PEBBL 1.0 User Guide

Jonathan Eckstein* Cynthia A. Phillips† William E. Hart†

February 24, 2009

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 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 tune PEBBL's behavior.

*Business School and RUTCOR, Rutgers University, 640 Bartholomew Road, Piscataway, NJ 08854-8003
†Sandia National Laboratories, Mail Stop 1110, P.O. Box 5800, Albuquerque, NM 87185-1110

Contents

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 algorithm. It is thus similar to other software projects such as PUBB [16, 15], BoB [10], PPBB-Lib [18], and ALPS [13] (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 [17].

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 is a relatively new name for what was conceived as 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 [4, 6, 5]. In fact, significant portions of this user guide are derived from Eckstein et al. [5, 6].

PEBBL's parallelization strategies are patterned after CMMIP [2, 3], 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 shell 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 [8] 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 auxillary libraries, like UTILIB.

2.1 Downloading PEBBL

The Acro software library can be installed from a distribution file (tape, etc.) or using a checkout from the Concurrent Version System (CVS) repository. If you are accessing current files from the CVS repository, you need to use the **cvs.a** and **ssh.cvs** scripts, which can be downloaded from http://software.sandia.gov/Acro. The Acro library can be checked out of the cvs repository by executing:

```
cvs.a checkout acro-pebbl
```

The use of **cvs.a** requires an account on the machine software.sandia.gov, which is generally only available to Sandians and academic collaborators of Sandians.

The latest version of PEBBL can also be acquired in compressed tarball form from

```
http://software/sandia.gov/Acro
```

Once downloaded, compressed tarballs for the version of the day (VOTD), for example, can be downloaded and extracted with the following:

```
gzip -d pebbl-VOTD.tar.gz | tar xf -
```

2.2 Configuring and Building

PEBBL can be configured using the standard build syntax:

```
cd acro
./configure
make
```

This builds the library libpebbl.a in acro/lib, along with supporting libraries. The library headers are installed in acro/include. PEBBL developers using CVS need to build with the autoconf tools. This can be done with the following syntax:

```
cd acro
autoreconf
./configure
make
```

The acro/setup command can also be used to simplify this process:

```
cd acro
./setup configure make
```

The **setup** command generates the following files in the acro/test directory:

```
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
```

The Acro build can be modified through the use of environment variables and with command line options to the **configure** script. If you set no environment variables and use no command line options, the **configure** script will choose sensible values.

```
configure --help
```

This will list all the configuration options, and in some instances the default action. Typical use might be to define where MPI is located, and in which directory to install the Acro libraries and header files:

```
configure --with-mpi-compilers=/usr/local/mpich-1.2.4/ch_p4/bin
       --prefix=/Net/usr/local
```

A detailed discussion of these configuration options is given in the file

```
acro/INSTALL
```

and on the Acro web pages (see http://software/sandia.gov/Acro).

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

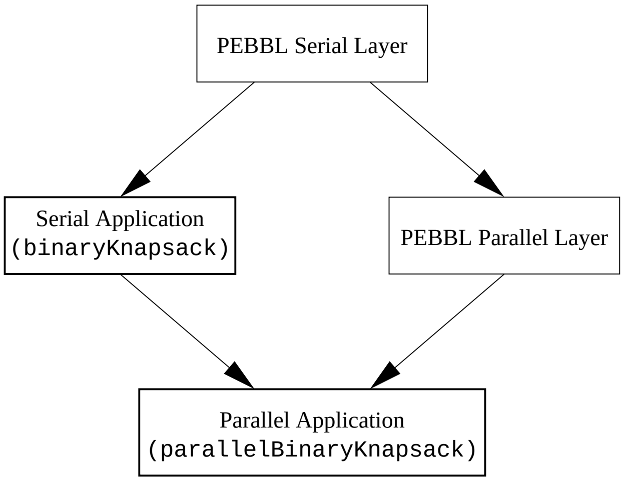

Figure 1: The conceptual relationships of PEBBL's serial layer, the parallel layer, a serial application (in this case, `binaryKnapsack`), and the corresponding parallel application (in this case, `parallelBinaryKnapsack`).

