Taxonomizing Optimization Algorithms for Metacomputing

Ian Foster
Jean-Pierre Goux
Adriana Iamnitchi
Jeff Linderoth
Stephen Wright
Golbon Zakeri

April 5, 1999

1 Introduction

The goal of this research effort is to investigate the suitability of optimization problem classes, algorithms, and implementation techniques for metacomputing environments. In this paper, we characterize several optimization algorithms from the perspective of metacomputing platforms. For the purposes of this study, we assume that metacomputing platforms have the following five key properties:

I: Dynamic availability. The quantity of resources available to us over time may vary, as may the amount of computation delivered by a single resource.

II: Unreliability. Resources may disappear without notice.

III: Poor communications properties. Latencies between any given pair of processors may be high, variable, and unpredictable; while bandwidth is correspondingly low, variable, and unpredictable. Connectivity (as measured, for example, by bisection bandwidth) may be particularly low.

IV: Heterogeneity. Resources may have varying physical characteristics (for example, the amount of memory and speed will vary between different processors on the metacomputer).

V: Scale. The number of resources available to us can potentially be much larger than in a conventional parallel computer.

For our purposes, the term "resource" will typically mean a processor (including CPU, memory, and secondary storage). In more general metacomputing applications, the term "resources" can also include specialized devices such as large tape drives, sophisticated visualization devices and the computers that drive them, and so on.

In the next section, we briefly review the optimization problem classes and algorithm types that we believe to be most suitable for metacomputing implementation. In Section 3, we list the issues involved in implementing these algorithms on a metacomputer, classifying the issues according to their relevance to parallel computing or to their specific relevance to metacomputing. In Section 4, we evaluate each of the algorithm types from Section 2 according to the issues described in Section 3. We hope that our comments will illuminate the specific areas in which further work is needed, both in the development of algorithms and in the design of metacomputing infrastructure tools.

2 Optimization Problems and Algorithms

2.1 Global Optimization

COMMENT — Golbon, please insert multistart stuff.

2.2 Stochastic Programming Algorithms

Stochastic programming problems arise when decisions affecting the future have to be made at the current time, in the presence of uncertainty in the problem data. (The decisions correspond of course to variables in the problem formulation.) For instance, the outcome of investment decisions depend in an obvious way on fluctuations in stock prices, interest rates, and so on—data which is not available at the time at which the decision is made. In general, the uncertainty can appear anywhere in the objective or constraints. (For problems with linear constraints, uncertainty appears most commonly in the right-hand side.)

For compuational purposes, the uncertainty is typically discretized into a finite number of scenarios, to each of which is assigned a probability. The most complex stochastic programs are staged over a number of time periods, where decisions must be made in each period. span over a number of time periods. In these multistage stochastic programs, the dependendence between scenarios at different stages can be plotted as a tree. See Figure 1 for an example of a three-stage problem. Each node of the tree depicted in Figure 1 contains a mathematical program (MP), often a linear program, in which the variables correspond to the decisions to be made at this stage, given that the scenario whose data appears in the node has come to pass. In general, at stage k, data at the stages $1, 2, \ldots, k-1$ is known, while data at future stages remains uncertain. The tree structure of Figure 1 is reflected in the presence of *nonanticipativity constraints* in the formulation of the overall stochastic program.

Algorithms for stochastic programs approach the problem in a number of different ways. One possibility is to formulate the entire problem as a large MP, and then solve it with standard algorithms for linear (or nonlinear) programming. The problem constraints have much structure, which can be exploited in performing the linear algebra operations for the algorithm, possibly even parallelized.

Another possibility is to use algorithms that *decompose* the large problem into smaller individual problems. These methods use "master" problems to coordinate the information from the individual subproblems. This approach is naturally parallel (in that the subproblems can be solved independently). The master problems are more difficult to parallelize and may become bottlenecks for the parallel implementation if naive formulations are used. However, we have control over the decomposition strategy. At one extreme, we can use a strategy in which each task is the solution of an MP that corresponds to a single node in the tree. The number of variables at each node can range from 1 to 1000s.(Of course, this task must be performed repeatedly, so that the solutions from the different nodes can be coordinated appropriately.) To increase the granularity, we can define tasks that consist of groups of nodes. These groups could be subtrees rooted at a particular node, or they could consist of a group of nodes within the same stage.

