

SANDIA REPORT

SAND2000-3000

Unlimited Release

Printed December 2000

PICO: An Object-Oriented Framework for Branch and Bound

Jonathan Eckstein, William E. Hart, and Cynthia A. Phillips

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94OR21400.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.doe.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: <http://www.ntis.gov/ordering.htm>



SAND2000-3000
Unlimited Release
Printed December 2000

PICO: An Object-Oriented Framework for Branch and Bound

Jonathan Eckstein*
MSIS Department
Faculty of management and RUTCOR
Rutgers University
640 Bartholomew Road
Piscataway, NJ 08854-8003
email jeckstei@rutcor.rutgers.edu

William E. Hart and Cynthia A. Phillips
Optimization/Uncertainty Estimation Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0111
email: wehart@sandia.gov, caphill@sandia.gov

Abstract

This report describes the design of PICO, a C++ framework for implementing general parallel branch-and-bound algorithms. The PICO framework provides a mechanism for the efficient implementation of a wide range of branch-and-bound methods on an equally wide range of parallel computing platforms. We first discuss the basic architecture of PICO, including the application class hierarchy and the package's serial and parallel layers. We next describe the design of the serial layer, and its central notion of manipulating subproblem states. Then, we discuss the design of the parallel layer, which includes flexible processor clustering and communication rates, various load balancing mechanisms, and a non-preemptive task scheduler running on each processor. We describe the application of the package to a branch-and-bound method for mixed integer programming, along with computational results on the ASCI Red massively parallel computer. Finally we describe the application of the branch-and-bound mixed-integer programming code to a resource-constrained project scheduling problem for Pantex.

Acknowledgements

We would like to thank ILOG, Inc. and DASH Optimization, Inc. for making their linear programming solvers (CPlex and XPress respectively) available to us.

We would like to thank Ed Kjeldgaard, Dean Jones, and Craig Lawton of Sandia National Laboratories Critical Infrastructure Surety Department, and Mark Turnquist and Linda Nozick of Cornell University for including us on the PPM team, explaining the Pantex problem to us, and providing us with data. We thank Vitus Leung and Robert Carr of the Optimization and Uncertainty Quantification Group for helpful discussions.

*This work was supported by the Laboratory-Directed Research and Development Program and supported in part by National Science Foundation Grant CCR-9902092.

Preface

This document summarizes the research conducted under the “Parallel Combinatorial Optimization for Scheduling Problems” LDRD, which was funded for fiscal years 1998 through 2000. A portion of this research will appear in the conference proceedings entitled *Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*.

Contents

1	Introduction	8
2	The general design of PICO	9
3	The serial layer	12
3.1	Subproblem states	12
3.2	Pools, handlers, and the search framework	14
3.3	Serial layer virtual methods and run-time parameters	17
3.4	Memory management	17
4	The parallel layer	20
4.1	Processor clustering	22
4.2	Tokens and work distribution within a cluster	23
4.2.1	Random release of subproblems	23
4.2.2	Subproblem tokens	24
4.2.3	Hub operation and hub-worker interaction	24
4.2.4	Rebalancing	25
4.3	Work distribution between clusters	25
4.4	Thread and scheduler architecture	27
4.4.1	The scheduling algorithm	28
4.4.2	Thread group organization and thread types	29
4.4.3	Beginning the parallel search	29
4.4.4	The worker thread	30
4.4.5	The incumbent heuristic thread	31
4.4.6	The hub thread	31
4.4.7	The incumbent broadcast thread	31
4.4.8	The subproblem server thread	32
4.4.9	The subproblem receiver thread	33
4.4.10	The worker auxiliary thread	33
4.4.11	The load balancer thread	33
4.5	Termination detection	34
4.5.1	The case of a single cluster	34
4.5.2	Multiple clusters	35
5	Application to mixed integer programming	36
5.1	Pseudocosts	37
5.2	Other serial aspects of the algorithm	38
5.3	Serial implementation	38
5.4	Parallel implementation	39
5.5	Preliminary computational results	43
6	Resource-constrained project scheduling	52
6.1	Problem description	52
6.2	The Pantex MIP Formulation	54

6.3	Pantex Incumbent Heuristics	56
6.4	Input/Output and Variable Mapping	57
6.5	Modeling Issues	58
7	Debugging and correctness	59
8	Conclusion and future development plans	60

1. Introduction

This report describes PICO (*Parallel Integer and Combinatorial Optimizer*), an object-oriented framework for parallel implementation of branch-and-bound algorithms. Parts of PICO are based on CMMIP [7–10], a parallel branch-and-bound code for solving mixed integer programming problems on the CM-5 parallel computer. Although CMMIP exhibited excellent scalability to large numbers of processors, its design had a number of limitations: first, it implemented only one specific branch-and-bound algorithm for a single (if fairly general) class of problems; adapting it to more specialized classes of problems or to use more advanced algorithmic techniques, such as branch and cut, proved awkward. Second, CMMIP was designed to showcase certain properties of the CM-5, whose communication network was fast relative to its processors, with specialized hardware and operating system support for particular kinds of interprocessor communication. To run efficiently on systems with less specialized communication capabilities, it had to be significantly restructured, as in [9].

By contrast, PICO is meant to be a very general parallel branch-and-bound environment. Using object-oriented techniques, the parallel search “engine” is cleanly separated from the details of the application and computing platform. This approach allows the same underlying parallel search code to be used on a wide variety of branch-and-bound applications, ranging from those not requiring linear programming bounds to branch-and-cut methods. The basic search engine also has a large number of run-time parameters that allow the user to control the quantity and pattern of interprocessor communication. On systems with relatively slow, unsophisticated communication abilities, such as networks of workstations, these parameters can be “tuned” so that the code attempts a relatively low level of interprocessor communication. For “MPP” supercomputers with efficient hardware and software communication support, the code can be adjusted to make full use of the available communication bandwidth. A key design goal is that, in such MPP environments, PICO retain and extend the level of scalability exhibited by CMMIP.

Flexible software environments, sometimes called “shells,” for constructing branch-and-bound algorithms are not a new idea. Broadly, prior research in this area divides into two main categories. On the one hand, there are a number of packages aimed at serial implementation of sophisticated linear-programming-based branch-and-bound methods, like branch and cut or branch and price. Perhaps the most popular of these environments is MINTO [29], and another noteworthy contribution is the ABACUS object-oriented branch-and-cut environment [18]. PICO bases some of its basic class hierarchy structure on ABACUS.

On the other hand, there have also been a number of tools for parallel implementation of general branch-and-bound algorithms, such as PUBB [35,36], BoB [4], and PPBB-Lib [39]. These efforts stem primarily from the computer science community, and emphasize parallel implementation, but appear to be designed primarily for applications with simple bounding procedures not based on linear programming.

More recently, there have been efforts at parallel implementation of advanced linear-programming-based branching methods. Some recent contributions and works in progress include PARINO [24], SYMPHONY [32], and BCP [1]. SYMPHONY and BCP, which are similar to one another, are broadly extensible libraries, but their design does not emphasize scalability to large numbers of processors.

The primary goal of PICO is to eventually combine capabilities similar to all of these tools with the scalability and flexibility of the work-distribution scheme of CMMIP, with additional adjustments to accommodate a large variety of hardware platforms. PICO allows a wide range of branch-and-bound methods, linear-programming-based and otherwise, to use the same basic parallel search engine. This sharing occurs at the link level, without requiring recompilation. While branch-and-cut capabilities are not yet present, PICO’s design should allow them to be added cleanly, without major changes to the components already developed.

The literature of parallel branch and bound is vast, and it is not possible to give a comprehensive review here. Two fairly comprehensive but not particularly recent surveys may be found in [14] and [23, Chapter 8]; [5] is more recent but less comprehensive survey.

The remainder of this paper describes the design of current components of PICO. Section 2 describes the overall design of PICO, including its class hierarchy and the separation of the package into serial and parallel layers. Section 3 discusses the design of the serial layer, which contains a number of novel features not present in earlier branch-and-bound “shells,” including the ability to use variable search “protocols,” and the key notion of subproblem state. Section 4 describes the parallel layer, and how to migrate an application from the serial to the parallel layer. The parallel layer implements a compound work distribution scheme that generalizes CMMIP’s, but can run on general hardware platforms. We also discuss the parallel layer’s use of multiple threads of control arbitrated by a non-preemptive “stride” scheduler, and the issue of terminating the computation. Section 5 describes a sample application of PICO to mixed integer programming, without cutting planes, and gives preliminary computational results on the “Janus” massively parallel computer, which consists of thousands of Pentium-II processors. Section 6 describes the application of the mixed-integer-programming code to a resource-constrained project scheduling problem for Pantex. We describe how to exploit problem-specific structure to improve the efficiency and useability of the basic search engine. Section 7 describes tools and techniques for debugging and insuring correctness. Section 8 gives conclusions and outlines future development plans for PICO.

2. The general design of PICO

PICO is currently structured as a C++ class library. It provides a hierarchically-organized set of capabilities which users may combine and extend to form their own applications. As with ABACUS, extending the core capabilities of PICO requires the development of derived classes that incorporate the additional required functionality. This design is in some sense more demanding than interfaces like MINTO, which simply require the user provide auxiliary functions that are linked into the executable. However, we believe that the class library approach is more powerful and flexible, allowing the use of multiple inheritance, which is critical to PICO’s design.

Figure 1 shows a simplified conceptual design, or *inheritance tree* of the library; elements with solid boundaries have been completed or are in an advanced state of development, while those with dashed boundaries are in the planning or early development stages. At the root of the inheritance tree is the *PICO core*, which provides basic capabilities for describing and parallelizing branch-and-bound algorithms. Branch-and-bound methods that have specialized bounding procedures not requiring direct use of linear programming can be

defined as direct descendents of the PICO core. Currently there is one such algorithm, for solving binary knapsack problems.

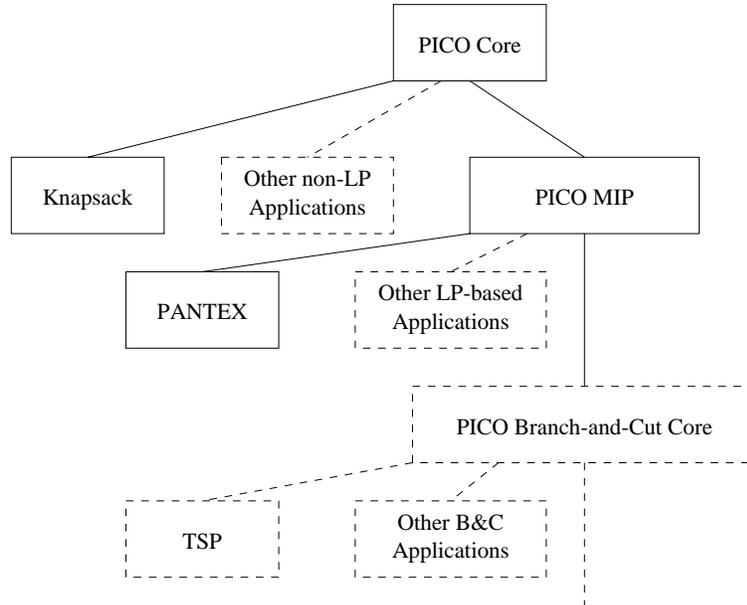


Figure 1. The current conceptual inheritance tree for PICO, in simplified form. Dashed lines indicate components in the planning or early development stages.

The PICO MIP package extends the PICO core by providing generic capabilities for solving mixed integer programs, using commercial LP solvers to solve their linear programming relaxations, as will be described in Section 5. For specialized MIP applications, the PICO MIP can itself be extended and refined by, for example, employing application-specific branching rules, fathoming rules, and heuristic methods for generating incumbent solutions. For example, it is straightforward to extend the PICO MIP to include LP-based approximation algorithms for scheduling problems, using the LP relaxation available at each node. We have exploited this flexibility for applications like the PANTEX production planning problem, which addresses a difficult scheduling problem within the U.S. Department of Energy. This application will be discussed in a separate paper.

We plan to extend the PICO hierarchy by creating a generic branch-and-cut capability that extends PICO MIP. This generic branch-and-cut could then be extended and refined as needed to handle specific applications such as the traveling salesman problem (TSP).

PICO consists of two “layers,” the *serial layer* and the *parallel layer*. The serial layer provides an object-oriented means of describing branch-and-bound algorithms, with essentially no reference to parallel implementation. The serial layer’s design, described in Section 3, has some novel features, and we expect it to be useful in its own right. For users uninterested in parallelism, or simply in the early stages of algorithm development, the serial layer

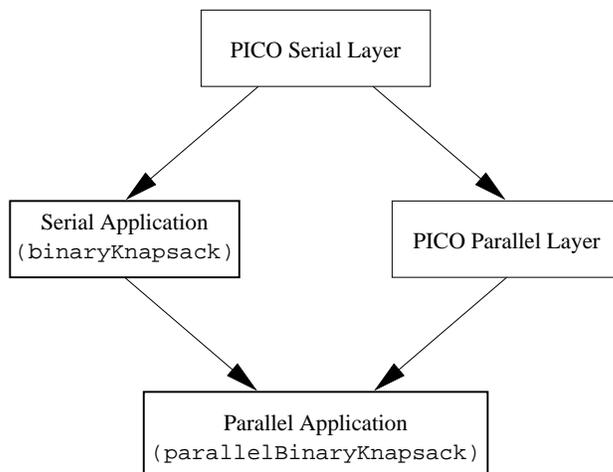


Figure 2. The conceptual relationships of PICO’s serial layer, the parallel layer, a serial application (in this case, `binaryKnapsack`), and the corresponding parallel application (in this case, `parallelBinaryKnapsack`).

allows branch-and-bound methods to be described, debugged, and run in a familiar, serial development 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, the user simply defines 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 PICO how to pack application-specific problem and subproblem data into MPI message buffers, and later unpack them.

Any parallel PICO 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. Application-specific refinements to the parallelization can then be added by the user, but are not required. Section 4 describes the parallel layer, and Figure 2 shows the conceptual relationship between the two layers, a serial application, and its parallelization.

PICO’s parallel layer was designed using a distributed-memory computation model, which requires message passing to communicate information between processors. We expect that this design will be effective on a wider range of systems than a design based on a shared-memory model. Although it is always possible to emulate distributed memory and message passing on hardware with memory-sharing capabilities, it is much more difficult to do the reverse. Emulating shared memory without hardware support may involve significant loss of efficiency and low-level control. Furthermore, shared memory, either hardware-supported or emulated, becomes rarer, more expensive, or both as the number of processors increases. Aside from portability considerations, we are particularly interested in the application of PICO on DOE’s massively parallel systems, for which distributed-memory parallel models have proven particularly effective.

The parallel layer is implemented using the MPI [37] standard for message-passing between processors. There are currently two portable, standard message-passing subroutine libraries for constructing distributed-memory programs, MPI and PVM [13]. We selected MPI because it is designed to be customized for maximum performance on MPP systems like Janus, the ASCI Red supercomputer. The design of PVM stresses the ability to operate on heterogeneous platforms, at some sacrifice in performance.

3. The serial layer

To define a serial branch-and-bound algorithm, a PICO user extends two fundamental PICO classes, `branching` and `branchSub`, the principal classes in the PICO serial layer. The `branching` class stores global information about a problem instance and contains methods that implement various kinds of serial branch-and-bound algorithms, as described below. The `branchSub` class stores data about each subproblem in the branch-and-bound tree, and it contains methods that perform generic operations on subproblems. This basic organization is borrowed from ABACUS [18], but it is more general, since there is no assumption that cutting planes or linear programming methods 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, undecided); this class describes each 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 class, in this case `binaryKnapsack`. Through this pointer, each subproblem object can find global information about the branch-and-bound problem. Finally, both `branching` and `branchSub` are derived from a common base class, `picoBase`, containing mainly common symbol definitions and run-time parameter objects. Figure 3 illustrates the basic class hierarchy for a serial PICO application.

3.1. Subproblem states

A novel feature of PICO, even at the serial level, 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 4.

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 PICO 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, PICO may decide to branch on it, a process also called “separation” or “splitting.” 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` contains three abstract virtual methods, namely `boundComputation`,

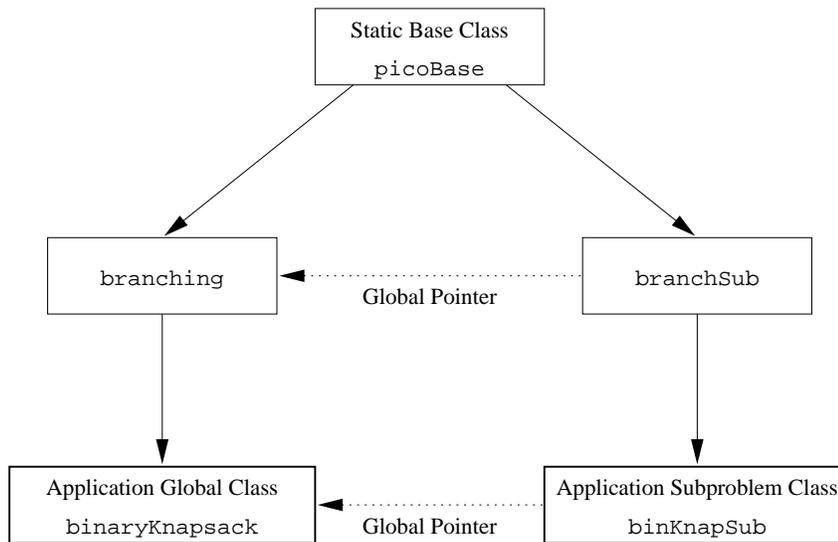


Figure 3. Basic class hierarchy for a serial PICO application (in this case, `binaryKnapsack`, with corresponding subproblem class `binKnapSub`).