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`. 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 Special Search Features

3.1.1 Tolerances and enumeration capabilities

In its normal 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 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 the 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 acquires a new incumbent solution (that is, a solution better than all previously found ones) that exists for at least m minutes, it immediately writes it to disk so that it is available if the process crashes. Early output is also useful if you wish to voluntarily terminate a run and still have access to the best feasible solution. This is useful in early testing of an application where you may not yet know good tolerance values. For example, if you initiate a run with a relTolerance of 5%, but after considerable computation, the 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 work 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 each processor have access to disk I/O, but not necessarily in a shared directory.

The computation can be restarted from the time of the checkpoint by specifying either of the command-line parameters `restart` or `reconfigure`. Using `restart` is 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.

3.3 Serial layer architecture

3.3.1 The `branching` and `branchSub` classes

To define a serial branch-and-bound algorithm, you must extend the two key classes in the PEBBL serial layer, `branching` and `branchSub`. The `branching` class stores global information about a problem instance, and contains methods that implement various kinds of serial branch-and-bound algorithms, as described below. Each `branchSub` object stores data about a subproblem in the branch-and-bound tree, and the `branchSub` class contains methods that perform generic operations on subproblems. This basic organization is borrowed from ABACUS [8], but is more general, since there is no assumption that cutting planes or linear programming are involved.

For example, 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 `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 overall branch-and-bound process. Finally, 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

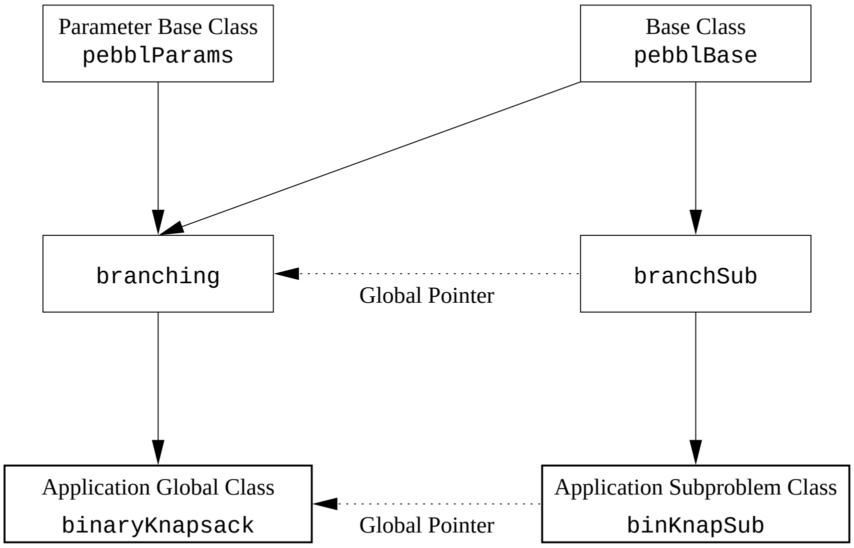

Figure 2: Basic class hierarcy for a serial PEBBL application (in this case, `binaryKnapsack`, with corresponding subproblem class `binKnapSub`).

the UTILIB class parameter package. Figure 2 illustrates the basic class hierarchy for a serial PEBBL application.

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

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

and `binKnapSub` is defined via

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

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*`.

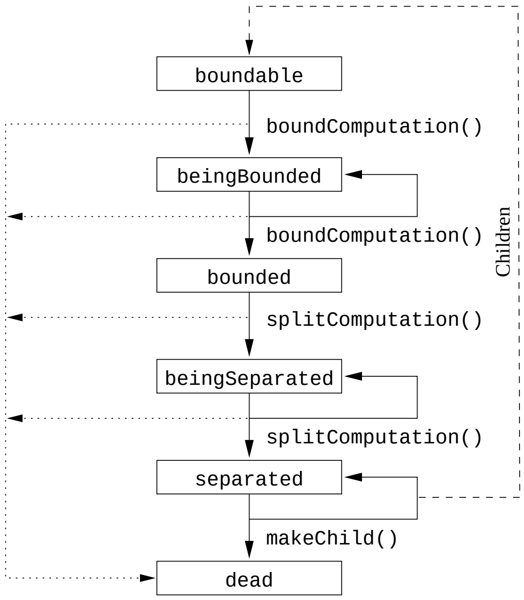

Figure 3: PEBBL's subproblem state transition diagram. It is possible that a single application of `boundComputation` may take a subproblem from the `boundable` state, through `beingBounded`, to `bounded`. Similarly, a single use of `splitComputation` may move a subproblem from `bounded`, through `beingSeparated`,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

3.3.2 Manipulating subproblem states

A key feature of PEBBL, first published in Eckstein et al. [6], 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, it is set aside, and another task or subproblem is consider. 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. 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 4.1.