A third class of algorithms makes use of sampling techniques. These methods are important when the tree is simply too large to allow full solution of the problem. Instead, we use sampling to choose a subset of scenarios, and solve a more tractable stochastic program based on the sample data to produce an approximate solution to the underlying problem. More powerful computational resources may permit us to use a larger sample and thereby obtain a more accurate approximation to the true solution of the full problem.

Stochastic programs scale by increasing the time horizon (number of stages) and the number of possible outcomes at each stage (that is, by refining the discretization). A large SP would span, say, 30 periods with time periods where each stage has 5 possible outcomes. These specifications would lead to a scenario tree with approximately 5^{30} nodes.

Section 4 provides a more detailed description of the metacomputing related proper-

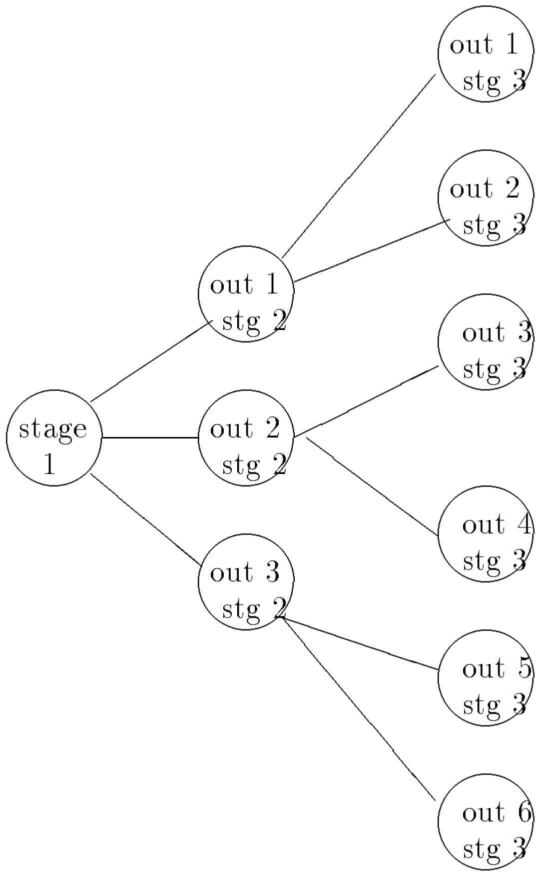

Figure 1: stochastic tree for 3 time periods and 6 scenarios

ties of these three classes of algorithms for solving SPs.

2.3 Stochastic Programming: Verifying Solutions

COMMENT — Golbon, please insert Morton stuff, just describe the approach in words, mention the scale of the problem.

2.4 Integer Optimization

An important and computationally intensive class of optimization problems arises from applications in which some of the decision variables are required to take on integer values. An important subclass is the mixed integer nonlinear programming problem (MINLP), which has the following form (see [?] [?]):

$$(MINLP) \qquad z_P \equiv \min\{f(x,y)\ :\ g(x,y) \leq 0,\ \ x \in Z_+^n, y \in R_+^p\}, \tag{1}$$

where $f : R^n \times R^p \mapsto R$ and $g : R^n \times R^p \mapsto R^m$. If f and g are linear functions of their arguments, the problem is known as a mixed integer programming problem (MIP) (the "linear" qualifier is understood); that is,

$$(MIP) \qquad z_P \equiv \min\{c^T x + d^T y\ :\ Ax + By \leq g,\ \ x \in Z_+^n, y \in R_+^p\}, \tag{2}$$

MIP is of great practical importance, so a great deal of study has been devoted specifically to this problem [?].

Possibly the most widespread and effective methodology for solving MINLP and MIP goes by the name of *branch-and-bound*. In a branch-and-bound procedure, we form relaxed versions of the problem (1) in which some of the integer variables are fixed at certain integer values, while the others are allowed to vary as continuous, real-valued variables. Since the number of different ways we can fix some variables and relax others is exponential in n (the number of integer variables), it is clear that we cannot examine all possible relaxations for any but the smallest cases. Branch-and-bound is a strategy for performing a systematic search of the set of possible relaxations, using procedures that avoid unpromising relaxations, thereby avoiding a large part of the potential search space. It keeps track of the best feasible solution found to date. If it finds that the solution of a particular relaxation is not feasible, but decides that some variant on this relaxation could lead to a feasible solution that beats the best one found so far, it breaks up its feasible set into separate regions and formulates a relaxed version for each subproblem so generated. This is the "branching" part of the procedure.