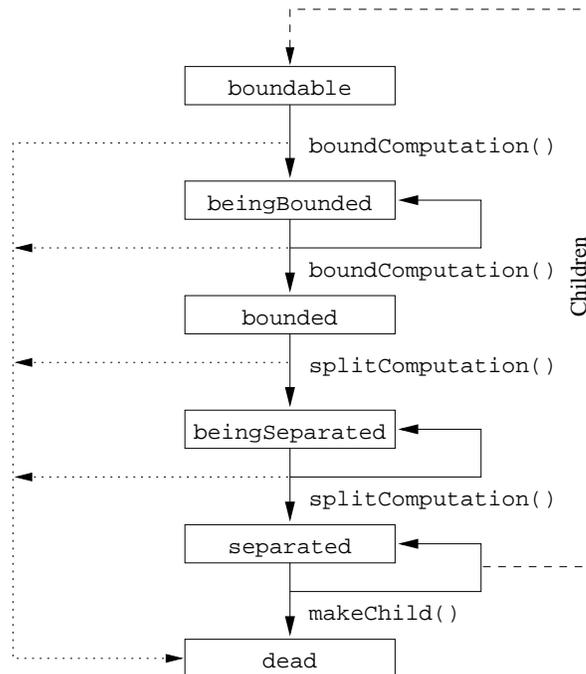


Figure 4. PICO's subproblem state transition diagram. It is possible that a single application of `boundComputation` may take a subproblem from the `boundable` state, through `beingBounded`, to `bounded`. Similarly, a single use of `splitComputation` may move a subproblem from `bounded`, through `beingSeparated`, to `separated`.

`splitComputation`, and `makeChild`, that are responsible for applying these state transitions to subproblems. PICO’s search framework interacts with applications primarily through these methods; defining a branch-and-bound application with PICO primarily 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 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 PICO to support branch-and-bound variants where one can suspend bounding one subproblem, set it aside, and turn one’s attention to another subproblem or task in the meantime. The subproblem’s bound, reflected in the data member `bound`, may change at each step of this process.

The `splitComputation` method’s job is similar to `boundComputation`’s, but it manages the separation process. Eventually it must change the problem state to `separated`, set the data member `totalChildren` to the number of child subproblems. Before that, however, it is allowed to return an indefinite number of times with the problem left in the `beingSeparated` state. This feature allows PICO 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 can be updated by `splitComputation` if the separation process yields additional information about it.

Finally, `makeChild` returns a single child of the subproblem it is applied to, which must be in the `separated` state. After the last child has been made, the subproblem becomes `dead`.

In addition to `boundComputation`, `splitComputation`, and `makeChild`, several additional virtual methods must to be defined to complete the specification of a branch-and-bound application. These definitions are described in Section 3.3.

3.2. Pools, handlers, and the search framework

PICO’s serial layer orchestrates serial 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, the “pool” and the “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: heap sorted by subproblem bound (biased slightly toward more integral problems if the bounds are all approximately equal), stack, and FIFO queue. If the user specifies the heap pool, then PICO 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. For particular applications, however, users may implement additional kinds of pools, thus specifying other search orders.

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 the user 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 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 often called “lazy” bounding (see for example [6]), because it views the bounding procedure as something time-consuming (like solving a large linear program) that should be delayed if possible. In the PICO framework, lazy bounding is implemented by keeping 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 [6] 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 more 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 nodes can be immediately fathomed, but has a larger task granularity, resulting in somewhat less potential for parallelism.

Because PICO’s serial layer stores subproblem states and lets the user specify a handler object, it gives users 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`, although the user is free to write additional handlers if greater flexibility is required.

The `eagerHandler` and `lazyHandler` methods implement eager and lazy bounding respectively by trying to keep subproblems in the bounded and boundable states respectively. Problems that become fathomed or dead anywhere in the process of applying the `boundComputation` and `splitComputation` methods are immediately discarded. Further, to permit users to pause the bounding or separation processes, any subproblem that remains in the `beingBounded` state after the application of `boundComputation`, or in the `beingSeparated` state following the application of `splitComputation`, is immediately returned to the pool.

The `hybridHandler` implements a strategy that is somewhere between eager and lazy bounding, and is perhaps the most simple and natural given PICO’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.

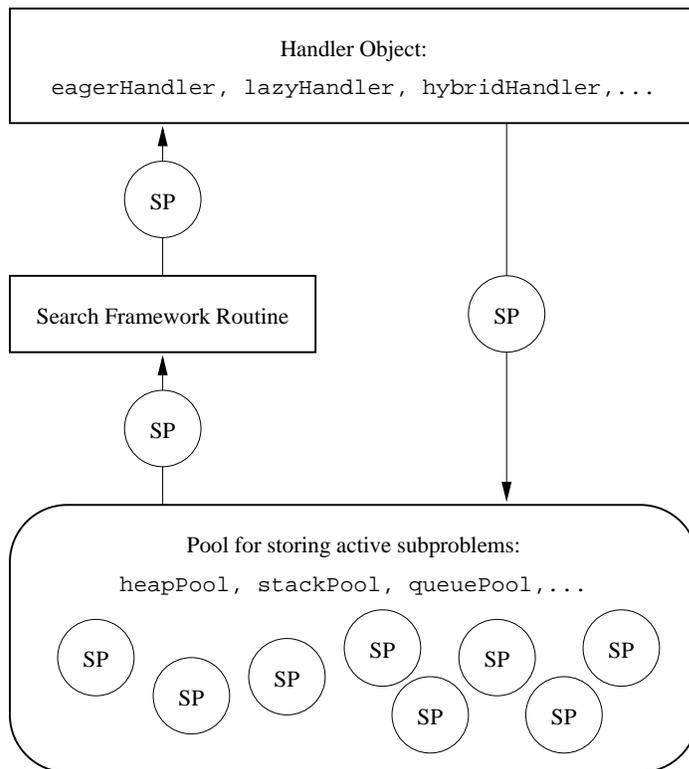


Figure 5. The search framework, pool, and handler. Each “SP” indicates a branch-and-bound subproblem.

It discards any subproblems becoming fathomed or dead at any point in this process.

The combination of multiple handlers, multiple pool implementations, and the user’s freedom in implementing `boundComputation` and `splitComputation` gives rise to enormous flexibility in the kinds of branch-and-bound methods that the serial layer can implement. Figure 5 depicts the relationship of the search framework, pool, and handler.

3.3. Serial layer virtual methods and run-time parameters

In addition to `boundComputation`, `splitComputation`, and `makeChild`, there are a number of additional abstract virtual methods in classes `branching` and `branchSub` that the user must define in order to fully describe an application of the PICO serial layer. The two classes also have a large number of other virtual methods that may be overridden at the user’s option. Table 1 describes all the required virtual methods and the more commonly-overridden optional ones. The most noteworthy are `candidateSolution`, `updateIncumbent`, and `incumbentHeuristic`, all members of `branchSub`.

The `candidateSolution` method tells the PICO search handlers whether a bounded subproblem needs to be separated at all. If this method returns `TRUE`, PICO assumes that the computed bound is in fact the objective value of the best feasible solution within the portion of the search space corresponding to the subproblem. If this bound is better than the current incumbent, the handler calls `updateIncumbent` to replace the current incumbent with the solution corresponding to the subproblem. By default, `updateIncumbent` simply stores the corresponding objective value; in most applications, the user will override this function to also store a representation of the solution for possible later output.

The `incumbentHeuristic` method provides a way for the user to specify a heuristic that takes a subproblem for which `candidateSolution` returns `FALSE` and attempts to perturb it into a feasible solution. In applications with linear-programming-based bounds, for example, `incumbentHeuristic` might try to round the fractional variables found in the linear programming relaxation. If it succeeds in finding a solution better than the incumbent, the heuristic should call `updateIncumbent`. PICO’s handlers only call `incumbentHeuristic` for a subproblem if the method `haveIncumbentHeuristic` returns `TRUE`. The default implementation of `haveIncumbentHeuristic` always returns `FALSE`.

PICO also provides a general mechanism for specifying run-time parameters. Table 2 shows the limited number of parameters that control the operation of serial PICO. Technically, the parameters are static objects that are members of the base class `picoBase`. User applications can add an unlimited number of their own run-time parameters, so long as their names are different from those in `picoBase`.

3.4. Memory management

Managing the pool involves frequent allocation and deallocation of small pieces of memory. This can incur a significant time overhead from system calls, especially on a parallel machine. Perhaps even more importantly, in systems such as Janus, the constant memory overhead for allocation from the system heap can be close to the size of the memory request, which halves the useable memory.

Therefore, PICO has its own memory management system for small, regular items such as objects for pools or subproblem tokens (see section 4.2). PICO requests memory in large blocks, and subdivides these blocks. This effectively eliminates the memory overhead and almost eliminates calls to the system heap, at the cost of managing freelists.

Required virtual method definitions: class branching

<code>readIn</code>	Read problem instance data from the command line and/or data file.
<code>blankSub</code>	Construct an empty subproblem.

Required virtual method definitions: class branchSub

<code>setRootComputation</code>	Turn a blank subproblem into the root problem.
<code>boundComputation</code>	Compute (perhaps only partially) the bound of a subproblem in the <code>boundable</code> or <code>beingBounded</code> state.
<code>splitComputation</code>	Separate (perhaps only partially) a subproblem in the <code>bounded</code> or <code>beingSeparated</code> state.
<code>makeChild</code>	Create a child of a subproblem in the <code>separated</code> state.
<code>candidateSolution</code>	Return <code>TRUE</code> if a <code>bounded</code> subproblem does not need further separation.

Optional virtual method definitions: class branching (selected)

<code>preprocess</code>	Preliminary computation before starting to search
<code>aPrioriBound</code>	“Quick and dirty” bound on the best possible solution (<i>e.g.</i> for knapsack, the sum of all item values).
<code>initialGuess</code>	Initial heuristic feasible solution value (<i>e.g.</i> for knapsack, the result of a simple greedy heuristic).
<code>haveIncumbentHeuristic</code>	Return <code>TRUE</code> if there is a heuristic for forming possible feasible solutions from <code>bounded</code> subproblems.
<code>serialPrintSolution</code>	Write incumbent solution to an output stream.

Optional virtual method definitions: class branchSub (selected)

<code>incumbentHeuristic</code>	Attempt to produce a feasible solution from the current (<code>bounded</code>) subproblem.
<code>updateIncumbent</code>	Store a new incumbent.

Table 1

Virtual method members of the `branching` and `branchSub` classes. Derived classes should also have their own constructors and destructors.

Name	Meaning	Default
<code>statusPrintFrequency</code>	Number of subproblems to bound between status printouts	1000
<code>depthFirst</code>	Use a stack as the pool, causing depth-first search	FALSE
<code>breadthFirst</code>	Use a queue for the pool, causing breadth-first search	FALSE
<code>lazyBounding</code>	Use the lazy bounding handler	FALSE
<code>eagerBounding</code>	Use the eager bounding handler	FALSE
<code>relTolerance</code>	A subproblem may be fathomed if its bound is within this factor of the incumbent objective value	10^{-7}
<code>absTolerance</code>	A subproblem may be fathomed if its bound is within this absolute distance of the incumbent objective value	0
<code>validateLog</code>	Causes quality-control output to be dumped to a file for later analysis	FALSE

Table 2

Run-time parameters defined in the static base class `picoBase`, which control the generic operation of the serial layer. The default pool is a heap, which causes best-first search, and the default handler is `hybridHandler`.

4. The parallel layer

The parallel layer’s capabilities are encapsulated 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, static base class `parallelPicoBase`, whose function is similar to `picoBase`, containing mainly common symbol and run-time parameter definitions. 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 public base classes. We call this class the *global parallel class*. For each problem instance, the information in the global parallel class is replicated on every processor.

The global parallel class’s 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 6 depicts the entire inheritance structure for the parallel knapsack application.

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, a few more methods must be defined, as summarized in Table 3.

First, the parallel application global and subproblems classes both require constructors and destructors. However, these methods are essentially trivial to define: the destructors may have empty bodies, and the constructors may simply invoke the constructors for their underlying classes. For technical reasons, the user must define two related methods, `blankParallelSub` in the global class, and `makeParallelChild` in the subproblem class. These methods fulfill the same roles as `blankSub` and `makeChild`, respectively, but in a parallel setting. Typically, these methods do nothing but call the constructor for the parallel subproblem class.

The packing and unpacking virtual methods are of greater interest. The parallel global and subproblem objects each require the definition of two methods, `pack` and `unpack`. The `pack` method is responsible for packing all the application-specific data in an instance of the class into a buffer suitable for sending between processors using the MPI datatype `MPI_PACKED`. The `unpack` method is responsible for unpacking the same data from an MPI receive buffer, and reconstituting the data members of the class instance. The parallel global class `pack` and `unpack` methods are used to distribute the global problem definition when setting up PICO, while the subproblem class `pack` and `unpack` methods are used to send subproblems from one processor to another.

Optionally, the user may also define a `packChild` method, whose functionality is equivalent to `makeChild` followed by a `pack` on the resulting subproblem. This method is discussed

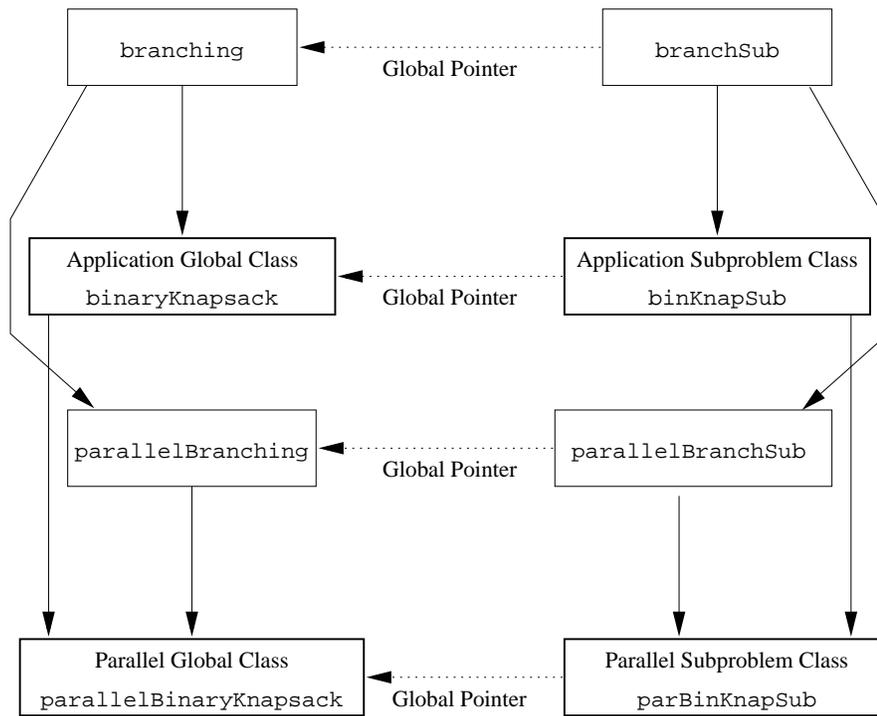


Figure 6. Inheritance structure of the parallel knapsack application. Other parallel applications are similar.

further in Section 4.4.8.

In addition to the `pack` and `unpack` methods, the parallel application must define one additional packing-related method, `spPackSize`. This method, a member of the global parallel class, should return an integer giving the maximum number of bytes required to buffer the application-specific data for a single subproblem. It is allowed to use any information in the global parallel class, and will not be called until the global information has been replicated on all processors. The parallel layer uses this method when allocating buffer space for incoming subproblem information.

We now describe the operation of the parallel layer. The layer is very flexible, but as a result it is also quite complex. For reasons of space, our description is somewhat abbreviated.

Required virtual method definitions: class `parallelBranching`

<code>pack</code>	Pack application-specific global problem information into a buffer.
<code>unpack</code>	Unpack application-specific global problem information from a buffer.
<code>spPackSize</code>	Estimate the maximum buffer space needed to pack the application-specific portion of one subproblem.
<code>blankParallelSub</code>	Construct an empty subproblem.

Required virtual method definitions: class `parallelBranchSub`

<code>pack</code>	Pack application-specific subproblem data into a buffer.
<code>unpack</code>	Unpack application-specific subproblem data from a buffer.
<code>parallelMakeChild</code>	Construct a single child of the current subproblem, which must be in the <code>separated</code> state. Similar to <code>makeChild</code> , but returns an object of type <code>parallelBranchSub</code> .

Table 3
Abstract virtual methods of the `parallelBranching` and `parallelBranchSub` classes.

4.1. Processor clustering

PICO’s parallel layer employs a generalized form of the processor organization used by the later versions of CMMIP [8,10]. 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. Unlike CMMIP, however, 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; CMMIP only permitted this kind of function overlap in clusters consisting of just one processor.

Three run-time parameters, all defined in `parallelPicoBase`, govern the partitioning of processors into clusters: `clusterSize`, `numClusters`, and `hubsDontWorkSize`. First PICO finds the size k of a “typical” cluster via the formula

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

where \bar{p} is the total number of processors. Processors are then gathered into clusters of size k , except that if k does not evenly divide n , the last cluster will be smaller. In clusters whose size is greater than or equal to `hubsDontWorkSize`, the hub processor is “pure,” that is, it cannot 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 from its workers, slowing down the entire cluster. In this case, a “pure” hub is more advantageous.

The value of `hubsDontWorkSize` must be at least 2, so it is impossible to form a cluster with no workers.