3.3.3 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 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 separate issue from the search order; the "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 [1]), 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 [1] for an example of this terminology). Here, all subproblems in the pool have already been bounded. One picks a subproblem out of 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 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.

[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 Subsection 4.1. 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.

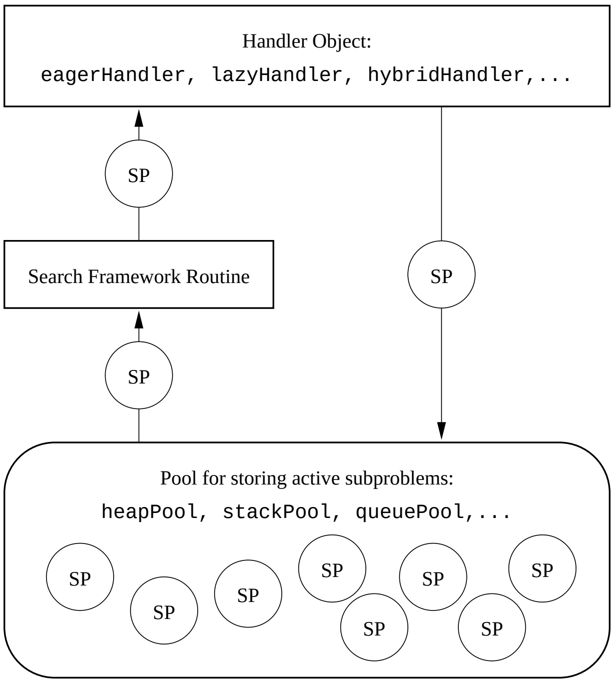

Figure 4: The search framework, pool, and handler. Each "SP" indicates a branch-and-bound subproblem — an object of type derived from `branchSub`.

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. 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.

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.

3.4 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 by using MPI to emulate a message passing system.

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 3.6 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.

3.4.1 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 header file `pebbl/src/example/parKnapsack.h` defines this inheritance structure. It defines `parallelBinaryKnapsack` via