The branch-and-bound procedure can be understood by means of a *tree*, in which the nodes correspond to different relaxations of the problem. The root node is the relaxed problem in which all integrality restrictions are ignored, while a node may have child nodes, generated by the branching procedure just described.

To give more details on the branch-and-bound approach, we introduce some notation. Let z_U denote an upper bound on the optimal objective value z_P; usually z_U is the lowest objective value obtained so far among all feasible points (x, y) that have been encountered in the search process. For a given node N in the branch-and-bound tree, we use $\mathcal{F}(N)$ to denote the feasible set for the subproblem at node N. This will be a subset of the feasible set for (1) and will include the integrality requirements. We use $\mathcal{R}(N)$ to denote the *relaxed* feasible region for node N, which is specified by the same set of constraints as $\mathcal{F}(N)$ except that the integrality requirements are discarded. We let z_{RP}^N be the optimal objective value calculated by solving the relaxation at node N (that is, minimizing the objective over $\mathcal{R}(N)$) and z_P^N be the objective values of the non-relaxed problem (i.e. the minimum over $\mathcal{F}(N)$). Since $\mathcal{F}(N) \subset \mathcal{R}(N)$, we certainly have that $z_P^N \geq z_{RP}^N$. We let z_L^N be a lower bound on z_P^N that is known prior to solving the node-N problem. (For instance, z_L^N could be the optimal objective value obtained from the relaxed subproblem at the parent node of N. We would have $z_L^N \leq z_{RP}^N$ in this case because the feasible region for the child problem is a subset of the feasible region for the parent.) If the feasible region for the relaxation at node N is empty, we set $z_{RP}^N = z_L^N = \infty$.

The *active set* $\mathcal{L}$ is a list of relaxed problems that have been generated by the branching procedure but not yet solved. We use (x^*, y^*) to denote the current guess of the solution of (1).

Using all this notation, we specify the branch-and-bound procedure for solving (1) in Algorithm 1.

The sole difference between a simple branch and bound algorithm for MINLP and for MIP occurs at the **Evaluate** stage. For MINLP, the relaxed problem to be solved is a nonlinear program, while for MIP the relaxed problem is a linear program. The procedures used to implement the **Select**, **Evaluate**, and **Divide** steps are all critical; a wide range of possibilities is available in each instance. The **Divide** step is commonly handled by simple branching, in which we select a component x_i^N of the solution of the relaxed problem at node N that is not integral, and define the child problems N^i as having the same constraints as N except that x_i is restricted to certain integer values. For instance, we we have $x_i^N = 2.34$, we could define problems N^1, N^2, N^3, N^4 by adding the constraint $x_i \leq 1$ to N^1, $x_1 = 2$ to N^2, $x_1 = 3$ to N^3, and $x_1 \geq 4$ to N^4. In general, we prefer branching rules for which the relaxed solution (x^N, y^N) is excluded from the union set $\cup_{i=1}^k \mathcal{R}(N^i)$.

Even with the best heuristics, however, the branch-and-bound procedure just out-

Algorithm 1 The Branch and Bound Algorithm

0. **Initialize.**
   $\mathcal{L} \leftarrow P$, with integrality constraints ignored; $z_U \leftarrow \infty$;
1. **Terminate?**
   Is $\mathcal{L} = \emptyset$? If (x^*, y^*) has been assigned by this point, it is optimal.
2. **Select.**
   Choose problem $N \in \mathcal{L}$ and delete it from $\mathcal{L}$.
3. **Evaluate.**
   Solve the relaxation of N (that is, minimize $f(x, y)$ over $\mathcal{R}(N)$) to obtain z^N_{RP}. If the problem is infeasible ($z^N_{RP} = \infty$), go to step 1, else let (x^N, y^N) be the solution obtained at this node.
4. **Prune.**
   If $z^N_{RP} \geq z_U$, go to step 1. If $x^N \notin Z^n_+$, go to step 5, else set $z_U \leftarrow z^N_{RP} (= z^N_P)$, $(x^*, y^*) \leftarrow (x^N, y^N)$, and delete from $\mathcal{L}$ all relaxed problems M for which $z^M_L \geq z_U$. Go to step 1.
5. **Divide.**
   Form k new problems $N^1, N^2, \ldots, N^k$ by dividing the feasible region $\mathcal{F}(N)$ into k parts, such that $\cup_{i=1}^k \mathcal{F}(N^i) = \mathcal{F}(N)$. Assign a value of $z^{N^i}_L$ for each $i = 1, 2, \ldots, k$, for example, $z^{N^i}_L = z^N_{RP}$. Add the problems $N^1, N^2, \ldots, N^k$ to the active set $\mathcal{L}$. Go to step 1.

lined may not work on many practical problems, simply because the number of relaxed problems to be solved is too large. Sophisticated algorithms for MINLP and MIP include a number of enhancements to widen the range of problems that can be solved.