4.2. Tokens and work distribution within a cluster

Unlike some “master-slave” implementations of branch and bound, each PICO worker maintains its own pool of active subproblems. This pool may be any of the kinds of pools described in Section 3.2, although all workers use the same pool type. Depending on how various run-time parameters are set, however, the pool might be extremely 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. There are currently three parallel search handlers, called `eagerHandler`, `lazyHandler`, and `hybridHandler`, which behave in a similar manner to their respective serial counterparts, but with the additional ability to *release* subproblems from the worker to the hub.

For simplicity throughout the remainder of this section, we describe these handlers for a configuration with a single cluster consisting of all available processors.

4.2.1. Random release of subproblems

The parallel version of `eagerHandler` decides whether to release a subproblem as soon as it has become bounded. The parallel version of `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 $1/k$ of the total work in the cluster, then it 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`.

4.2.2. Subproblem tokens

When a subproblem is released, only a small portion of its data, called a *token* [33,7], 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. On a typical 32-bit processor, a token requires 48 bytes of storage, much less than the full data for a subproblem in most applications. Since the hub receives only tokens from its workers, these space savings translate into reduced storage requirements and communication load at the hub.

When making tokens to represent new, `boundable` subproblems, the parallel version of `lazyHandler` and `hybridHandler` 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.

4.2.3. Hub operation and hub-worker interaction

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

The hub processor maintains a pool of subproblem tokens that it has received from workers. Again, this pool may be any one of the pools described in Section 3.2. 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 sufficiently far from the incumbent). Quality balancing is controlled by the run-time parameter `qualityBalance`, which is `TRUE` by default. Workload quantity evaluation is via the run-time 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 [7,8,10,33]. This process will be described in more detail in Section 4.4.8.

Only one subproblem is dispatched at a time; if a token in the hub pool represents several problems, the hub splits it into two, with one token representing one subproblem, and the other any remaining subproblems. It sends the former token and retains the latter.

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 reimplementing or recompiling.

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.

4.2.4. Rebalancing

If the probability of workers releasing their subproblems is set too low, or the search process is nearing completion, workers in a cluster have workloads that are seriously out of balance, yet the hub’s token pool is 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.

4.3. Work distribution between clusters

With any system-application combination, there will be a limit to the size of a cluster that can operate efficiently, even if its hub does not have any worker responsibilities. Depending on the application and the hardware, the hub may simply not be able to keep up with all the messages from its workers, or it may develop excessively long queues of incoming messages.

At this point, one option is to adjust the PICO’s run-time parameters to reduce the amount of intra-cluster communication, but if communication is reduced too much, the hub may have difficulty maintaining a proper load balance in the cluster. To be able to use all the available processors, it may then be necessary to partition the system into multiple clusters. Another reason for such partitioning is that particular classes of applications may

simply perform better with the more randomized search pattern that results from multiple clusters [8,10].

PICO’s method for distributing work between clusters resembles CMMIP’s [8,10], with some additional generality: there are two mechanisms for transferring work between clusters, *scattering* and *load balancing*. Scattering comes into play when subproblems are released by the handlers. If there are multiple clusters, and a worker has decided to release a subproblem, the handler makes a second random decision as to whether the subproblems should be released to the worker’s own hub or to some other, randomly-chosen cluster’s hub. This random decision is controlled by the apparent workload of the cluster relative to the entire system, and the parameters `minNonLocalScatterProb`, `targetNonLocalScatterProb`, and `maxNonLocalScatterProb`.

To supplement scattering, PICO also uses a form of “rendezvous” load balancing that resembles CMMIP’s [8,10]; [26] and [19] 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

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

Here, $C(c, t)$ denotes the set of subproblems that c ’s hub knows are controlled by the cluster at time t , $\bar{z}(c, t)$ 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 ρ is either 0, 1, or 2, at the discretion of the user, much as in CMMIP. If $\rho = 0$, only the number of subproblems in the cluster matters. Values of $\rho = 1$ or $\rho = 2$ give progressively higher “weight” to subproblems farther from the incumbent. The default value of ρ is 1.

The rendezvous load balancing mechanism organizes all the cluster hub processors into a balanced binary tree. Periodically, messages “sweep” through this entire tree, from the leaves to the root, and then back down to the leaves. These sweeps are organized into repeating “rounds,” each consisting of three sweeps, *synchronization*, *survey*, and *balance*, as follows:

Synchronization Sweep: Each hub waits until its cluster appears to be idle, its cluster has bounded a sufficient number of subproblems, or a sufficient amount of time has passed (“sufficient” is defined by run-time parameters). Once these conditions are met, it makes sure that it has received a message from each of its child hubs, if any. Once all such messages have been received, the hub then sends a message to its parent, unless it is at the root of the tree. Once the root receives messages from all its children, it initiates a broadcast down the tree that the rest of the load balancing process may proceed. This technique is designed so that the survey sweep, which follows immediately, can start in a roughly synchronized way.

Survey Sweep: Each processor waits to receive workload information from its children, if any. It adds these workloads to its own current workload, and forwards the result up

the tree. The root is then able to compute an approximate total workload for the system, which it broadcasts down the tree. The messages in this sweep also contain information on the incumbent values $\bar{z}(c, t)$ used to compute the cluster workloads, and other miscellaneous information that is aggregated as the messages pass up the tree. If there is any mismatch between the incumbent values used at the various clusters, or a similar mismatch between the hub and any worker within a cluster, the survey is repeated immediately. Such a mismatch means that a new incumbent value is currently being broadcast (as will be described in Section 4.4.7), and some processors have not had an opportunity to prune their subproblem pools to reflect this new incumbent. The aggregate workload information is therefore inconsistent, and must be gathered again. Once consistent information has been gathered, the balance sweep may begin.

Balance Sweep: First, each processor determines whether its cluster should be a *donor* of work, a *receiver* of work, or (typically) neither. Donors are clusters whose workload exceeds the average by a factor of at least `loadBalDonorFac`, while receivers must be below the average by at least `loadBalReceiverFac`. A single message sweep of the tree then counts the total number of donors d and receivers r , and also assigns a unique donor and receiver number. The first $y = \min\{d, r\}$ donors and receivers then “pair up” via a rendezvous procedure involving $2y$ point-to-point messages; see [15, Section 6.3] or [8,10] for a more detailed description of this process. Within each pair, the donor sends a single message to the receiver, containing enough subproblem tokens to approximately equalize their workloads. Thus, the sweep messages are followed by a possible additional $3y$ point-to-point messages. After these messages, if any, the entire load balance round process repeats, starting with another synchronization sweep.

Under certain conditions, including at least once at the end of every run, a *termination check* sweep is substituted for the balance sweep. This mechanism will be described in Section 4.5.

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.

4.4. Thread and scheduler architecture

From the preceding discussion, it is clear that the parallel layer requires each processor to perform a certain degree of multitasking. CMMIP handled multitasking by combining user-level interrupts for the highest-priority tasks with an *ad hoc* round-robin scheme for the remaining ones. The former mechanism was not portable, and the latter lacked flexibility (for example, a hub could not simultaneously serve as a worker and still control other worker processors). Instead, PICO defines a *thread* of control for each required task on a processor, and manages these threads through a *scheduler* module. The threads share a common global memory space through a pointer to the instance of `parallelBranching` being solved.

PICO is not truly multithreaded code, however. We do not use POSIX or other standard thread packages, and, on each processor, PICO appears to the operating system as only a single thread of control. The scheduler is non-preemptive, much like the schedulers in the Macintosh and Windows 3.x operating systems: the scheduler explicitly calls each thread as

a subroutine, and the thread restores control to the scheduler, only when it is ready, through a standard subroutine return. There are several reasons why we took this approach:

- Many versions of MPI are not “thread-safe” or make their own use of threads. If PICO used “true” threads, it could not be ported to such systems.
- The approach simplified debugging and development.
- Since PICO’s tasks can be trusted to cooperate, a non-preemptive discipline is adequate to control them.
- Since threads only return control at times of their choosing, they can leave global data structures in a known state, and there is no need to worry about memory access conflicts and locks.
- The approach allowed us to use our own customized scheduling algorithm.

PICO contains a general-purpose scheduler, which is designed to be useful for other applications as well. We now describe the general algorithm used by the scheduler. Further detail for an earlier but similar version of the scheduler may be found in [11].

4.4.1. The scheduling algorithm

At any given time, each thread is in one of three states, *ready*, *waiting*, or *blocked*. Only threads in the ready state are allowed to run. Threads in the waiting state are waiting for the arrival of a particular kind of message, as identified by an MPI tag. The scheduler periodically tests for message arrivals, and changes thread states from waiting back to ready as necessary. Threads in the blocked state are waiting for some event other than a message arrival. The scheduler periodically polls these threads by calling their `ready` virtual methods; when a blocked thread’s `ready` method starts returning `TRUE`, the scheduler changes it back to the ready state.

Threads are organized into *groups*, each group having its own priority. Group priorities are absolute, in the sense that the scheduler only runs threads from the highest priority group that contains ready threads. Only if all higher-priority thread groups are empty will the scheduler permit thread in lower groups to run.

Each group may use one of two scheduling disciplines. The first is a simple round-robin scheme, in which ready threads are selected in a repeating cyclic order. The second possibility is a variant of *stride scheduling* [40,20]. This scheme allows the user to specify the approximate fraction of CPU resources that should be allocated to each thread.

In the stride scheduling discipline, each thread i has a *bias* b_i that specifies its importance. Let R denote the *ready list*, the set of ready threads in the highest nonempty group, then the fraction of the CPU devoted to thread $i \in R$ will be approximately $b_i / \sum_{j \in R} b_j$. Each thread also has a value v_i which specifies its current position in the run queue. The scheduler runs the ready thread with the lowest v_i , and when the thread returns, updates $v_i \leftarrow v_i + u/b_i$, where u is the amount of time just used by the thread.

All stride-scheduled threads start with $v_i = 0$. When a waiting or blocked thread is about to enter the ready list again, its v_i is reset to $\max\{v_i, v_* + k_i\}$, where $v_* = \min_{j \in R}\{v_j\}$, and $k_i \in \mathfrak{R}$ may be user-specified. To prevent numerical precision problems, a constant is periodically subtracted from all the v_i , $i \in R$, so that $v_* = 0$.

4.4.2. Thread group organization and thread types

The threads used by PICO belong to two broad categories: *message-triggered* threads and *compute* threads. There are two thread groups: the message-triggered threads occupy the higher-priority group, which uses round-robin scheduling, and the compute threads make up the lower-priority “base” group, which uses stride scheduling.

A message-triggered thread typically spends most of its time waiting for messages. When a message with the right tag arrives, the scheduler changes the thread’s state from waiting to ready. Since message-triggered threads are in the high-priority group, they tend to run soon after they become ready. Once it runs, the thread processes the message, issues a nonblocking receive for another message, changes its state back to waiting, and returns.

Compute threads are usually in the ready state, but may be in the blocked state if they have exhausted all their available work. These threads are scheduled in the proportional-share manner described above, so long as no message-triggered threads need to run. By default, all of PICO’s compute threads contain logic to actively manage their *granularity*, that is, the amount of time u they consume before returning to the scheduler. There is a run-time parameter `timeSlice` which specifies an ideal time quantum for compute threads. Compute threads try to manage the amount of work they do at each invocation so that the average value of u is approximately equal to `timeSlice`. The best value of `timeslice` depends on the hardware, the MPI implementation, and the application. A very small value means that message-triggered threads will run soon after their messages arrive, giving fast communication response, whereas a large value will minimize the overhead expended on the scheduler and entering and exiting compute threads. Ideally, one attempts to balance these two goals; in preliminary testing, we have had good results with a value approximately 20 times the time the scheduler needs to check for arriving messages.

4.4.3. Beginning the parallel search

To read in a problem instance, the parallel layer uses the `readAndBroadcast` method. `ReadAndBroadcast` first uses the `readIn` method, inherited from the serial application, to construct the global class information on a single processor. It then uses the the global class `pack` method to copy this information to a buffer, which it then broadcasts. All other processors receive the broadcast, and then use the global class `unpack` method to construct their replicas of the global class object.

Once a problem has been created and replicated on all processors, the application calls the `parallelSearch` method to search for a solution. Before starting the scheduler, this routine first calls the `preprocess` virtual method on all processors. By redefining `preprocess` for the parallel global class, the application may parallize its preprocessing procedure; Section 5 gives an example of this technique.

`ParallelSearch` next initializes the scheduler, creating the thread groups and calling the virtual method `placeTasks` to create the threads and place them the groups. The default version of `placeTasks` should suffice for many applications, and we describe below all the threads that it creates. If the application requires additional threads, it can redefine `placeTasks` to call the default `placeTasks`, and then create and place any additional threads. Again, we will present an example in Section 5.

Once the scheduler has been initialized, the first worker in the first cluster creates a blank subproblem, calls `makeRoot` to turn it into the root problem, and then inserts the root

problem into its local pool. On all processors, `ParallelSearch` then calls the scheduler to begin running all the threads. On each processor, the scheduler then runs until some thread sets the scheduler’s global termination flag, after which `parallelSearch` exits.

We now describe all the threads created by the default version of `placeTasks`, as also depicted in Figure 7.

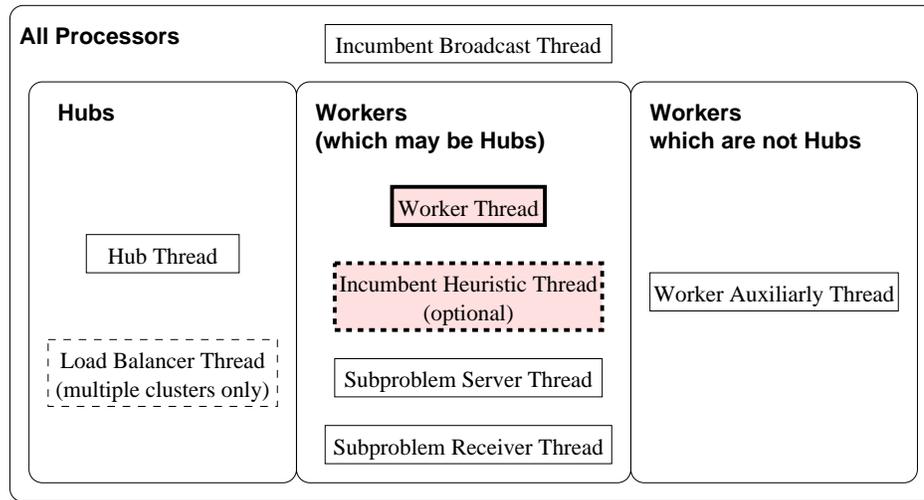


Figure 7. Threads used by the PICO core. Compute threads are shaded; all other threads are message-triggered.

4.4.4. The worker thread

The *worker thread* is a compute thread that is present on every worker processor. This thread simply extracts subproblems from the worker’s local pool and passes them to the search handler. If the local pool is empty, the worker thread enters the blocked state.

The worker thread attempts to regulate its granularity by adjusting the number of subproblems it passes to the handler before returning to the scheduler. For applications in which the bounding or separation procedure is very time consuming, and may need to be interrupted to allow message-triggered threads to run, the application may modify this granularity-adjustment scheme. Essentially, the virtual methods `boundComputation` and `splitComputation` can set the argument `controlParam` to some value proportional to the amount of work they have done. On subsequent calls, the granularity-control algorithm will pass, via this same argument, a suggested amount of work to perform.

The worker thread is also responsible for pruning the local subproblem pool and server pool on its processor. If running on a processor that is also a hub, the worker thread also calls the method `parallelBranching::activateHub` before returning control to the scheduler. This call allows the hub logic to respond to any changes in the cluster’s load resulting from the work just performed, and is described in more detail in Section 4.4.6.

4.4.5. The incumbent heuristic thread

The *incumbent heuristic thread* is a second, optional compute thread. It is only created on worker processors, and only if the both the application's `haveIncumbentHeuristic` virtual method returns `TRUE` and the run-time parameter `useIncumbentThread` is also set to `TRUE`. This thread's task is to search for better incumbent solutions. The algorithm and granularity-adjustment procedure used are entirely application-specific.

When any of the parallel search handlers move a subproblem to the `bounded` state, and the incumbent heuristic thread exists, they call the `feedToIncumbentThread` virtual method, a member of the `parBranchSub` class. This method can decide whether the subproblem partial solution is of interest to the incumbent heuristic, and, if so, can copy any necessary data to the incumbent heuristic's application-specific internal data structures. There is also a `quickIncumbentHeuristic` virtual method which may be called for any `bounded` subproblem. This method is meant for a "quick and dirty" procedure not requiring a separate thread (such as filling out a knapsack by the greedy method, or rounding up a fractional solution to a set covering problem).

On worker processors, the scheduler uses stride scheduling to arbitrate between the worker and incumbent heuristic threads. The biases of these threads are controlled by the run-time parameters `boundingPriorityBias` and `incSearchPriorityBias`, respectively. In the near future, we plan to add a feature whereby these biases may be dynamically adjusted throughout the course of a run. This technique will allow applications to make heavy use of the incumbent heuristic early in a run, when it is important to locate good incumbents, and then phase it out as the run progresses and it is better to concentrate on proving optimality of the current incumbent.

4.4.6. The hub thread

The *hub thread* is a message-triggered thread that runs on hub processors, and listens for messages with the tag `hubTag`. These messages may originate from any worker in the system. These messages contain workload status information, tokens of subproblems that are being released or scattered to the hub, and/or acknowledgements of receipt of subproblems dispatched from the hub.

When it awakes, the hub thread processes the contents of one of these messages, making the requisite changes in hub logic data structures. It then calls the method `activateHub`.

Calling `activateHub` triggers all the functions of the hub, including pruning the hub's pool of active tokens, distributing subproblems to any deserving workers, and possibly sending messages to workers informing them of the workload distribution in the cluster. When an event occurs that might alter the workload situation in the cluster, `activateHub` may be called by any thread running on a hub processor, and not just the hub thread.