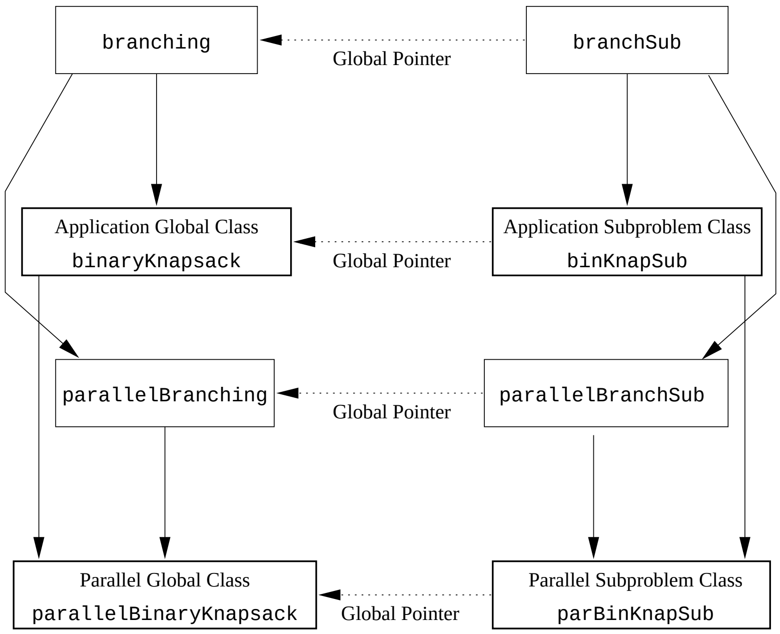

Figure 5: Inheritance structure of the parallel knapsack application. Other parallel applications are similar.

```
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 4.2. The most critical of these methods are `pack` and `unpack`, which must be defined both for the global parallel class and for subproblems. These methods respectively describe how to encode and decode objects into a UTILIB `Packbuffer` or `UnpackBuffer` object [7]. For the global parallel class, PEBBL uses `pack` and `unpack` when it broadcasts the problem description to all processors at the outset of the parallel search. For subproblems, PEBBL uses `pack` and `unpack` in the transmission of subproblems between processors.

3.4.2 Processor clustering

PEBBL's parallel layer employs a generalized form of the processor organization used by the later versions of CMMIP [2, 3]. 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 `parallelPebblBase`, govern the partitioning of processors into clusters: `clusterSize`, `numClusters`, and `hubsDontWorkSize`. First PEBBL finds the size k of a "typical" cluster via the formula

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

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 $\overline{p} \bmod k$, which will be between 1 and $k-1$. 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 slow the hub's response to messages, slowing down the entire cluster. In such cases, a "pure" hub is more advantageous.

3.4.3 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 Subsection 3.3.3, although all workers use the same pool type. Depending on various parameter settings, however, the pool might be very small, in the extreme case never holding more than one subproblem. Each worker processes its pool in the same general manner as the serial layer: it picks subproblems out of the pool and passes them to a search handler until the pool is empty. 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. 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 subproblems is always returned to the hub at a certain point in their lifetimes (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, where c denotes the cluster and $w(c)$ the total number of workers in the cluster, 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* [14, 2], 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 Subsection 3.3.3. 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 make sure 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 parameter `workerSPThreshHub`; if a worker appears to have fewer than this many subproblems in its local pool, the hub judges it "deserving" of more subproblems. 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 [2, 3, 14]. 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'. 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. If a worker detects that it has a number of subproblems exceeding a user-specifiable factor `workerMaxLoadFactor` times the average 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.

3.5 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 [2, 3], 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 [3]; [11] and [9] 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

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

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. Periodically, messages "sweep" through this entire tree, from the leaves to the root, and then back down to the leaves. For the details of the rendezvous scheme, see [6, Section 4.4] or [5, Section 4.3]. 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. Note that unlike ALPS [13], PEBBL does not employ a "master of masters" or "hub of hubs" processor, and its parallelization scheme is in principle indefinitely scalable.

The load balancing scheme has an important secondary function of detecting termination, which can in general be quite tricky in highly asynchronous parallel programs. The general approach is a varient of the "four counters" technique proposed in [12], although only three counters are actually necessary. More details may be found in [6, Section 4.6] or [5, Section 4.5].

3.6 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 significant negative impact on algorithm 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 is elongated and it may be hard to make efficient use of all available processors.

To help you improve ramp-up performance, PEBBL implements 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 develop the same search nodes around the root of the branch-and-bound tree. During this phase, 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 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 when returning.

Ramp-up execution is controlled by two virtual methods, `continueRampup()` and `forceContinueRampUp()`. 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{`rampUpPoolLimit`, $\overline{p}$ · `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 implemetations more specific to your application.

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 MPI implementations and MPP 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 schedule 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 coordi-

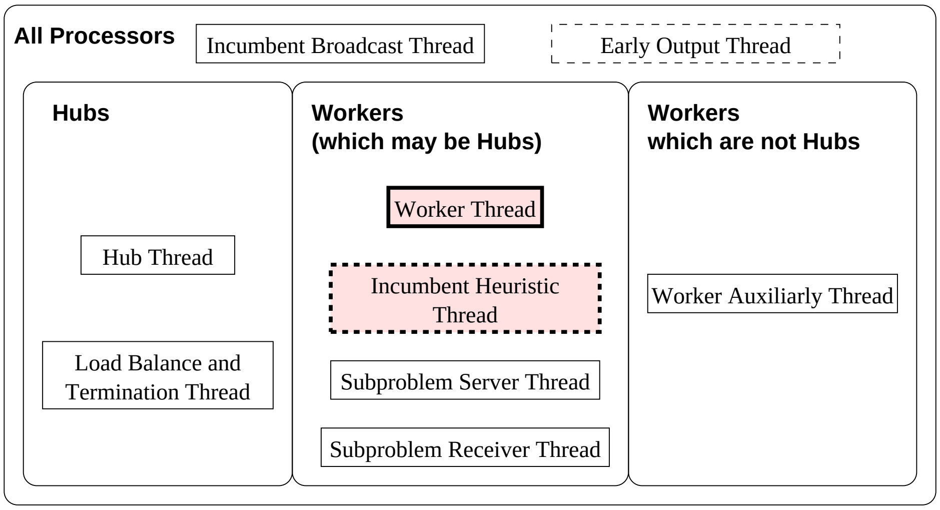

Figure 6: PEBBL's standard threads. Compute threads are shaded; all other threads are message-triggered. A dashed outline indicates that the thread may not exist in some cases.

nates 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).

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 balance and 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 hub sends this thread various instructions, such as to write a checkpoint, check for termination, or terminate the search process. It also periodially receives information relevant to the load balancing algorithms.

3.8 Enumeration capabilities

In response to the requirements of some applications at Sandia National Laboratories, PEBBL also has the ability to enumerate alternative optima and near-optimal solutions to problems. These capabilities are currently only supported in PEBBL's serial layer. These capabilities are controlled by the parameters `enumAbsTolerance`, `enumRelTolerance`, and `enumCount`. If exactly one of the two enumeration tolerance is set, PEBBL attempts to enumerate all solutions within the given additive or relative distance from optimality. If `enumCount` is also set, at most `enumCount` solutions will be returned. If `enumCount` is set, but neither of the enumeration tolerances is, PEBBL attempts to return the best `enumCount` solutions. When both `enumAbsTolerance` and `enumRelTolerance` are set, then fathoming is not considered if a subproblem is within either the absolute or the relative enumeration tolerance.

The enumeration parameters `enumAbsTolerance` and `enumRelTolerance` complement the bound parameters `relTolerance` and `absTolerance`. The bound parameters control how fathoming is performed, comparing the value of an incumbent with a subproblem's lower bound. The enumeration parameters determine whether fathoming considered. These tolerances compare the incumbent with the lower bound in a similar fashion, but the goal is to determine whether this subproblem could possibly contain a near-optimal solution.

The enumeration mechanism maintains a cache of incumbent solutions. This cache uses a splay tree to enable effective insertion as well as search for the incumbents that are within tolerance of the best observed incumbent. As new points are added to the cache, other points are removed if they are not within tolerance or if the number of solutions in the cache exceeds `enumCount`. This generic mechanism does not need to be adapted for new applications, but it can be customized (for example to control the final output).

Note: the early output mechanism and enumeration mechanisms currently are independent. When using enumeration, the early output mechanism simply outputs the best incumbent.

4 Creating PEBBL Applications

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

4.1 Defining a serial application

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

- Define a class extending `branching`
- Define a class extending `branchSub`
- Create a "driver" program that runs your algorithm.

You may do all these things 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 C++ source containing the driver.

To load the basic definitions of `branching` and `branchSub` 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 should also refer to the files `serialCore.h`, and `serialKnapsack.{h,cpp}` in directory `pebbl/src/example/`.

4.1.1 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

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

using namespace pebbl;

class myBranchSub; // Forward declaration

class myBranching : virtual public branching
{
    friend class myBranchSub;

public:
  ⋮
}

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(); }
  ⋮
}
```