Warm starts, pseudocosts, and cutting planes are enhancements found in many advanced branch and bound codes for solving MIP; see for example [?], [?]. We discuss these techniques briefly now.

Warm-start procedures exploit the fact that the relaxed problem at each child node is closely related to the relaxed problem at its parent to construct a good initial guess for the child. For MIP, the dual solution at the parent can be used to construct a starting point for the dual of the child that is feasible and, in many cases, nearly optimal. Often, just a few steps of the dual simplex method applied to the child node are needed to identify a solution. Chvátal [?] gives more details. In the nonlinear case, we can use the relationship between the dual problems if appropriate, or otherwise modify the primal solution of the parent to produce a (generally infeasible) starting point for the primal problem at the child node.

Pseudocosts are a technique used during the braching stage to decide the component on which integer branching should be performed. With each integer variable x_j, we associate two quantities P_j^- and P_j^+ that are estimates of the per-unit increase in objective function value that would result from fixing x_j to its next lowest integer value $\lfloor x_j \rfloor$ and its next highest integer value $\lceil x_j \rceil$, respectively. One way to determine values for P_j^- and P_j^+ is to observe the true increase in objective function value whenever integer braching on variable x_j is performed *anywhere in the branch-and-bound tree.* An alternative method is to *tentatively* divide subproblem at node N into some of its possible children, and perform some computations at the tentative child nodes to obtain an estimate of the relaxed solution at each. This method has the advantage that pseudocost information can be calculated when necessary, but computational effort must be expended in computing this information. The concept of pseudocosts was introduced by Bénichou *et al.* [?]. Linderoth and Savelsbergh [?] discuss a number of specific pseudocost implementation issues, including how to initialize and update pseudocost values.

Cutting planes are (linear) inequalities added to the description of P that exclude portions of the feasible region $\mathcal{R}(0)$ of the relaxed problem without cutting away any portion of the feasible region $\mathcal{F}(0)$ of P. The addition of cutting planes can increase the value of z_{RP}^N for each node N in the tree, which in turn allows more nodes to be fathomed (that is, removed from the active set $\mathcal{L}$) at the **Prune** stage of the branch-and-bound algorithm. The idea of using cutting plance to solve integer programming problems dates back to the work of Gomory [?]. The computational effectiveness of incorporating cutting planes in a branch-and-bound procedure was demonstrated by Crower, Padberg, and Johnson [?] and by Padberg and Rinaldi [?]. Like pseudocosts, valid inequalities generated at one node of the branch and bound tree are often useful in cutting off fractional solutions at other nodes of the tree. Thus, it may be useful to share cutting planes between nodes, though the benefit of doing so must be balanced against the costs. The cost of generating different classes of inequalities varies widely. For example, the inequalities of Gomory [?] require only a few arithmetic operations to generate; the inequalities used by Crowder, Padberg, and Johnson [?] require solving a knapsack problem; whereas the inequalities described by Balas, Ceria, and Cornejouls [?] are more expensive, since they can require solution of a linear program of roughly twice the size of the original linear program at a node.

COMMENT — JP - Talk about features of sophisticated MINLP algorithms. Maybe not now, nost of it has been mentioned. Perhaps talk about cutting planes – any work besides Mehotra and Stubbs?

3 Taxonomy

In this section we list common design issues for parallel optimization algorithms. The issues can be divided into two main categories: those that are present for all parallel algorithms and those for which metacomputing poses unique challenges and opportunities. By describing these parallel algorithm characteristics, we hope to shed light on the resource, architecture, and communication demands that optimization algorithms place on a metacomputing infrastructure. We hasten to add that our taxonomy is not complete, and that there is overlap in the categories presented here.

3.1 General Parallel Computing Issues

Certain features of parallel algorithms are present on all types of parallel computing platforms, but take on distinctive properties when implemented on metacomputing platforms.