4.4.7. The incumbent broadcast thread

The *incumbent broadcast thread* is a message-triggered thread that runs on all processors, and listens for *incumbent broadcast* messages. Each processor stores both the best objective value it currently knows for the incumbent, `incumbentValue`, and the rank of the processor that generated that value, `incumbentSource`. PICO's incumbent broadcasting scheme is similar to CMMIP's: when a new incumbent is found, one uses the `parallelBranching` class `signalIncumbent` method to begin the broadcast. The incumbent broadcasting procedure organizes all processors into a balanced tree rooted at the initiating processor. The tree's

radix may be specified by a run-time parameter; the default is a binary tree. The broadcast messages contain the objective value of the newly-found incumbent, and the processor number of the tree root.

When the incumbent thread receives an incumbent broadcast message, it compares the message’s objective value to the incumbent value currently known at the current processor. If the received value is worse, the incumbent thread does not attempt to continue the broadcast. If the two values are equal, it then compares the processor rank of the processor initiating the broadcast to `incumbentSource`. Only if the broadcast root is smaller than `incumbentSource` will the thread attempt to continue the broadcast. This procedure guarantees that if several processors simultaneously try to broadcast incumbents, that one of the broadcasts with the best value will reach all processors, while the others will be aborted.

If the broadcast should continue, the incumbent broadcast thread updates the local values of `incumbentValue` and `incumbentSource` to those in the message, and forwards this information along the broadcast tree. It sets flags forcing the worker thread to become ready, and then prune the server and local worker pools. The incumbent broadcast thread also sets a similar flag to force the hub thread, if present, to perform pruning of the hub pool.

4.4.8. The subproblem server thread

The *subproblem server* is a message-triggered thread that runs on all workers, and listens for work dispatch messages from the hubs. These messages contain a subproblem token and the processor rank of a worker to which the corresponding subproblem should be delivered. The subproblem server thread’s task is to deliver the full information about the specified subproblem to the worker in question.

Upon receiving a message, the subproblem server thread decodes the subproblem token and the rank w of the target worker. It also checks the bound on the token to make sure that the problem cannot be fathomed because of some recently broadcast incumbent value. If the subproblem can be fathomed, it sends an acknowledgement message to the originating hub indicating that the subproblem was properly received, but does not bother actually trying to send the subproblem data to the worker. If, as usual, the subproblem cannot be fathomed, the thread then calls the method `parallelBranching::deliverSP` to deliver the subproblem data

If a hub is also a worker, and it wishes to send some other worker a subproblem stored in its own server pool, the hub simply calls the `deliverSP` method directly, rather than sending a message to itself.

The `deliverSP` method first matches the token with the corresponding subproblem P on the worker; this step is very efficient, because the token contains the memory address of P . `DeliverSP` then separately handles four possible cases, depending on whether the token is a “child” or “self” token, and whether the target worker w is the same as the current processor p , or some other worker. “Self” tokens refer directly to some subproblem in the server pool, while “child” tokens refer to a child of such a subproblem. The cases are handled as follows;

1. If the token is a self token, and $w = p$, P is transferred from the server pool to the local worker pool, and is marked as “delivered.”
2. If the token is a self token, and $w \neq p$, the server thread uses the application’s subproblem `pack` method to pack P into a buffer, and sends this buffer to w . P is then

deleted from the server pool.

3. If the token is a child token, and $w = p$, the server thread uses `parallelMakeChild` to extract a child P' of P , and places P' in the the local worker pool, marking it as delivered. If P has no children left, it is deleted.
4. If the token is a child token, and $w \neq p$, the server thread uses the application's `packChild` method to pack a child P' of P into a buffer. The `packChild` method has the default implementation of creating a child with `makeParallelChild`, placing it in a buffer with `pack`, and then deleting it. However, the application is free to substitute a more efficient, application-dependent implementation. The buffer is sent to w , and P is deleted if it has no children left.

The messages sent in cases 2 and 4 have MPI tag `deliverSPTag`.

After a subproblem is marked as delivered on a worker that is not a hub, an acknowledgement for that subproblem is included in the worker's next communication to its hub. If the worker is itself a hub, the acknowledgement is entered directly into the hub data structures.

4.4.9. The subproblem receiver thread

The *subproblem receiver* thread is a message-triggered thread present on all workers, and listens for messages with the tag `deliverSPTag`. Upon receipt of such a message, the thread calls `blankParallelSub` and then the application's subproblem `unpack` method to recreate the data structures for the subproblem, which it then marks as delivered. If the subproblem cannot be fathomed, the thread inserts it into the local worker pool.

4.4.10. The worker auxiliary thread

The *worker auxiliary thread* is a message-triggered thread that exists only on workers that are not also hubs. It listens for messages with the MPI tag `workerTag`, which are sent by hubs to their workers. Each of these messages can contain one of three possible "signals" from the hub to the worker:

Load Information Signal: The message contains information on cluster and system-wide workloads. The worker auxiliary thread copies this information to the workers local data structures.

Termination Check Signal: This signal indicates PICO is double-checking whether the system is indeed fully idle and ready to terminate. The worker auxiliary thread immediately replies with a message containing a count of the total messages the worker has sent. The reason for this procedure will be described in Section 4.5 below.

Terminate Signal: The hub has determined that the branch-and-bound search has terminated. The worker auxiliary thread sets the scheduler's global termination flag. When the thread exits, the scheduler terminates, and the call to `parallelSearch` returns.

4.4.11. The load balancer thread

The *load balancer thread* orchestrates the load balancing scheme described in Section 4.3. It runs on all hub processor. It contains finite state machine logic to move all processor

approximately synchronously through the various message sweeps and other operations described in Section 4.3. It is basically a message-triggered thread that listens for various kinds of messages, depending on what phase of the load balancing procedure it is currently in. One exception is that it may enter the blocked state when at the beginning of the synchronize sweep, waiting for conditions to be right for another round of load balancing. The scheduler unblocks it when the sweep is allowed to start.

The load balancer thread is responsible for terminating the search computation, as described below. If there is only one cluster, then no load balancing is necessary, and termination is the thread's only function. In this case, it immediately puts itself in the termination check sweep mode, first listening for termination check reply messages from the workers.

4.5. Termination detection

Proper detection of termination can be a tricky issue in asynchronous distributed-memory computations. CMMIP's termination procedure relied on specific properties of the CM-5's communication hardware and operating system, and could not be generalized to PICO.

In parallel branch and bound, it is important to terminate as soon as, but not before, there are no active subproblems left to be bounded or separated anywhere in the system. In some implementations of MPI, it is also important that when a processor calls `MPI_Finalize` to terminate its computation, that it have received all messages sent to it by other processors, except any that were cancelled via `MPI_Cancel`. If not, `MPI_Finalize` may "hang" or generate system errors.

So, for PICO to be able to terminate, all worker subproblem pools and hub token pools must be empty, and all messages sent must be received. We call this situation *quiescence*.

4.5.1. The case of a single cluster

If there is only one cluster, quiescence is relatively straightforward to detect. The hub knows which subproblems it has assigned to which workers, and through the delivery marking and acknowledgement mechanism, which of these problems have been received. It also has recent workload information from each of its workers. Furthermore, the workload information reported by workers contains counts of total messages sent and received, so hubs can also detect messages in transit.

Once a worker becomes idle, it has no more subproblems in its active pool, and it reports its workload to its hub immediately. If it is idle and receives a message of any kind, it resends its idle report to the hub, with updated message send and receive counts included.

Suppose that the following conditions hold:

1. The hub has an empty token pool.
2. All the clusters workers have reported themselves idle.
3. All subproblems dispatched from the hub have been acknowledged as delivered.
4. All processors agree on the objective value of the incumbent, and which processor stores the incumbent.
5. The total counts of message sends and receives appear to match when summed over all processors in the cluster.

In this situation, no more work can possibly arise in the normal operation of the system, and no more messages can be sent by any of the PICO core threads. However, there is still a possibility of premature termination if any application-specific threads have messages in transit; even though the total count of messages sent and received may appear equal, messages may still be in transit due to the phenomenon of “aliasing,” as described in the next section. To check for this possibility, the hub sends a termination check signal to all workers. The workers’ replies to these messages wake up the hub’s load balancing thread, which double-checks the message counts, and terminates the computation if appropriate. This process is described in more detail immediately below.

4.5.2. Multiple clusters

When there are multiple clusters, properly detecting quiescence is more difficult, even if no application-specific threads are present. Basically, termination is detected at the end of the survey sweep, when the total workload information summed over all clusters is distributed to all hub processors. If all clusters have no active subproblems and the total counts of sent and received messages match, then it is likely that the search can terminate. Note that messages used by the load balancing process itself must necessarily be in balance at this point, and so do not need to be included in the message counts.

However, even if the survey sweep detects that all clusters appear to be idle and the total counts of sent and received messages match, it is still possible that the system is not quiescent. Thus, we call this state *pseudoquiescence*. The reason is that it is not possible to sample the message send and receive counts from all processors at exactly the same time. Thus, a message can contribute to the total reception count collected by the sweep, without yet contributing to the total send count. The reception of such a message can then masquerade as the reception of another message whose send operation is included in the count but has not been received even by the end of the sweep. Such “aliasing” can cause premature detection of termination. If such a message contained a scattered subproblem, then PICO might terminate with an incomplete proof of optimality, or possibly an incorrect solution. This phenomenon can also occur if there is only a single cluster, but there are application-specific threads that send interprocessor messages.

To prevent such premature termination, we use a variant of the “four counter” method due to Mattern [27], which seems to be the most efficient technique available (the name is misleading, since it is shown in the original reference [27] that the method can be implemented with only three counters). In PICO’s case, the procedure works as follows: at the end of the survey sweep, the load balancer threads on all hubs detect pseudoquiescence. Instead of proceeding to the balance sweep, they substitute a termination check sweep instead.

At the start of the termination check sweep, each hub sends a termination check signal message to all its workers (except itself, if it is also a worker). The worker auxiliary threads on these workers respond with their total message sent counts. Once all its workers have responded, each hub adds the counts for its entire cluster, including itself, together. The sweep messages now proceed, adding this information recursively up the cluster hub tree. The overall message count sums form at the the cluster tree root, and are broadcast back down the tree. If the total message sent count collected by the termination check sweep is the same as that collected by the immediately preceding survey sweep, then aliasing is impossible, and the system was actually quiescent at the end of the last survey sweep;

see [27] for a proof. In this case, the load balancer thread on each hub sends a termination signal message to all its cluster’s workers, and then sets the scheduler termination flag. If the counts do not match, then true termination has not occurred, and the load balancer thread simply commences another round of load balancing.

Note that if there is only one cluster, the “tree” used in the termination check “sweep” just consists of a single node and no edges. If the termination check fails in the one-cluster case, the load balancer simply starts another termination check sweep, listening for termination check reply messages from the workers. The signal to send these messages will come from the hub thread the next time it detects possible termination.

5. Application to mixed integer programming

To demonstrate how the PICO core can be used, we now describe the application of the PICO class library to the solution of general mixed integer programming problems, without the use of cutting planes. This application is the “PICO MIP” referred to in Section 2 and Figure 1.

We stress that this application is not yet meant to be a completely state-of-the-art MIP solver, as it is lacking a number of features present in the best commercial codes. At this point, we present it to illustrate how the PICO core can be easily extended to include additional, advanced features for applicaions, and to illustrate the degree of parallelism that PICO can acheive.

Technically, the PICO MIP can solve any problem in the industry-standard MPS format. For convenience, we will assume in our discussion that the problem being solved is to find $x \in \Re^n$ satisfying

$$\begin{aligned} \min \quad & c^\top x \\ \text{S.T.} \quad & Ax = b \\ & \ell \leq x \leq u \\ & x_j \text{ integer } \forall j \in Z, \end{aligned} \tag{1}$$

where $c \in \Re^n$, $b \in \Re^m$, A is an $m \times n$ matrix, $\ell \in [-\infty, +\infty]^n$, $u \in (-\infty, +\infty]^n$, $\ell \leq u$, and $Z \subseteq \{1, \dots, n\}$ is a nonempty set of indices of variables that are required to take whole-number values. Note that inequality constraints can easily be accomodated in this formulation by introducing slack variables, as is the case with most linear programming solver software.

In the standard branch-and-bound algorithm for this problem, the root problem is simply (1) with the integrality constraints removed. The remaining subproblems are similar, but with some of the lower bounds ℓ_j increased or upper bounds u_j decreased, for $j \in Z$. Let $\ell(P)$ and $u(P)$ denote the lower and upper bound vectors for any given subproblem P . The bound $z(P)$ for subproblem P is obtained by solving the corresponding linear program, yielding some linear programming solution $x(P)$. The value $z(P) = c^\top x(P)$ is a lower bound on the objective value any solution x of (1) that has $\ell(P) \leq x \leq u(P)$. If $x_j(P)$ is integer (to within some specified numerical tolerance) for all $j \in Z$, then $x(P)$ represents a feasible solution to (1) that dominates all other solutions with $\ell(P) \leq x \leq u(P)$.

If there exists any $j \in Z$ for which $x_j(P)$ is not integer, then the subproblem must be separated. We select some such j , call it $j(P)$, and create a *down child* subproblem P^- with $u_{j(P)}(P^-) = \lfloor x_j(P) \rfloor$ and an *up child* P^+ with $\ell_{j(P)}(P^+) = \lceil x_j(P) \rceil$.

5.1. Pseudocosts

One of the keys to making this “textbook” branch-and-bound method work efficiently in practice is to make a good choice of the branching variable $j(P)$ from among the set $J(P)$ of indices $j \in Z$ for which $x_j(P)$ is not integer.

The modeler may specify branching “priorities” to aid in this decision: for example, variables specifying whether or not a particular production plant is to be built would have higher priority than variables specifying which products would be made in each plant. However, priorities are not available for all problems, and often there are many eligible variables with the same priority.

To choose among branching variables, we use a time-tested technique employing *pseudocosts* [2]. At any time t , let $\mathcal{K}(t)$ denote the collection of all subproblems P for which $z(P)$ is known. Then define

$$\mathcal{S}_j^+(t) = \{P \in \mathcal{K}(t) \mid P^+ \in \mathcal{K}(t), j(P) = j\}.$$

The “up” pseudo-cost of variable x_j , $j \in Z$, at any time t such that $\mathcal{S}_j^+(t) \neq \emptyset$, is

$$\phi_j^+(t) = \left(\frac{1}{|\mathcal{S}_j^+(t)|} \right) \sum_{P \in \mathcal{S}_j^+(t)} \left(\frac{z(P^+) - z(P)}{\lceil x_j(P) \rceil - x_j(P)} \right).$$

This quantity attempts to measure how rapidly the subproblem optimal objective value increases, on average, as x_j is forced upward. We define the “down” pseudocost in a similar way, but this time tracking how the objective value changes as variables are forced downward:

$$\begin{aligned} \mathcal{S}_j^-(t) &= \{P \in \mathcal{K}(t) \mid P^- \in \mathcal{K}(t), j(P) = j\}. \\ \phi_j^-(t) &= \left(\frac{1}{|\mathcal{S}_j^-(t)|} \right) \sum_{P \in \mathcal{S}_j^-(t)} \left(\frac{z(P^-) - z(P)}{x_j(P) - \lfloor x_j(P) \rfloor} \right). \end{aligned}$$

The method for choosing a branch variable is similar to CMMIP’s: for each $j \in J(P)$, we calculate a “score” and branch on the variable maximizing the score. To compute the score, we use the pseudocosts to estimate the respective degradations D_j^+ and D_j^- in the objective value for the up and down children, via:

$$\begin{aligned} D_j^+ &= \phi_j^+(t) (\lceil x_j(P) \rceil - x_j(P)) \\ D_j^- &= \phi_j^-(t) (x_j(P) - \lfloor x_j(P) \rfloor). \end{aligned}$$

The score is then computed by

$$\sigma_j = \alpha_0 Q_j + \alpha_1 \min\{D_j^+, D_j^-\} + \alpha_2 \max\{D_j^+, D_j^-\},$$

where Q_j is the priority of variable j and α_0 , α_1 , and α_2 are specified via run-time parameters. Typically, α_0 is chosen very large, so that priority is the overriding consideration. Also, one typically sets $\alpha_2 = 0$, or at any rate $\alpha_2 \leq \alpha_1/10$. Thus, after priority, the next most important consideration is trying to simultaneously “push up” the bounds of both child subproblems.

A critical issue is what to do when $\mathcal{S}_j^+(t) = \emptyset$ or $\mathcal{S}_j^-(t) = \emptyset$. Here, we take a different approach than CMMIP, shown to be superior by Linderoth [24]. Every time the algorithm

encounters an index $j \in J(P) \subseteq Z$ that has not been fractional in any prior subproblem solution, it “probes” — that is, computes the objective values for — the subproblems \hat{P}_j^+ and \hat{P}_j^- that would result if j were the branching variable. These subproblems do not necessarily appear in the search tree unless j is later chosen to be the branching variable, but are immediately incorporated into the set $\mathcal{K}(t)$, and thus into the pseudocost calculations above, so there will be at least one element present in each of the sets $\mathcal{S}_j^+(t)$ and $\mathcal{S}_j^-(t)$. If either of the subproblems \hat{P}_j^+ or \hat{P}_j^- is infeasible, we narrow the bounds of the variable accordingly, and set the pseudocost to infinity. If both directions are infeasible for any variable, the problem has no solution. Once a pseudocost for a variable has a finite pseudocost, all previous infinite pseudocosts are treated as `inFeasFactor` times that finite pseudocost and any subsequent infeasible branches found during branching are treated as `inFeasfactor` times the current pseudocost.