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

not advisable. 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); ··· };
```

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:

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

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

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

Here, *myBranchSub*`::setGlobalInfo` should be a routine that copies any necessary information from the *myBranching* object into a *myBranchSub* object. At a minimum, it should set `globalPtr`.

`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 in `argv[1]`.

`serialPrintSolution(const char* header,const char* footer,std::ostream& s)`

Typically, the `branching`-derived class should have the ability to store a single solution to the optimization problem. The `serialPrintSolution` method should print this solution to the stream `s`, preceded by the `header`, and followed by `footer`. Strictly speaking, you do not have to implement this method — it has a default implementation of { }. However, it should be defined if you need PEBBL to print out anything besides the *value* of the solution.

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

Constructor

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

```
myBranchSub () { ··· };
```

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 `branchComputation()`, 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 description 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 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 and it is not necessary to separate the subproblem — that is, a subproblem is a *terminal* node of the branch-and-bound tree. 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 tells PEBBL that you have computed a feasible solution to the overall optimization problem, as opposed to just a bound. PEBBL will most likely invoke `updateIncumbent()` (see below) to retrieve this solution.

`int splitComputation()`

This method should attempt to separate the subproblem. Similarly to `boundComputation()`, it will be called 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.

`branchSub* makeChild(int whichChild)`

This method should create a child problem and return 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.

`void updateIncumbent()`

This method should set the state of the `branching`-derived class — *e.g.* *`myBranching`* —

to reflect a feasible problem solution correspondig to this subproblem. PEBBL will typically call this method after `candidateSolution()` returns `true`. A typical implementation will look like:

