hubs, and the rest as workers.

References

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

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

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

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

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

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

[7] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: an object-oriented framework for parallel branch and bound. RUTCOR Research Report RRR 40-2000, Rutgers University, 2000.

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

[9] W. D. Hillis. *The Connection Machine.* MIT Press, 1985.

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

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

[12] B. Le Cun and C. Roucairol. BOB: a unified platform for implementing branch-and-bound like algorithms, 1995. `http://citeseer.ist.psu.edu/cun95bob.html`.

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

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

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

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

[17] Y. Shinano, K. Harada, and R. Hirabayashi. Control schemes in a generalized utility for parallel branch-and-bound algorithms. In *IPPS: 11th International Parallel Processing Symposium*. IEEE Computer Society Press, 1997. `http://citeseer.ist.psu.edu/shinano97control.html`.

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

[19] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[20] S. Tschöke and T. Polzer. Portable parallel branch-and-bound library PPBB-Lib user manual, 1996. `http://citeseer.ist.psu.edu/tsch96portable.html`.