During the initialization of pseudocosts for a variable, we adjust the bound of the subproblem to (in the case of minimization), the maximum objective value of the two branches if it is higher than the current bound. This means that each subproblem must store its parent’s LP bound for calculation of future pseudocosts, rather than using its parent’s bound.

5.2. Other serial aspects of the algorithm

Our algorithm incorporates several other features that are standard in “industrial-strength” MIP solvers. Before starting the search, we run a preprocessor, based on that in MINTO and PARINO [34,29,24]. This preprocessor removes some redundant constraints and fixes the values of some variables, if it can deduce the values they must take in the optimal solution. Variable x_j ’s value is fixed by setting $\ell_j = u_j$.

The algorithm also applies a standard “locking” procedure after solving the linear program associated with each subproblem. Let $\bar{z}(t)$ denote the objective value of the incumbent at time t . If the absolute value of the reduced cost of a nonbasic variable x_j , $j \in Z$, exceeds $\bar{z}(t) - z(P)$, then x_j may be fixed at its present value in all of P ’s descendants. This procedure is valid because any descendant solution with a different but still integral value of x_j would necessarily be fathomed. Again, the locking is accomplished by setting $\ell_j = u_j$.

There are a number of other features that are now becoming common in commercial MIP solvers, but are not yet present in our algorithm: cutting plane methods to improve the linear programming bounds, various kinds of rounding heuristics to obtain feasible solutions from subproblem solutions $x(P)$ that do not meet the integrality constraints, and repeated application of the preprocessor at branch-and-bound nodes. We plan to add these features in later implementations or derived applications. Of these features, only an incumbent heuristic was present in CMMIP; we plan to implement a more sophisticated approach.

5.3. Serial implementation

Mapping the MIP branch-and-bound algorithm to the PICO serial layer classes and virtual functions is fairly straightforward. Class `MILP` is derived from `branching`, and contains simple arrays for the vectors b , ℓ , and u , along with sparse matrix representation of c the matrix A . It also contains tables required to calculate and update the pseudocosts $\phi_j^+(t)$ and $\phi_j^-(t)$.

Subproblems are represented by the class `MILPNode`, which is derived from `branchSub`. Essentially, a subproblem P is completely described by the two n -vectors $\ell(P)$ and $u(P)$.

In addition, however, each subproblem object stores a compacted representation of a corresponding linear programming basis. For problems in the `boundable` state, this basis corresponds to the optimal solution of the parent problem. For the `bounded` and `separated` states, it describes the optimal solution to the problem itself.

The `preprocess` method for class `MILP` executes the preprocessing procedure, and `MILP`'s `readIn` method reads an MPS data file into the `MILP` data structures. `MILPNode`'s version of the `boundComputation` method uses a commercial linear programming package to calculate $z(P)$ and $x(P)$ for a subproblem. The linear programming solver is encapsulated in a special interface class, allowing different LP packages to be specified at compile time. At present, we are using ILOG, Inc.'s CPLEX 6.x packages, but we have also built encapsulations for DASH Optimization's XPRESS-MP package and the public-domain solver SoPLEX.

Except for the root problem, `boundComputation` always begins from the optimal basis of the parent problem, which greatly speeds the calculations. If the parent was the last problem run, then we don't need to reset the basis. This preserves the LP-solver internal state and is generally another significant factor faster than the case where we must load the parent basis. We call this favorable situation a *warm start*. For both root and non-root problems, the user can specify whether the optimization is via primal simplex, dual simplex, or a barrier method. The default for non-root problems is dual simplex.

Once the linear program has been solved, `boundComputation` executes the reduced-cost-based variable locking procedure and identifies the set $J(P)$ of variables violating the integrality constraints. The `candidateSolution` method for `MILPNode` simply returns `TRUE` if $J(P) = \emptyset$, and otherwise `FALSE`.

The `splitComputation` method for `MILPNode` scans $J(P)$ for any indices j that have not appeared in $J(P')$ for any prior subproblem P' . For each such index, it computes the objective value of the probe subproblems \hat{P}_j^+ and \hat{P}_j^- , using P 's optimal basis as a starting point for the first computation and thereafter using the existing basis from the solution of the previous closely-related subproblem. This procedure may be significantly more time consuming than the original bound computation itself, but should become increasingly rare as the computation progresses. After this probing process is complete, $\mathcal{S}_j^+(t), \mathcal{S}_j^-(t) \neq \emptyset$ for all $j \in J(P)$. `SplitComputation` then calculates the scores σ_j for all $j \in J(P)$ and chooses the branching variable index $j(P)$ to maximize σ_j .

Finally, `MILPNode`'s `makeChild` method creates child subproblems. It creates a fresh subproblem and copies the bound information $\ell(P)$ and $u(P)$ to the child, modifying the bound $\ell_{j(P)}(P^+)$ for the up child, and $u_{j(P)}(P^-)$ for the down child. `MakeChild` also copies P 's optimal basis information to the child.

Table 4 describes the runtime parameters for the serial MIP.

5.4. Parallel implementation

To make a parallel version of the MIP algorithm, we used the same procedure described in Figure 6 and at the outset of Section 4. We defined a parallel global class `parMILP` with `parallelBranching` and `MILP` as `virtual public` base classes. Further, we also defined a parallel subproblem class `parMILPNode` with `parallelBranchSub` and `MILPNode` as `virtual public` base classes. We also provided straightforward implementations of the constructors and destructors for these classes, along with the virtual methods described in Table 3. These definitions are sufficient to provide a working parallel implementation. Since there is at

Name	Meaning	Default
<code>pcostUseDistances</code>	Do we care how large the up/down movement is in calculating branching priority?	TRUE
<code>importSplitfac</code>	Weight on user-supplied priority in calculating branching priority	10^{10}
<code>nearIntSplitfac</code>	Weight on how close a variable is to integer in calculating branching priority	0.0
<code>objBestSplitFac</code>	Weight on the better direction for a variable (calculating branching priority)	10.0
<code>objWorstSplitFac</code>	Weight on the worst direction for a variable (calculating branching priority)	0.0
<code>upSplitFac</code>	Weighting for general preference to branch up first	5.0
<code>downSplitFac</code>	Weighting for general preference to branch down first	0.0
<code>tableInitFrac</code>	Fraction of noninitialized (pseudocost) integer-violating variables to initialize through probing	1.0
<code>infeasFactor</code>	Weighting for pseudocost updates from infeasible branches or those beyond cutoff	10
<code>relVarSelectTol</code>	We can branch on any variable who's score is within this factor of optimal 10^{-5}	
<code>sendSolutionToFile</code>	Write the solution (as a vector) to the file PICO-Solution. If already tracking solutions (see <code>checkFathomOnSolution</code>), append this solution if it's new	FALSE
<code>checkFathomOnSolution</code>	Exit with an error if the any of the solutions stored in PICO-Solution are fathomed while their value is better than the incumbent.	FALSE
<code>rootSimplexMethod</code>	Which simplex method to use to solve the root (primal, dual, barrier)	MILPNode::primal
<code>warmSimplexMethod</code>	Which simplex method to use when a node's parent was the last problem solved	MILPNode::dual
<code>nonwarmSimplexMethod</code>	Which simplex method to use when a node's parent was not the last problem solved	MILPNode::dual
<code>preprocessLP</code>	Run the MILP preprocessor	TRUE

Table 4

Run-time parameters defined in the branching class `MILP`, which control the selection of branching variables and other aspects of the mixed-integer-programming search and parameters from the subproblem class `MILPNode`.

present no incumbent heuristic in the serial application, there is currently no incumbent heuristic thread in the parallel version.

However, we chose to extend the basic parallelization in a two ways, both relating to pseudocosts. We expect this situation to be an example of a standard pattern: pseudocosts constitute a type of information that is not part of the incumbent or active subproblem pool, but is nevertheless global, in the sense that it is not localized within a particular subproblem. Such global information typically needs some kind of special treatment in a parallel implementation.

Consider how the default parallelization provided by PICO would operate in the case of the MIP algorithm we have just described. The pseudocost tables, needed to calculate $\phi_j^+(t)$ and $\phi_j^-(t)$, data members of MILP, will by default be maintained completely independently on each processor. Initially, the first worker in the first cluster solves the root problem P_0 , while the other workers remain idle because there is no incumbent heuristic. Typically, the set $J(P_0)$ of the root's integrality-violating variable indices will be large. To initialize the pseudocost information needed to split the root problem, the first worker must solve an additional $2|J(P_0)|$ linear programs (albeit from a good starting basis). During this time, all other processors will remain essentially idle, although the work could easily be partitioned into $2|J(P_0)|$ independent tasks.

Once the search tree starts to grow, and other workers become busy, a second source of inefficiency would arise. Because the pseudocost tables are maintained separately and independently on each processor, the pseudocost probing operation will be performed whenever a variable x_j , $j \in Z$, is detected to be fractional for the first time *on a given processor*. Thus, probing for any particular variable might occur as many as \bar{w} times, where \bar{w} is total the number of processors, as opposed to once in the serial layer implementation.

To obtain more parallelism at the outset of the algorithm, we designed the **preprocess** routine in **parMILP** so that it functions differently from MILP's. Recall that the parallel search calls **preprocess** before running the scheduler; furthermore, this call is executed on every processor. The parallel version of the preprocessor, **parMILP::preprocess**, starts by first calling the serial version **MILP::preprocess**, to eliminate redundant constraints and fix variables. This calculation is done redundantly on all processors.

Instead of returning at this point, however, the parallel MIP preprocessor now instructs all processors to solve the root problem's linear program. Again, this operation is done in parallel and redundantly on all processors. The preprocessor then identifies the set of integrality violating indices $J(P_0)$, which require $2|J(P_0)|$ linear programs to be solved to initialize the pseudocost tables. The preprocessor partitions these linear programs into \bar{p} approximately equal-sized groups, each of size approximately $2|J(P_0)|/\bar{p}$. In parallel, without redundancy, each of the \bar{p} processors solves the problems in one of these groups. The preprocessor then makes the combined results of these calculations collectively available to all processors via an **MPI_Allgather** communication operation. The preprocessor then returns. In this manner, the work required to separate the root problem is significantly parallelized.

ParMILPNode's version of **makeRoot** sets the state of the root problem to **bounded** instead of the usual value of **boundable**, since the work of bounding the root problem has already been performed. When the first worker first processes the root problem, it immediately performs separation and chooses a branch variable, a rapid operation since all the necessary pseudocost information is available.

To limit possible redundancy in initializing the pseudocost information for indices $j \notin J(P_0)$, we employ a second strategy. Whenever a worker probes a to initialize the pseudocost data for a variable x_j , it places the resulting information in a special buffer, as well as in its regular pseudocost tables. As soon as all newly-fractional variables have been probed for a given subproblem P , the worker broadcasts the buffer to all other workers, as recommended in [24]. The buffer is then reset to empty. Upon reception, all other workers incorporate this information into their own pseudocost tables, making it unnecessary for them to probe any of the variables in $J(P)$ in the future. Otherwise, however, pseudocost information is maintained completely separately by each processor.

Although substantial interprocessor communication is involved, each of the broadcast operations may prevent as many as $2(\bar{w} - 1)$ redundant linear program solutions. Each pseudocost pair is broadcast along a balanced tree consisting of all workers, with the originating worker at the root; the radix of this tree is controlled by the run time parameter `pCostTreeRadix`, which defaults to 2.

To receive and forward the messages required for pseudocost broadcasts, we introduce one additional thread, the *pseudocost broadcast thread*. This message-triggered thread listens for incoming pseudocost data and incorporates the contained data into the local pseudocost tables. If the current processor is not a leaf of the tree for the broadcast in question, it forwards the broadcast to its children. To include this thread in the scheduler, `parMILP` overrides `parallelBranching`'s default implementation of the virtual method `placeTasks`. The substitute implementation first calls the original implementation, in order to create all the standard threads. It then creates an additional thread object (of type `pCostCastObj`) and inserts it into the message-triggered, higher-priority thread group.

It is possible, under this scheme, that some variables may still be probed redundantly: several processors could encounter the same newly-fractional variable at about the same time, with one or more beginning to probe before the broadcast from the first one reaches them. In practice, we find that there is very little redundancy.

The code also includes an option whereby pseudocost information for an index $j \in Z$, may be broadcast by processor p not only the first time processor p encounters a fractional value of $x_j(P)$, but the first k times, where k is set by a run-time parameter. This generalization allows the code to better deal with problems where pseudocosts behave in an “unstable” way, but such problems appear to be rare in practice. Even with this generalization, the approach is considerably simpler than CMMIP's [8,10].

Finally, we note that the current version of the parallel MIP application does not pause either the bounding or separation, that is, `boundComputation` always completes bounding a subproblem, leaving it in the `bounded` state, and `splitComputation` always completes the separation process, leaving a subproblem in the `separated` state. In the future, if we observe situations where invocations message-triggered threads are unacceptably delayed by very long bounding or separation operations by the worker thread, we could alter our approach. For example, `boundComputation` could return, leaving a subproblem in the `beingBounded` state, after a fixed number of dual simplex pivots. Applying `boundComputation` again would resume the computation. A similar procedure could be applied when evaluating probe subproblems in `splitComputation`.

Table 5 describes the runtime parameters for the parallel mixed-integer-programming code.

Name	Meaning	Default
<code>pCostShareCutOff</code>	Stop broadcasting pseudocosts for a variable after we have this many values	2
<code>pCostMinBcastSize</code>	Minimum number of (up and down) pseudocosts to share at a time if none are new	5

Table 5

Run-time parameters defined in the class `parMilp` which control the level of pseudocost broadcasting.

5.5. Preliminary computational results

To illustrate the parallelism PICO can attain, we now describe the performance of the PICO MIP on the “Janus” ASCI Red supercomputer at Sandia National Laboratories [28]. This system consists of 4,536 nodes, each with two 333-megahertz Pentium II processors and 256 megabytes of RAM. By default, one processor on each node functions as a compute processor, and the second as a communications processor for interacting with the internal network. Optionally, in what is called “virtual node mode,” the two processors on each node can be used as compute processors, each with 128 megabytes of RAM. The nodes are linked by an extremely fast $76 \times 32 \times 2$ communications grid, which also includes some “service” nodes that do not directly run user programs. We have measured this network’s delivery time for a 256-byte message at about 18 microseconds. The system implements “space sharing”, rather than time sharing; typically, each active job has full control of a subset of the processing nodes.

In this section we describe the solution of some moderately difficult MIP problems from the MIPLIB [3], using between 1 and 128 Janus processors. For our initial experiments, we selected six problems, all solvable on a single processor using the basic branch-and-bound algorithm described above, but still requiring a substantial amount of tree search. Table 6 describes these problems.

Tables 7 through 9 show the solution of these problems using the PICO MIP. The one-processor data are for a single run of the serial layer on a single Janus processor. The data for all other numbers of processors are the average of five runs of the parallel layer.

The parallel runs were configured with a `clusterSize` of 4, so processors were grouped in sets of four, each consisting of one worker-hub and three pure workers. For the two- and three-processor runs, there was just a single cluster of two or three processors, respectively. The `hybridHandler` bounding protocol was used, in combination with the `heapPool` subproblem pool, which implements best-first search. The scattering parameters were set so that an average of 65% of newly-created subproblems were released to the hubs; 25% of the time, subproblem releases were forced to go to the local hub, and the remaining 75% of the time they were sent to a randomly-chosen hub. These parameter settings result in a fairly high level of communication, but Janus’ internal network seems able to support this level.

The \bar{p} columns in the tables represent the number of compute nodes, the “nodes” column gives the total number of subproblems bounded, and the run times are in seconds. The “speedup” column displays the speedup relative to the corresponding one-processor run. The

Problem	General			Rows	Application
	Binary Variables	Integer Variables	Continuous Variables		
bell3a	39	32	52	133	Fiber optic network design
lseu	89			28	(Unknown)
misc07	259		1	212	(Unknown)
mod008	319			6	Machine loading
qiu	48		792	1192	Fiber optic network design
stein45	45			331	(Unknown)

Table 6
Description of the test problems.

“idle” column is the percent of the total processor seconds spent in an idle state, and the “scheduler” column is the percent of total processor seconds devoted to scheduler overhead. Figures 8 through 10 display the same information graphically on a log-log scale. Each “+” data point represents a run of the code, and dashed lines connect the average run times for each processor configuration. The straight dashed line represents an “ideal” situation in which the speedup factor on \bar{p} processors would be exactly \bar{p} .

In many of the problems, the size of the search tree inflates fairly dramatically as one moves from the serial to the parallel version of the algorithm. This inflation occurs because a parallel algorithm cannot follow a strict best-first ordering, and the current implementation lacks an incumbent heuristic. If a high-quality incumbent is unavailable for a significant fraction of the run, a parallel algorithm can spend significant amounts of time investigating noncritical subproblems. Experience from [7,10] suggests that even a crude incumbent heuristic can greatly dampen such tree inflation; we hope to add a (more sophisticated) heuristic to the PICO MIP soon.

After the initial tree inflation phenomenon, speedups generally remain fairly linear, with some “noise” and gentle degradation, until about 48-64 processors, after which they “tail off.” In the near future, we plan to experiment with more difficult problems, larger total numbers of processors, and larger cluster sizes.