```
void myBranchSub::updateIncumbent()
{
   bGlobal()->incumbentValue = bound; // Set the incumbent value
   (Copy the solution itself to *global())
   bGlobal()->signalIncumbent();
}
```

Once the solution has been copied to a `branching`-derived class, it can be written by the `serialPrintSolution(···)` method. The call to `signalIncumbent()` tells PEBBL that the subproblem pool should be pruned. Once the application is ported to the parallel layer, `signalIncumbent()` will also initiate interprocessor communication to broadcast the new incumbent value.

4.1.3 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 generality. Note that once you migrate your application to the parallel layer, `preprocess()` is called *after* the problem instance is broadcast to all processors, and could potentially be parallelized.

`double branching::initialGuess()`

This method is provided to allow you to compute 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 anything: for example, the default implementation is an empty stub. If you do find something, you should set `incumbentValue` and the application-specific data structures (within *myBranching*, for example), describing the present solution.

`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()`

PEBBL will only call this method if `haveIncumbentHeuristic()` returns `true`, and only for `bounded` subproblems for which `candidateSolution()` returns `false`. If called, this method should attempt to create a feasible solution to this problem. If this problem has a better objective value than `bGlobal()->incumbentValue`, you should call `updateIncumbent()`, or perform an equivalent sequence of operations.

`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, it 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.

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. Parameter creation statements usually 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 data member in which to store the value of the parameter.

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

defaultValue is the default value for the parameter. Data member *internalName* will take this value if command line parameter --*commandLineName* is not specified.

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 specifies simple constraints on the value of the parameter. For details, refer to the UTILIB Parameter class documentation [7].

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`

You may specify parameter error checks more elaborate than those available in ***error check object*** by overloading the method `bool branching::checkParameters`; if you detect an error in this routine, print a diagnostic message and return `false`. If you do not, execute `return branching::checkParameters()` to invoke the standard checks on PEBBL's built-in parameters.

4.1.4 Constructing a serial driver

Once you have defined `branching`- and `branchSub`-derived clases as above, you need to create a "driver" program that invokes them. A typical form for a driver program would be:

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

Note that the main PEBBL header file `<pebbl/branching.h>` automatically includes the necessary UTILIB header files to define the method `InitializeTiming()`.

Note 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. 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 `serialPrintSolution(···)` implementation to write the final solution to a file whose name is derived from the first command line argument not recognizable as a parameter setting (if there is none, the file is called `solution.dat`).

Supposing your driver were called *myDriver*, both of the command lines

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

would call setup with `argv[1]` containing the null-terminated string *datafile*. In the second case, `setup` would automatically recognize the PEBBL parameters `relTolerance` and `earlyOutputMinutes`, and their settings would be reflected when executing `solve()`. For a catalog of recognized parameters, refer to Section 5.

The `setup(···)` method 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)`.

`--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 *file* — see the UTILIB Parameter class documentation for a description of the file format.

Note that PEBBL applications can be invoked in ways other than the simple driver above; for example, they could be embedded in more complicated programs. However, the techniques for doing so are not covered in this version of the user's guide.

4.2 Defining a parallel application

Suppose now that your serial layer applications runs acceptably, and you wish to parallelize it. Figure 8 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 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. You must configure with MPI options in order to develop a parallel application. Otherwise, none of the code for parallel PEBBL applications will be compiled.

Note that PEBBL uses `parallelBranchSub::pGlobal()` to find the `parBranching` object associated with a subproblem object. The method `global()` in Figure 8 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.

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

using namespace pebbl;

class myParSub; // Forward declaration

class myParBranching :
   virtual public parallelBranching,
   virtual public myBranching
{
public:
   ⋮
};

class myParBranchSub :
   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(); }
   ⋮
};
```

Figure 8: Standard code pattern for creating a parallel PEBBL application.