One problem, qiu, has dramatic oscillations in node counts (and hence runtime) as the number of processors grows. Though we haven’t completed our investigation, we believe this is due to the sensitivity and numerical properties of the problem. In particular, all the initial pseudocosts for the first integer violations found after solving the root problem are very close. When a processor calculates a number of pseudocosts, it repeatedly modifies the bounds of two variables between calls to the linear-programming solver. This dramatically increases the solver speed because it can use internal state left from the previous solve. However, this means that each variable is “probed” in a slightly different environment depending upon the number of processors, which determines which group of variables is initialized as a group on some processor. Because the pseudocosts are all so close and the problem is difficult, and

hence pushes the LP-solver tolerances, the initial set of global pseudocosts differs slightly for each number of processors. We conjecture this effects initial tree growth and that qiu can have radically different behaviors depending upon these initial branching choices. In fact making slightly-suboptimal branching strategies according to our ranking function can lead to an order of magnitude increase in node count. When we forced a parent-basis reload before each pseudocost initialization, and therefore forced a uniform set of initial costs, the node counts were monotonically slightly increasing with the number of processors. Though this provides more stability for this problem, this is not the default behavior because it dramatically slows down this computation.

Problem	\bar{p}	Nodes	Time	Speedup	Idle	Scheduler
bell13a	1	54,111	152.8	1.0	0.0%	0.0%
bell13a	2	57,745	87.6	1.7	0.5%	1.0%
bell13a	3	57,758	59.6	2.6	0.5%	0.9%
bell13a	4	57,186	44.1	3.5	0.6%	1.0%
bell13a	6	55,218	30.0	5.1	1.5%	1.3%
bell13a	8	54,518	22.1	6.9	1.2%	1.4%
bell13a	12	54,436	15.1	10.1	1.3%	1.3%
bell13a	16	54,214	11.4	13.4	2.1%	0.9%
bell13a	24	53,962	7.8	19.6	2.3%	1.3%
bell13a	32	54,012	6.0	25.5	4.0%	1.7%
bell13a	48	54,045	4.7	32.5	14.0%	2.5%
bell13a	64	54,019	3.3	46.3	8.4%	3.0%
bell13a	96	54,093	2.4	63.7	12.4%	0.0%
bell13a	128	54,155	2.1	72.8	17.1%	0.0%
lseu	1	11,383	24.8	1.0	0.0%	0.0%
lseu	2	18,780	19.9	1.2	0.3%	1.0%
lseu	3	20,828	14.6	1.7	0.7%	1.0%
lseu	4	16,343	8.8	2.8	0.5%	1.1%
lseu	6	17,970	6.8	3.6	1.5%	1.5%
lseu	8	21,152	6.0	4.1	1.0%	1.7%
lseu	12	19,733	3.9	6.4	2.6%	2.6%
lseu	16	20,805	3.1	8.0	3.2%	3.2%
lseu	24	23,210	2.4	10.3	4.2%	1.7%
lseu	32	22,748	1.9	13.1	6.5%	0.0%
lseu	48	23,832	1.6	15.5	9.9%	0.0%
lseu	64	25,189	1.2	20.7	12.1%	0.0%
lseu	96	31,169	1.1	22.5	20.8%	0.0%
lseu	128	28,418	0.9	27.6	28.3%	0.0%

Table 7
Computational results for bell13a and lseu.

Problem	\bar{p}	Nodes	Time	Speedup	Idle	Scheduler
misc07	1	30,449	465.3	1.0	0.0%	0.0%
misc07	2	63,905	511.0	0.9	0.5%	0.3%
misc07	3	68,017	381.3	1.2	0.6%	0.3%
misc07	4	57,830	251.1	1.9	0.8%	0.3%
misc07	6	92,293	270.9	1.7	1.0%	0.4%
misc07	8	81,182	176.3	2.6	0.9%	0.4%
misc07	12	73,939	113.9	4.1	2.2%	0.5%
misc07	16	122,378	142.4	3.3	1.6%	0.4%
misc07	24	87,108	70.4	6.6	5.4%	0.7%
misc07	32	133,126	81.1	5.7	3.2%	0.5%
misc07	48	95,658	42.4	11.0	9.3%	0.9%
misc07	64	97,652	34.4	13.5	12.8%	1.0%
misc07	96	96,452	25.9	18.0	19.9%	1.5%
misc07	128	114,486	23.8	19.6	19.6%	1.3%
mod008	1	18,079	112.9	1.0	0.0%	0.0%
mod008	2	21,792	61.2	1.8	0.2%	0.5%
mod008	3	21,953	40.7	2.8	0.3%	0.5%
mod008	4	22,678	31.5	3.6	0.5%	0.6%
mod008	6	24,881	24.0	4.7	0.8%	0.8%
mod008	8	25,597	18.5	6.1	0.9%	0.5%
mod008	12	26,290	13.0	8.7	1.5%	0.8%
mod008	16	25,395	9.5	11.9	2.1%	1.1%
mod008	24	28,664	7.4	15.3	3.5%	1.4%
mod008	32	24,571	5.0	22.6	5.6%	2.0%
mod008	48	28,182	4.1	27.5	7.8%	1.5%
mod008	64	27,534	3.1	36.4	11.5%	0.6%
mod008	96	29,783	2.5	45.2	16.1%	0.0%
mod008	128	30,289	2.4	47.0	23.7%	0.8%

Table 8
Computational results for misc07 and mod008.

Problem	\bar{p}	Nodes	Time	Speedup	Idle	Scheduler
qiu	1	30,653	4,718.0	1.0	0.0%	0.0%
qiu	2	32,646	2,216.4	2.1	1.6%	0.1%
qiu	3	17,940	854.1	5.5	2.1%	0.1%
qiu	4	41,278	1,557.0	3.0	1.5%	0.1%
qiu	6	19,932	466.6	10.1	3.4%	0.2%
qiu	8	30,536	595.5	7.9	2.7%	0.2%
qiu	12	22,409	278.0	17.0	4.8%	0.3%
qiu	16	32,750	332.7	14.2	6.5%	0.4%
qiu	24	32,356	233.7	20.2	9.9%	0.6%
qiu	32	23,157	145.7	32.4	17.7%	1.0%
qiu	48	28,656	123.9	38.1	21.0%	1.2%
qiu	64	32,649	111.7	42.2	24.7%	1.4%
qiu	96	36,938	92.6	51.0	33.7%	1.9%
qiu	128	52,278	98.6	47.8	35.5%	2.0%
stein45	1	53,461	333.5	1.0	0.0%	0.0%
stein45	2	69,504	212.7	1.6	0.8%	0.6%
stein45	3	71,666	146.3	2.3	0.9%	0.6%
stein45	4	67,462	104.4	3.2	1.1%	0.6%
stein45	6	66,236	70.1	4.8	1.6%	0.8%
stein45	8	68,015	53.8	6.2	1.6%	0.7%
stein45	12	69,770	37.3	8.9	2.5%	0.8%
stein45	16	65,151	27.0	12.4	3.7%	0.7%
stein45	24	65,634	18.9	17.6	5.8%	1.1%
stein45	32	68,001	14.9	22.4	8.7%	1.3%
stein45	48	62,188	10.1	33.0	14.3%	1.2%
stein45	64	68,455	8.5	39.2	17.0%	1.2%
stein45	96	69,903	6.5	51.3	25.0%	1.5%
stein45	128	73,220	5.7	58.5	29.7%	1.7%

Table 9
Computational results for qiu and stein45.

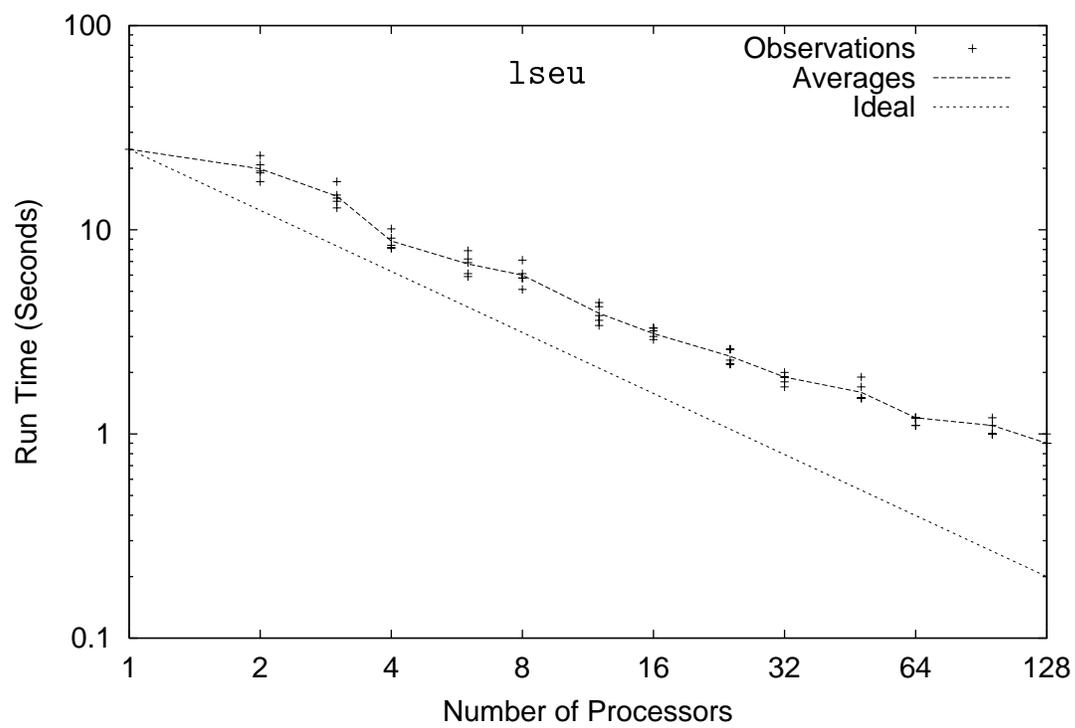
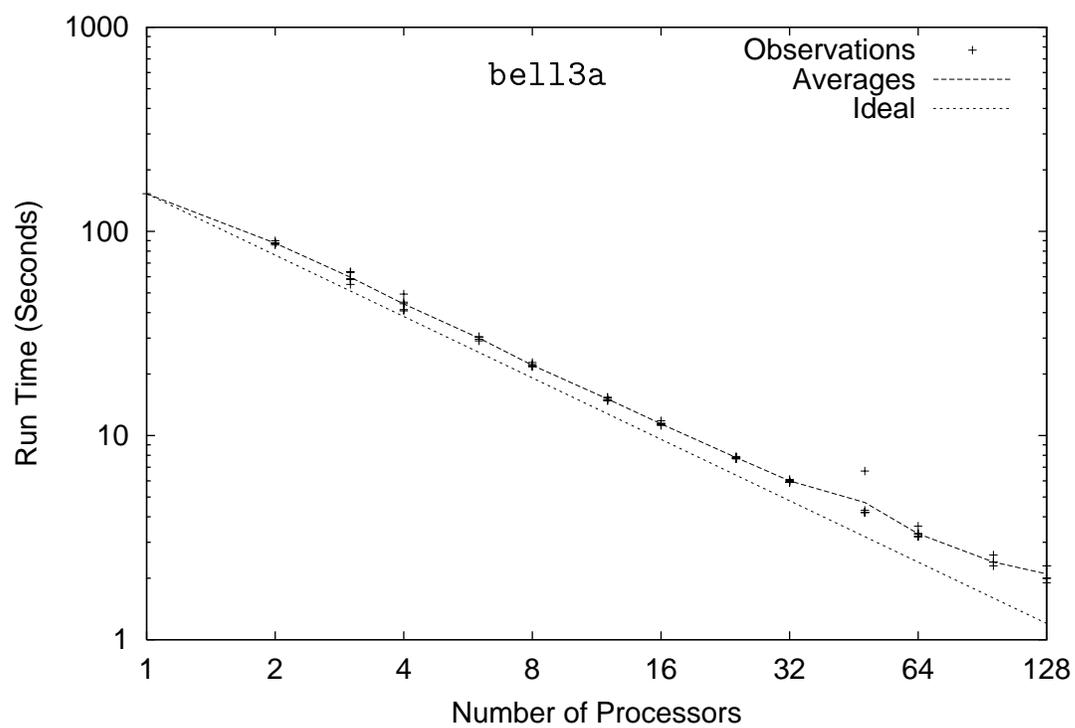


Figure 8. Computational results for bell13a and l1seu.

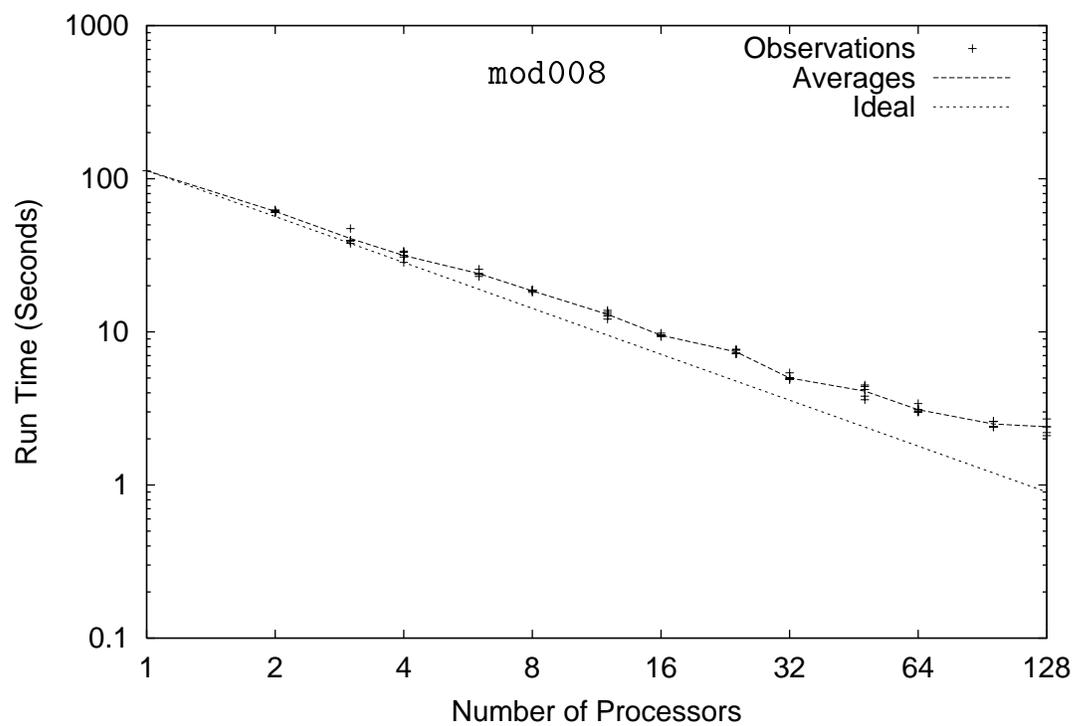
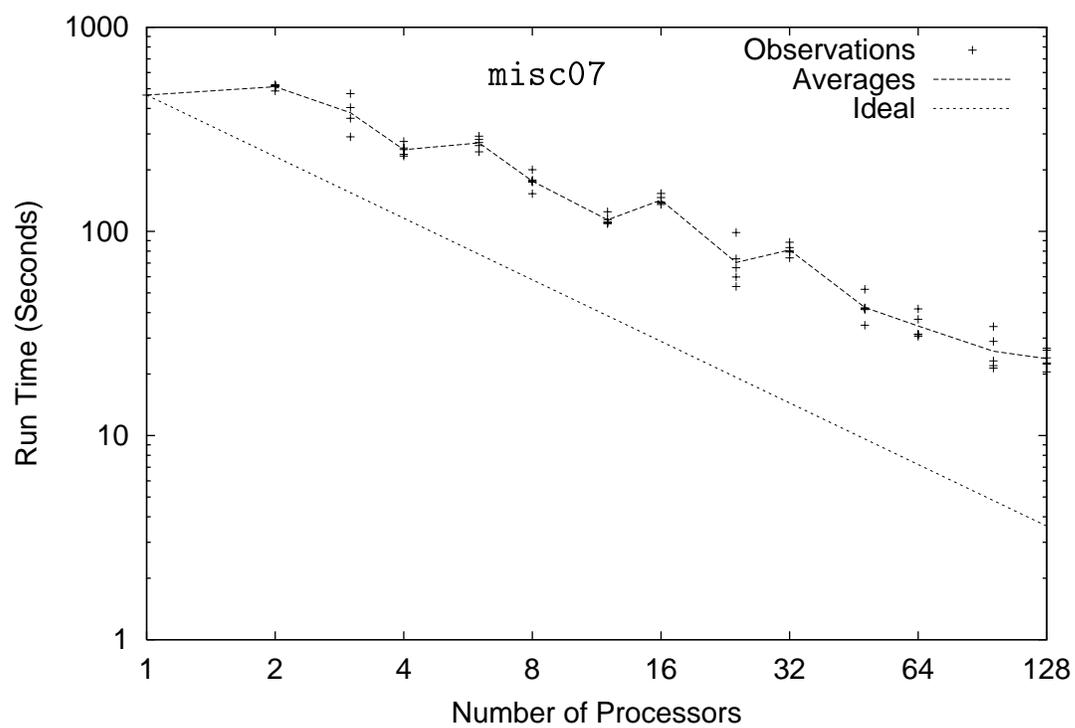


Figure 9. Computational results for misc07 and mod008.