4.2.1 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() { ··· };
```

`parallelBranchSub* blankParallelSub()`

This method is `blankSub`, but should return a parallel subproblem with correctly initialized global pointers, as in:

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

See below for a possible implementation of *myParSub*`::setGlobalInfo`.

`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 `IntVector` or `DoubleVector`, may also be written with `<<`. See the UTILIB documentation for the details of `PackBuffer`s.

`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 datatypes, 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` may be read via `inBuffer >> time >> money;` in `unpack`. The `<<` can also read entire UTILIB arrays.

`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 UTILIB arrays, the space required is a single sizeof(size_t) to hold the size of the array, plus storage for each of the data elements. 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 in all processors.

`void packSolution(PackBuffer& outBuffer)`

PEBBL uses this method to send problem solutions between processors. It should pack (typically using the << operator) a full description of the current problem solution into outBuffer.

`void unpackSolution(UnPackBuffer& inBuffer)`

This method should read the information written by packSolution, typically using the >> operator.

`int solutionBufferSize()`

Should return an upper bound on the size (in bytes) of the buffer needed by packSolution and unpackSolution.

Note that it is possible to implement packSolution and unpackSolution via stubs and have solutionBufferSize() simply return 0. To do so, however, you should set the parameter printSolutionSynch to false. This setting tells PEBBL's parallel layer that any processor is permitted to do I/O, and it is thus unnecessary to move subproblem solutions between processors.

4.2.2 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 8, 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 description of the subproblem into `outBuffer`, typically using the `<<` operator.

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

This method should unpack the description of the subproblem from `inBuffer`, typically using the `>>` operator.

```
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.

4.2.3 Standard disambiguations

While the PEBBL "diamond" inheritance pattern 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. When 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. 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.

4.2.4 Selected additional methods — incumbent heuristic

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 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.

In your implementations of these methods, keep in mind that whenever you change the incumbent, you should do the following things, as outlined in the sample implementation of `branching::updateIncumbent()` above.

- Update `branching::bound` to reflect the value of the incumbent.
- Copy a description of the solution into application-specific data structures in your `branching`-derived class (such as *myBranching*) so that it can be output via your `serialPrintSolution` method.
- Call `signalIncumbent()`.

In the parallel layer, the call to `signalIncumbent()` is crucial to making sure that all processors become aware of the new incumbent value.

`void parallelBranchSub::quickIncumbentHeuristic()`

The heuristic to be run for every bounded subproblem. The default implementation is a stub, and you need not attempt to run your heuristic every time `quickIncumbentHeuristic()` is called. For example, your implementation could immediately return if the subproblem 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.

`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 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.

4.2.5 Selected additional methods — ramp-up

Section 3.6 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.

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()`, or the ramp-up phase may deadlock.

`void parallelBranchSub::rampUpIncumbentHeuristic()`

This method substitutes for the usual `quickIncumbentHeuristic()` during the ramp-up phase; note that `feedToIncumbentThread()` is still called during ramp-up, even though the incumbent heuristic thread will not be running yet. The default implementation of `rampUpIncumbentHeuristic()` is:

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

`bool parallelBranching::continueRampUp()`

Ramp up will continue as long as either this method or `forceContinueRampUp()` return `true`. The default implementation is described in Section 3.6, 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 3.6, 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.

4.2.6 Selected additional methods — 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. In this case, you need to define a few extra methods in order for checkpointing to work properly:

`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

```
int main(int argc, char* argv[])
{
    bool flag = true;

    InitializeTiming();

    uMPI::init(&argc,&argv,MPI_COMM_WORLD);
    CommonIO::begin();
    CommonIO::setIOFlush(1);

    myParBranching instance;
    flag = instance.setup(argc,argv);
    if (flag)
       {
         instance.reset();
         instance.printConfiguration();
         instance.solve();
       }

    CommonIO::end();
    uMPI::done();

    return !flag;
}
```

Figure 9: Example parallel driver program.

dataset written by `appCheckpointWrite` when the checkpoint was created. When restarting with `--reconfigure`, the number of worker and hub processors may be different from when the checkpoint was written.

4.2.7 Constructing a driver

Driver programs for parallel PEBBL applications look quite similar to those for serial PEBBL applications; the main difference is that you need to initialize the MPI and UTILIB `CommonIO` environments before loading the problem. Figure 9 shows a simple parallel driver; note that the PEBBL header file `<pebbl/parBranching.h>` automatically defines the classes `uMPI` and `CommonIO` from UTILIB, along with the `InitializeTiming()` method. The call to `printConfiguration()` is optional, and prints information on the number of worker and hub processors.

A driver program can be built using the libraries in `acro/lib` and headers in `acro/include`. For example,

```
g++ -I. -Iacro/include knapsack.cpp -c -o knapsack.o
```

can be used to build an object file, and

```
g++ -o knapsack knapsack.o -Lacro/lib -lpebbl -lutilib -lm
```

is an example of how an executable can be built. Note that a convenient way to build with MPI is with a the MPI compiler script `mpiCC`. This script can simply replace the `g++` compiler in this example.

A parallel driver should in general be invoked with the `mpirun` command (or `mpiexec` in some batch environments). For example, if your parallel driver were called *`myParDriver`*, the command

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

would run it on four processors, without an incumbent heuristic thread, and with the input file *`datafile`*.

It is also possible to construct combined serial/parallel drivers that sense whether they are being run in serial or parallel, and invoke the appropriate PEBBL class, for example *`myBranching`* or *`myParBranching`*. For example, see `pebbl/src/example/knapsack.cpp` This driver also illustrates the trapping of exceptions that may be thrown by PEBBL.

If MPI detects only one processor, such adaptive serial/parallel drivers will use the serial application class, and otherwise the parallel one. Note that in some cases, usually for debugging purposes, you may want to run your parallel class, but only on one processor. To provide a means of doing so, the template function

```
parallel_exec_test<myParBranching>(int argc, char** argv, int nproc)
```

scans the command line for the parameter `--forceParallel`; it returns `true` if either `nproc` > 1 or `--forceParallel` is present, and otherwise `false`.

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 contructed in the manner described in this guide. You may add your own application parameters as described in Section 4.1.3.

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 aids

`debug`

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

Debugging diagnostic output level.

`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.

`forceParallel`

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

Force the use of a parallel PEBBL solver, even if there is only one processor. Requires correct driver implementation to function properly.

`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.

5.3 Enumeration

`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.

`enumCount`

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

If positive, indicates the limit on the number of enumerated solutions. If `enumRelTolerance` or `enumAbsTolerance` are set, return an arbitrary set of up to `enumCount` solutions meeting the tolerance criteria. If neither enumeration tolerance is set, return a set of `enumCount` solutions with the best acheivable objective values. A value of 0 disables the feature.

`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

`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.

`printFullSolution`

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

Print full solution 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.

`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.

`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 3.5. 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

`targetWorkerKeepFrac`

Layer:	Parallel only
Datatype:	`double`
Default value:	0.70
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 measured by the total weight of the subproblems; see (1) in Section 3.5. The fair share is defined to be a fraction `targetWorkerKeepFrac`$/W$ of the total workload, where W is the total number of worker processors. If a worker has exactly its fair share, 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, $\overline{b}$, and $\underline{b}$, respectively, the formula for the incumbent thread bias b is

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

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.

$\boxed{\texttt{timeSlice}}$

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.

$\boxed{\texttt{useIncumbentThread}}$

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`.

$\boxed{\texttt{workerThreadBias}}$

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 (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.

$\boxed{\texttt{minRampUpSubprobsCreated}}$

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 stacks. 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 thereof 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.

$\boxed{\texttt{initialDive}}$

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

$\boxed{\texttt{integralityDive}}$

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.

$\boxed{\texttt{eagerBounding}}$

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

Specifies the search protocol implemented by the "eager" handler, as described in Section 3.3.3. 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 `beginSeparated` 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.

$\boxed{\texttt{lazyBounding}}$

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

Specifies the search protocol implemented by the "lazy" handler, as described in Section 3.3.3. This handler attempts to delay bounding subproblems as long as possible, and fill all subproblem pools with `boundable` and possibly `beingBounded` or `beginSeparated` 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 3.3.3. 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`.

`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` exists in the serial layer because it is defined by the class `loadObject`, which both the serial and parallel layers use to track various subproblem statistics. Its value should not 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`.

5.11 Termination

`absTolerance`

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

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

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.

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.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] W. E. Hart. UTILIB 3.0 user manual. Technical report, Sandia National Laboratories, 2006. (to appear).

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

[9] 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.

[10] 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`.

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

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

[13] 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.

[14] 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.

[15] 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`.

[16] 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.

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

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