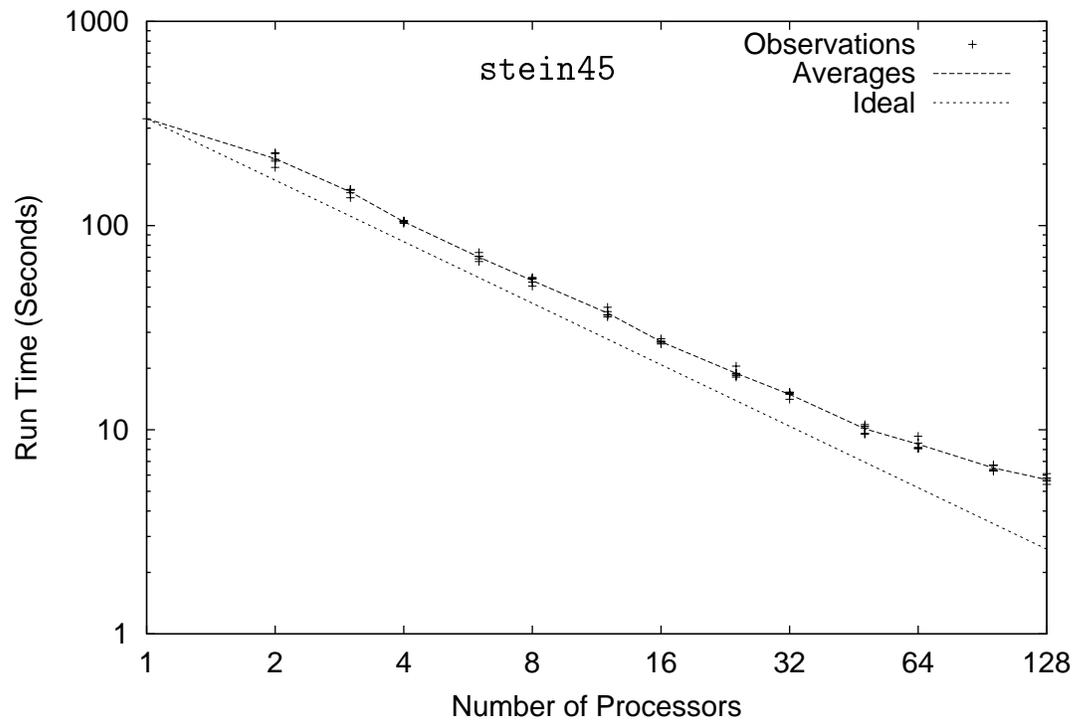
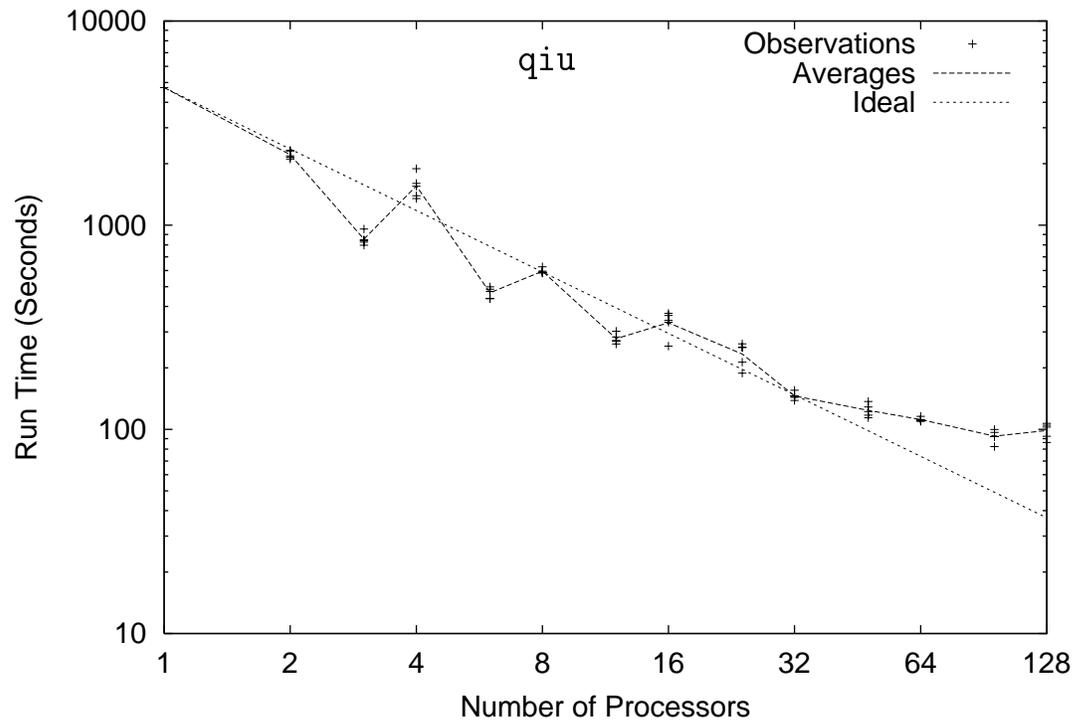


Figure 10. Computational results for qiu and stein45.

6. Resource-constrained project scheduling

This section describes the application of the mixed-integer-programming capability in PICO to a resource-constrained project scheduling problem. We derived problem-specific classes from the general MIP branching and subproblem classes to provide customized incumbent heuristics and output.

Sandia National Laboratories has developed and implemented the Pantex Process Model [21] to support the planning and scheduling activities at Pantex, a US Department of Energy production plant in Amarillo, Texas. The plant simultaneously supports three major DOE programs – nuclear weapon disposal, stockpile evaluation, and stockpile maintenance – which share its facilities, technicians, and equipment. We focused on one piece of the PPM, namely the Evaluation Planning Module (EPM) used to project facility and technician utilization over a given planning horizon (typically a year). Multiple mathematical formulations of this problem are introduced in [17,21,22].

The remainder of this section is organized as follows. Section 6.1 defines the EPM resource-constrained project scheduling problem. Section 6.2 gives the mixed-integer-programming formulation of the EPM problem. Section 6.3 describes the incumbent heuristics we implemented for this problem. Section 6.4 describes I/O and mapping variables from the Pantex MIP formulation to PICO’s linear variable organization. We used the AT&T Mathematical Programming Language (AMPL) to facilitate the input and mapping. Section 6.5 discusses modeling issues.

6.1. Problem description

A substantial portion of the Pantex workload relates to tests of weapons in the active stockpile. Each of these jobs involves partial disassembly of the weapon, one or more tests, and then re-assembly and return of the weapon to the active stockpile. The jobs are generally referred to as *evaluations*, and their planning and scheduling fits a job-shop paradigm.

Each job consists of a *set of tasks*. Some pairs of tasks have precedence constraints, where one job must complete before the other begins. The task precedence relationship forms a *forest*. That is, each task will have at most one “parent” task it must wait for. Of course, by transitivity, it must also wait for any task its parent must wait for. In practice the trees of precedence constraints are highly chainlike.

Tasks are assumed to have a fixed duration for purposes of the EPM planning problem. This duration can be as short as an hour or as long as several months. The scheduling of each task must obey a *time window*. It must start no earlier than its *earliest allowable start time (EAST)*, and must finish by its *latest allowable finish time (LAFT)*. These time-window boundaries are also called *release dates* and *deadlines* respectively in the scheduling literature. The first task in a job (one with no parent) often has an EAST that is tied to the arrival of the weapon. The task for the test itself often has a LAFT motivated by the availability of external resources (e.g., off-site engineers).

The time horizon is broken into time periods. Currently each period is six consecutive working *days*, more generically called *slots*. Each day is decomposed into a fixed number of *units* (currently the units are *hours* and each day is eight hours). Task lengths are given in units. The PPM, and some stochastic global heuristics for the EPM, model the start time of a job as a continuous variable. Humans generally don’t schedule to a finer granularity than the quarter hour, and there is sufficient uncertainty in the data to justify rounding or

shifting to align job starts with the hour (unit).

The evaluation of each task requires a specific facility type (e.g., a Task Bay with 220 electricity) and a qualified crew (e.g., two people holding a single specific certification). Facilities are hierarchically arranged to reflect how one facility can be replaced by another, more general, facility (at a price). However, in the current PPM, and hence our data sets, there is no hierarchy. Each technician has a list of certifications to which (s)he can be assigned. The availability of each technician and the number of facilities of a given type varies by time period in general. However, in our current model we assume all technicians are available at all times and only the facility availability is part of the input. We also assume that technician certifications don't expire.

In a true schedule, each job is assigned to a specific facility and given a specific team of qualified technicians. However, for planning future technician/facility needs, it is currently sufficient to assign tasks to a pool of facilities and technicians. Each technician is assigned to certifications by specifying the amount of time (possibly fractional) that will be devoted to each certification during each time period. No technician is assigned more time units for a particular certification during a time period than the sum of the task lengths (within that period) of tasks requiring that certification. For example, if there is only one task assigned to the time period and it requires 3 technicians for 2 units each, then no technician can have more than 2 units assigned to that certification during that time.

A production plan assigns a start time and facility type to each task. Preemption is not allowed, so a task will occupy that facility for its entire duration beginning at its start time. A production plan is feasible if:

1. All precedence constraints, release dates, and deadlines are obeyed.
2. Each task is assigned to an acceptable facility (type matches or exceeds requirement).
3. The total amount of work scheduled for each facility type during any particular time period does not exceed the availability of such facilities.
4. In each time period the requirements for technicians are matched by technician assignments and the total time assigned to each technician is not greater than an entire time period.
5. For each time period, no technician is assigned to a particular certification for more time units than the sum of the task lengths (within that period) of tasks requiring that certification (in the example above, constraint (4) could be satisfied by a single technician for 6 hours).

Typically, an EPM planning problem spans a year and involves at least 500 jobs and 1000 tasks. Each job has from one to six tasks. About 28 facility types are involved along with 300 technicians, each of whom holds 2–3 of the 80 possible certifications.

In practice, these planning problems are often infeasible. Consequently, the EPM module formulates the EPM planning problem using *ghost facilities* or *facility overage* and *ghost certification hours* or *technician overage* that reflect the number of additional resources that are required by a production plan. Thus the only constraints on a production plan are the time windows and precedence constraints, which are much easier to satisfy than resource

constraints in general. With this formulation, production plans are evaluated by summing the total number of hours of ghost facilities and ghost technicians over all time periods. These two factors (technicians, facilities) are weighted equally.

6.2. The Pantex MIP Formulation

This section gives the mixed-integer-programming formulation of the EPM problem implemented in the PICO system. We first discuss discretization issues and then give the explicit mathematical formulation.

As mentioned in the previous section, the PPM aligns task start times to the hour, or an even smaller granularity. To limit the size of the MIP, “big” tasks with length at least a slot are aligned with slots (days) by starting them at the beginning of a slot. Short tasks (strictly less than a slot long) can start on unit boundaries. This introduces less unforced idle time and makes the MIP output more comparable to output from the PPM. This is implicitly represented in the MIP with the function $b(j, j')$ defined below. Start times are specified as slots, but understood to be mid-slot if a predecessor is running at the start of the slot, and the task can finish before the end of the slot while obeying precedence constraints. That is, a short task will never be split across two slots.

To be more explicit, there are two types of precedence constraints: packed and normal. In a normal precedence constraint, the successor cannot start until the slot after the completion of the predecessor. In a packed precedence constraint, the (short) successor can “pack” into the remaining time in the slot where the predecessor finishes (without going into the next slot). Therefore, long tasks have only normal predecessor constraints with their single immediate ancestor (if it exists).

If a short job has a packing precedence constraint with its immediate ancestor, we may need to add one more (normal) constraint with a nonadjacent predecessor. For example, if there is a chain of many unit-sized jobs, each pair can share a slot, but the entire chain may not fit into a single slot. Specifically, let the precedence chain for short job j be p_1, p_2, \dots, p_d , where p_1 is job j 's immediate ancestor, p_2 is p_1 's ancestor and so on. There is a constraint between job j and job p_d if all the following conditions are met:

1. job j and jobs p_1 through p_{d-1} all fit in a slot,
2. adding job p_d to the group in (1) would overflow a slot, and
3. Jobs p_1 through p_d all fit in a slot.

The first condition implies there are no extra constraints required between job j and predecessor jobs p_1 through p_{d-1} . The second condition implies a normal constraint between job j and p_d . However, if the third constraint is not met, then there will already be a (nonadjacent) constraint earlier in the chain (e.g. between p_1 and p_d) and therefore adding one between j and p_d would be redundant.

We now give the mathematical formulation of the MIP for the EPM planning problem. The formulation uses the following **input parameters, constants, and shorthand**:

q_1	The number of time units in a time slot
q_2	The number of time slots in a time period
p_j	The processing time (in units) of task j
ρ_j	The minimum number of full time slots needed to process task j ($\lceil p_j/q_1 \rceil$). Used to constrain the start time of (long) successor jobs.
r_j	Release time slot of task j (possibly derived from the release dates of predecessors). This is the EAST, earliest available start time.
d_j	Deadline (aligned with a slot) for task j (possibly derived from deadlines of successors). This is the LAFT, last available finish time.
$T(j)$	Set of possible start slots for task j . Shorthand for all time slots from r_j to $d_j - \rho_j$.
$p(j, t, \tau)$	The amount of processing (in units) on task j performed during time period τ if j is started in time slot t .
$T(j, \tau)$	Set of start times t for task j such that $p(j, t, \tau) > 0$.
$K(j)$	The set of possible facility types for task j .
c_j	The technician certification for task j
s_j	Crew size for task j (number of required technicians)
$C(w)$	The set of certifications held by worker w
$f_{k,\tau}$	The number of time units available in facilities of type k during time period τ .
$j_1 \prec j_2$	Task j_1 precedes task j_2 .
$b(j, j')$	1 if $j \prec j'$ and task j' can be “packed” with j (see detailed documentation). Otherwise $b(j, j') = 0$.

The formulation has the following **integer variables**:

$$x_{jtk} = \begin{cases} 1 & \text{if task } j \text{ starts at time slot } t \text{ in a facility of type } k \\ 0 & \text{otherwise} \end{cases}$$

The MIP formulation uses **rational variables** for ghost facilities and ghost technicians:

$$\begin{aligned} y_{wc\tau} &= \text{fraction of time worker } w \text{ spends using certification } c \text{ in time period } \tau \\ F_{k\tau} &= \text{number of ghost facilities of type } k \text{ in time period } \tau \\ G_{c\tau} &= \text{number of ghost technicians with certification } c \text{ in time period } \tau \end{aligned}$$

There are two justifications for using rational rather than integer variables. First, within Pantex, resources are divided among various program managers, each responsible for evaluating a subset of the stockpile. Thus one problem instance could be a the plan for one such manager, which is a subset of the total plan for Pantex. In this case, the fractional piece of a ghost facility could correspond to borrowing a facility part-time for another manager. Even if borrowing part-time is not feasible in some instance, there is sufficient uncertainty in the values we eventually choose for the objective function that using the ceiling of these rational variables will probably still be sufficiently optimal. For technicians, the fractional portion of a person-hour also seems to be in the noise with respect to uncertainty in objective-function weights.

Finally, here is the MIP formulation:

$$\text{minimize } \alpha \sum_{k,\tau} F_{k\tau} + \beta \sum_{c,\tau} G_{c,\tau}$$

subject to

$$\sum_{\substack{t \in T(j) \\ k \in K(j)}} x_{jtk} = 1 \quad \forall j \quad (2)$$

$$\sum_{c \in C(w)} y_{wc\tau} \leq 1 \quad \forall w, \tau \quad (3)$$

$$\sum_{k \in K(j)} x_{jtk} \leq \sum_{\substack{k \in K(j') \\ r_{j'} \leq t' \leq t - \rho_j + b(j', j)}} x_{j't'k} \quad \forall t \in T(j), j' \prec j \quad (4)$$

$$\sum_{j: k \in K(j)} \sum_{t \in T(j, \tau)} p(j, t, \tau) x_{jtk} \leq f_{k, \tau} + q_1 q_2 F_{k\tau} \quad \forall \tau, k \quad (5)$$

$$\sum_{j: c_j = c} \sum_{\substack{t \in T(j, \tau) \\ k \in K(j)}} p(j, t, \tau) s_j x_{jtk} \leq \sum_w q_1 q_2 y_{wc\tau} + q_1 q_2 G_{c\tau} \quad \forall \tau, c \quad (6)$$

$$q_1 q_2 y_{wc\tau} \leq \sum_{j: c_j = c} \sum_{\substack{t \in T(j, \tau) \\ k \in K(j)}} p(j, t, \tau) x_{jtk} \quad \forall c, \tau, w : c \in C(w) \quad (7)$$

(8)

Constraints (2) assure that every task is done. Constraints (3) prevent overscheduling any technician in any time period. Constraints (4) assure a task is not started until all predecessors are completed. Constraints (5) ensure there is sufficient facility capacity in each time period to perform all the work that must be done in that time period, and constraints (6) are the analogous constraints on certification hours within each time period. Constraints (7) prevent some situations where a technician is taking the place of multiple technicians (see the detailed problem statement).

6.3. Pantex Incumbent Heuristics

This section describes the incumbent heuristics we implemented for the EPM planning problem. The heuristic uses the LP-relaxation of the MIP, available at each node of the branch-and-bound tree. It is an α -point schedule. This class of algorithms, first introduced in [30], gives good theoretical and practical performance for schedules with average weighted completion time as the objective function. It has also performed well in practice for the best-effort objective, where one finishes as many jobs as possible by their deadlines[31]. This is closely related to our objective function. In an α -point schedule, one solves an LP-relaxation of and integer-programming (IP) formulation for a scheduling problem. Then one sorts the jobs by the point in time where an α fraction (for $0 < \alpha \leq 1$) of the job is completed. Then one typically schedules the jobs greedily in that order.

We currently set $\alpha = .5$. However, when we have many processors, it would be reasonable for processors to try different values of α . There can be at most n interesting ranges of α , where all values in a given range yield the same ordering.

We need only specify the greedy procedure for scheduling the jobs given the ordering by the α point. First we compute the “resource availability” from the linear programming

solution. That is, we consider the ghost facilities to be “real” and we assume the technician availability dictated by the LP technician assignments (to certifications in each period) plus the ghost technicians. Since the objective-function value of this solution is a lower bound on the optimal integer solution, we are free to use all these resources in our heuristic solution without degrading the objective value. Each job must be placed within its time window and behind all its predecessors. Given this, we place each job in the earliest place that creates no overage with respect to the modified resources. If no such place exists, we place the job either as early as possible, or in the place with minimum overage. We try each of these strategies.

Note that as a subroutine in a branch-and-bound search, some of the jobs may be restricted to an exact start time, or forbidden to start in certain time slots. Those that are fixed are placed before all others. Forbidden slots are considered in determining the legal placements for a job when it’s time to place it.

The LP technician assignments may not be optimal with respect to these job start times. However, we can compute an optimal (rational) technician assignment for these job placements using a network flow algorithm. We compute one network-flow problem for each period.

The flow problem is formulated as follows. There is a source node s , a node W_i for each worker i , a node C_j for each certification j and a sink t . The sink is connected to each of the W_i nodes with capacity equal to the total time the worker is available to work during this period. In our current model, this is the same for all workers: 48 hours. There is an edge from each worker node W_i to each certification node C_j where worker i has certification j . The capacity is the total number of units of work for certification j scheduled in this period (according to our heuristic schedule). Finally, there is an edge from each certification node to the sink with capacity equal to the total man-hours of work for certification j in this time period. That is, for each (piece of a) job with certification j run in this time period, we multiply the length of the job (in this time period) by the crew size. The capacity on the sink-to-worker edges reflects the bound on technician availability. The capacity of the worker-to-certification edges reflects the total time a worker can spend on a certification (constraints 7 above). The capacity of the certification-to-sink edges reflects the total work requirement for each certification. The technician assignment is encoded in the flow from worker nodes to certification nodes. The residual capacity on the edge from certification j to the sink (that is the difference between the capacity and the flow) is the technician overage from this assignment. In particular, if the maximum flow in this network saturates all the certification-to-sink edges, then all the work is done and there is no overage.

We compute these network-flow problems using Andrew Goldberg’s maximum flow code that at one point was available from DIMACS (from the first implementation challenge). We could use the LP solver, but that would require constant exchanges of problems within the solver and significantly increase runtimes.

6.4. Input/Output and Variable Mapping

This section describes data input/output the mapping of variables from the Pantex MIP formulation to PICO’s linear variable organization.

The above incumbent heuristic used variables that had two or three indices. For example, $xjkt$ represented scheduling task j in facility type k and time slot t . However, PICO has a

linear ordering of the variables. Therefore, the Pantex Problem class defines mapping arrays. For example `XMap[j][t][k]` gives the integer variable number corresponding to $x_{j(t-r_j)k'}$. That is, the facility types are stored sparsely, and the legal start times are represented in a compact array. Similarly the class defines the reverse map arrays `xTaskMap`, `xSlotMap`, `xFacilityMap` which give the task, slot, and facility type for a given (linear-ordered) LP variable (and -1 if the variable is not an x variable). We have similar mapping arrays/matrix for the y variables.

We went through a number of steps to get specific instance data into the PICO Pantex class data structures. The PPM placed data into a series of ascii files. We read these files to fill in C++ data structures representing the problem. We created an AMPL (AT&T Mathematical Programming Language) model of the MIP formulation and used a C++ program to create an AMPL data file corresponding the PPM ascii files. We used AMPL to create an mps file (standard integer and linear program specification language) and mappings of the AMPL variables (logically named as in the above mathematical representation) to the rows and columns of the MPS matrix. These files together were sufficient to initialize our branching- and problem-class data structures.

We now describe how we (numerically) convert data from the PPM representation to legal input parameters for the MIP. The PPM measures task length in periods, eg. 2.6667 periods. We convert this to time units and round up (making tasks slightly longer).

In the PPM release dates (EASTs) are points in time, measured in (floating-point) number of time periods. The MIP interprets these as units, though they will almost always be aligned with the start of a slot. Generally, the release date is computed by converting periods to slots and rounding up (moving back in time). However, if a short task can run entirely in its release slot, the release date rounded up to the nearest unit (rather than slot). Because task lengths are rounded up to the nearest unit, there will be cases where a task can fit into its release slot according to PPM data, but not for the MIP. For example, if a task is released with 2.3 units remaining in the slot and is 2.3 units long, it can technically be run in its release slot. However, for the MIP granularity, the task is 3 units long and released with 2 units remaining in the slot. Therefore, its release date will be the start of the following slot. In the worst fcase, a job could be tightly constrained to a time window and the MIP rounding make it appear infeasible when it could fit in the PPM world. We could handle these cases by preprocessing.

Deadlines are a point in time in the PPM, given as a (floating point) number of periods since time 0. The MIP has deadlines aligned with units, rounded down (moved forward in time). This is used with the (unit-based) length of the job to determine the last possible starting time.

6.5. Modeling Issues

The plan modeled in the PPM is easier to solve or approximate than a true schedule, and given the uncertainty in the data (job load, task duration, etc), the extra resolution is probably not warranted. However, the optimal overage for this schedule is a lower bound on the optimal overage for a true schedule where jobs are assigned to specific facilities with specific teams of technicians and all resources constraints are met at the finest resolution (in this case, by the hour). Therefore, a decision to allocate more resources to Pantex can be justified based on the calculations of the PPM, but the resources specified in the plan may not be sufficient. We feel it is important to determine how much this plan underestimates

the resource requirements for a true schedule and how much the PPM overestimates the optimal plan. These comparisons will be made in a future paper. Modeling a true schedule with an hourly task-alignment will increase the size of the MIP by an order of magnitude.

7. Debugging and correctness

In this section we describe the tools we've developed and/or used to insure that PICO is working correctly at all levels. In particular we describe the log analyzer, the quality-assessment suite, the facility for "watching" the fathoming of specific solutions, utilib tools, and commercial tools.

PICO has a runtime option to create for each processor a log file containing information on subproblem creation, bounding, splitting, fathoming, etc. The log analyzer checks the set of files for consistency to make sure, for instance, that each subproblem is explicitly resolved. For example, each problem must be explicitly fathomed or it must be split with each of its children resolved.

The quality-assessment (qa) suite is a set of scripts to check PICO. The scripts run PICO in serial and in parallel on one to six processors for each problem in the test suite with each of a set of test parameter files. The problems are taken from MIPLIB except for a tiny bipartite matching problem that is included because the root problem has an integral LP solution. These test problems are small enough to run through all these test settings in a reasonable time. There is a pure binary integer program, one with general integer variables, one truly mixed problem, one that is integer infeasible, and a couple that tend to have pathological behavior and therefore tend to expose errors in the handling of special cases. The qa suite runs each of these tests using the log analyzer option and checks the objective value, giving a final summary of errors.

In cases where the optimal objective value is known, but PICO is failing to find an optimal solution, PICO has a facility for "watching" optimal solutions. This is particularly useful when, for example, the serial code can return an optimal solution, but the parallel code is incorrect. If the runtime parameter, `sendSolutionToFile` is TRUE, PICO will save the solution (as a vector in a format PICO can read) to the file `PICO-Solution`. If the runtime parameter `checkFathomOnSolution` is TRUE, PICO will read the solution(s) in `PICO-Solution` and exit with an error and a dump of state if it is about to fathom a subproblem containing any of these solutions and the incumbent is strictly worse than these solutions (i.e. they are still candidate solutions). If both of these runtime parameters are set, if PICO finds an optimal solution, it will append it to the `PICO-Solution` file if it is new. This is particularly useful when one is debugging a problem with multiple optimal solutions. One can run PICO multiple times (using the `repeat` parameter), saving all the optimal solutions that are found, and watching all of them for improper fathoming.

The PICO MIP class does a final feasibility check before accepting a new incumbent. It's possible that the solution is considered a candidate because it satisfies the integer tolerance on all integer variables, but the actual solution is slightly off true integers. PICO explicitly sets all the bounds in the LP solver to the rounded values of the variables and solves the problem again. If the problem is (slightly) infeasible, the incumbent is rejected. The `Pantex` class also does a feasibility check using its problem-specific view of the variables. For example, it checks that time windows are obeyed, that ghost variables are correct given the job placements and

resource availability and so on.

Finally, we used William Hart's utilib package of vector and matrix utilities. These provide automatic bounds checking, allowing easy detection of bugs. We have also run PICO through the purify and insure++ codes to check for memory leaks.

8. Conclusion and future development plans

We have just described a flexible, object-oriented approach to implementing parallel branch-and-bound algorithms, including an application to general mixed integer programming and an application of mixed-integer programming to resource-constrained project scheduling. Limited, preliminary computational testing on a small set of moderately difficult MIP's reveals some initial inflation of the search tree, most likely due to the absence of an incumbent heuristic, followed by fairly linear speedups through 32-48 processors.

The innovations of this work include:

- A novel object-oriented approach to describing branch-and-bound algorithms, using transition operators acting on subproblems moving through a state graph.
- The ability to describe both the search order and bounding protocol in a modular way.
- The division of the class library into serial and parallel layers.
- A continuously adjustable degree of communication between the hub and worker processors within a master-slave cluster.
- Use of stride scheduling to manage concurrent tasks within each processor executing the parallel branch-and-bound method.

In future, we plan to carefully investigate the performance of PICO on various processor configurations, refining its work distribution algorithm, so that the PICO core can be configured to operate efficiently on harder problems and larger processor configurations than described here. We also plan to refine the MIP application by including a parallel incumbent heuristic, as well as adding some other modern features including node-level preprocessing and cutting planes. The flexible underpinnings provided by the PICO core should make these enhancements relatively easy. Forthcoming papers will also describe some more specific applications of PICO, including a full description of computational results for the Pantex EPM scheduling problem with comparisons to other heuristics for this problem.

REFERENCES

1. R. Anbil, F. Barahona, L. Ladányi, R. Rushmeier, and J. Snowdon, *IBM Makes Advances in Airline Optimization*, Research Report RC21465(96887), IBM T. J. Watson Research Center, Yorktown Heights, NY, 1999.
2. E. M. L. Beale, Branch and bound methods for mathematical programming systems, in P. L. Hammer, E. L. Johnson, and B. H. Korte, eds., *Discrete Optimization II*, *Annals of Discrete Mathematics* **5** (North-Holland, Amsterdam, 1979) 201-219.

3. R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh, An Updated Mixed Integer Programming Library: MIPLIB 3.0, Technical Report 98-3, Department of Computational and Applied Mathematics, Rice University, 1998.
4. M. Benachouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol, Building a parallel branch and bound library, in: *Solving combinatorial optimization problems in parallel, Lecture Notes in Computer Science* **1054** (Springer, Berlin, 1996) 201-231.
5. J. Clausen, Parallel search-based methods in optimization, in: J. Wasniewski, J. Dongarra, K. Madsen, and D. Olesen, eds., *Applied Parallel Computing: Industrial-Strength Computation and Optimization: Third International Workshop, PARA '96 Proceedings* (Springer, Berlin, 1996) 176-185.
6. J. Clausen and M. Perregaard, On the best search strategy in parallel branch-and-bound: best-first search versus lazy depth-first search, *Annals of Operations Research* **90** (1999) 1-17.
7. J. Eckstein, Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5, *SIAM Journal on Optimization* **4** (1994) 794-814.
8. J. Eckstein, Control strategies for parallel mixed integer branch and bound, *Proceedings of Supercomputing '94* (IEEE Computer Society Press, Los Alamitos, CA, 1994) 41-48.
9. J. Eckstein, How much communication does parallel branch and bound need?, *INFORMS Journal on Computing* **9** (1997) 15-29.
10. J. Eckstein, Distributed versus centralized storage and control for parallel branch and bound: Mixed integer programming on the CM-5, *Computational Optimization and Applications* **7** (1997) 199-220.
11. J. Eckstein, W. E. Hart, and C. A. Phillips, Resource Management in a Parallel Mixed Integer Programming Package, *Proceedings of the Intel Supercomputer Users Group Conference*, Albuquerque, NM, (1997), <http://www.cs.sandia.gov/ISUG97/papers/Eckstein.ps>.
12. R. Fourer, D. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Boyd & Fraser Publishing Company, 1993.
13. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing* (MIT Press, Cambridge, MA, 1994).
14. B. Gendron and T. G. Crainic, Parallel branch-and-bound algorithms: Survey and synthesis, *Operations Research* **42** (1994) 1042-1066.
15. W. D. Hillis, *The Connection Machine*, (MIT Press, Cambridge, MA, 1985).
16. C. S. Horstmann, *Mastering C++: an introduction to C++ and object-oriented programming for C and Pascal programmers* (John Wiley, New York, 1996).
17. D. A. Jones, C. R. Lawton, G. F. List, M. A. Turnquist, The Pantex model: Formulations of the evaluation planning module, *Sandia National Laboratories Technical Report*, SAND99-2095, 1998.
18. M. Jünger and S. Thienel, Introduction to ABACUS — a branch-and-cut system, *Operations Research Letters* **22** (1998) 83-95.
19. G. Karypis and V. Kumar, Unstructured tree search on SIMD parallel computers: A survey of results, *Proceedings of Supercomputing '92*, (IEEE Computer Society Press, Los Alamitos, CA, 1992) 452-462.

20. M. Kim, H. Lee, and J. Lee, A proportional-share scheduler for multimedia applications, *Proceedings of the IEEE International Conference on Multimedia Computing and Systems '97* (IEEE Computer Society Press, Los Alamitos, CA, 1997) 484-491.
21. E. A. Kjeldgaard, D. A. Jones, G. F. List, M. A. Turnquist, Planning and scheduling for agile manufacturers: the Pantex Processing Model, *Sandia National Laboratories Technical Report*, SAND98=0030, 1998.
22. E. A. Kjeldgaard, D. A. Jones, G. F. List, M. A. Turnquist, J. W. Angelo, R. D. Hopson, and Hudson, Swords into plowshares: nuclear weapon dismantlement, evaluation, and amaintenance at Pantex, *Interfaces*, 1999.
23. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, (Benjamin/Cummings, Redwood City, CA, 1994).
24. J. T. Linderoth, *Topics in Parallel Integer Optimization*, Ph.D. thesis, Department of Industrial and Systems Engineering, Georgia Institute of Technology, 1998.
25. R. Lüling and B. Monien, Load balancing for distributed branch and bound algorithms, *Proceedings of the International Parallel Processing Symposium* (IEEE Computer Society Press, Los Alamitos, CA, 1992) 543-549.
26. A. Mahanti and C. J. Daniel, A SIMD approach to parallel heuristic search, *Artificial Intelligence* **60** (1993) 243-282.
27. F. Mattern, Algorithms for distributed termination detection, *Distributed Computing* **2** (1987) 161-175.
28. T. G. Mattson and G. Henry, An overview of the Intel TFLOPS supercomputer, *Intel Technical Journal* Q1 1998, http://developer.intel.com/technology/itj/q11998/articles/art_1.htm, 1998.
29. G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi, MINTO, a mixed integer optimizer, *Operations Research Letters* **15** (1994) 47-58.
30. C. A. Phillips, C. Stein, and J. Wein, Minimizing average completion time in the presence of release dates, *Mathematical Programming B*, Vol. 82, Nos. 1-2, June 1998, pp. 199-224.
31. C. A. Phillips, R. N. Uma, and J. Wein, Off-line admission control for general scheduling problems, *Proceedings of the Eleventh Annual ACM/SIAM Symposium on Discrete Algorithms*, January 2000, pp. 879-888, to appear in *Journal of Scheduling*.
32. T. K. Ralphs and L. Ladányi, *SYMPHONY User's Manual: Preliminary Draft*, <http://branchandcut.org/SYMPHONY/man/man.html>, 2000.
33. V. J. Rayward-Smith, S. A. Rush, and G. P. McKeown, Efficiency considerations in the implementation of parallel branch and bound, *Annals of Operations Research* **43** (1993) 123-145.
34. M. W. P. Savelsbergh, Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing* **6** (1994) 445-454.
35. Y. Shinano, M. Higaki, and R. Hirabayashi, Generalized utility for parallel branch and bound algorithms, *Proceedings of the 1995 Seventh IEEE Symposium on Parallel and Distributed Processing* (IEEE Computer Society Press, Los Alamitos, CA, 1995) 392-401.
36. Y. Shinano, K. Harada, and R. Hirabayashi, Control schemes in a generalized utility for parallel branch and bound, *Proceedings of the 1997 Eleventh International Parallel Processing Symposium* (IEEE Computer Society Press, Los Alamitos, CA, 1997) 621-627.
37. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The*

- complete reference* (MIT Press, Cambridge, MA, 1996).
38. B. Stroustrup, *The C++ programming language* (Addison-Wesley, Reading, MA, 1991).
 39. S. Tschöke and T. Polzer, *Portable Parallel Branch-and-Bound Library PPBB-Lib User Manual, Library Version 2.0*, Department of Computer Science, University of Paderborn, <http://www.uni-paderborn.de/~ppbb-lib>, 1996.
 40. C. A. Waldspurger and W. E. Wehl, Stride scheduling: Deterministic proportional-share resource management, Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science (Cambridge, MA, 1995).

UNLIMITED RELEASE
INITIAL DISTRIBUTION:

- 1 MS 0321 W. CAMP, 9200
- 1 1110 W. E. Hart, 9211
- 1 1110 D. E. Womble, 9222
- 1 1110 C. A. Phillips, 9211
- 1 1110 V. J. Leung, 9211
- 1 9217 J. C. Meza, 8908
- 1 0451 E. A. Kjeldgaard, 6515
- 1 0451 D. A. Jones, 6515
- 1 0451 C. R. Lawton, 6515
- 1 0451 M. A. Turnquist, 6515
- 1 0188 D. L. Chavez, 1030

- 1 MS 9018 Central Technical files, 8945-1
- 2 0899 Technical Library, 9616
- 1 0612 Review and Approval Desk, 9612
For DOE/OSTI