3.1.1 Control Paradigm

The control paradigm refers loosely to the manner in which the algorithm is parallelized. We consider three main categories:

- Batch computation or task-farming;
- Master-Worker;

- Distributed control.

Task-farming usually refers to parallel applications in which the problem can be broken up into many independent tasks that do not need to communicate with each other. These tasks are farmed out to the available processors. Each task may communicate its results to a central processor, which collates and processes the data.

In the master-worker paradigm, as in task-farming, we have a central "master" processor and a number of workers, who receive tasks from the master and communicate their results to the master in return. The difference with task-farming is that the application is more dynamic. The master may create many new tasks while the application is running, usually in response to results that have been generated by earlier tasks. Moreover, since the data that defines any two tasks may be quite similar, we may aim to improve the efficiency of the application by avoiding retransmission of common data between a master and a given worker processor. The algorithm should be designed so that the computational work and data handling that must be performed by the master does not become a bottleneck. Slight generalizations of this paradigm can be obtained by splitting the work of the master between several physical processors.

In the distributed-control paradigm, control of the algorithm does not lie with one or a few master processors, but rather is distributed over most or all of the active processors. In other words, there is no distinction between master and worker processors. Each processor may make decisions about the algorithm, new resource requests, distribution of its work to other processors, either alone or in consultation with neighboring processors. The complexity of implementing fault tolerant distributed control algorithms is quite large, since if control information is lost, the algorithm may fail. Hence, duplication of control information is necessary.

Generally, we will focus on algorithms for which a task-farming or a master-worker control paradigm is appropriate.

(Is this still true? What about Anda's work?)

COMMENT — Others: please check this stuff and correct/expand where necessary

3.1.2 Communication

The issue of communication between processors is closely related to the control paradigm, since communication between processors may be required in order to control the parallel algorithm. Important aspects of this issue include the following.

- Do we share between processors information that is *not* related to rthe correct execution of the algorithm? For example, in MIP we have the option of sharing warm start information, pseudocost information, and cutting plane information.

- The approximate ratio of communication time to computation time. Of course, this ratio depends strongly on the problem instances and the types of information to be communicated.

Due to the high latencies (characteristic III) present in a metacomputing environment, implementations of algorithms for which there are low communication requirement are desirable in a metacomputing context.

3.1.3 Synchronicity

In some parallel algorithms, the execution of certain subtasks must be synchronized in some way. Implementation of such algorithms are challenging to implement in a metacomputing environment, in large part due to characteristics II and III: unreliability and poor communications properties.

3.1.4 Load Balancing

For purposes of efficiency, the computational work in a parallel application should be approximate balanced between the available processors (taking into account their differing capabilities). Some algorithms are naturally load-balanced, while some intelligence is required to achieve this goal in other instances. Of course, the load balancing strategy is a function of the control paradigm.

3.2 Metacomputing Issues

In this section, we describe a few characteristics of optimization algorithms for which implementation on metacomputing platforms poses unique challenges.

3.2.1 State Persistence

The efficiency of some algorithms may be enhanced if some part of the state of each task persists after the task has completed execution. A later tasks assigned to the same processor may be able to exploit this information to reduce its runtime considerably. Two examples:

- In many algorithms, a great deal of overhead (and data) is needed to initialize a single task, whereas relatively little additional data is needed to needed for subsequent tasks. If the state of the original task persists (or at least the part of the state that contains the relevant shared data), it is necessary to communicate only the incremental data in order to start later tasks on this processor, thereby reducing the communication overhead.
- As a series of tasks are executed on a processor, information may be accumulated that makes it possible to solve later tasks in the series more efficiently. Examples include warm-start information or pseudocosts in MIP. State persistence allows us to use this information locally without incurring the overhead of including it in the task definition.

None of the algorithms we consider actually depends on state persistence for correct execution, since all information required to perform a subtask of the parallel optimization algorithm could be passed upon initialization of the subtask.

3.2.2 Fault Tolerance

Fault tolerance is the ability of the algorithm to cope with a loss of resources during execution. If resources are lost, must all the tasks that were assigned to those resources be rescheduled? If it is permissible to avoid rescheduling of some of the tasks, can the effect of this loss of information on the eventual solution be quantified? The latter concept is related to the issue of *approximability* described in Section 3.2.7.

3.2.3 Preemptivity

A parallel algorithm is preemptive if it can benefit when the amount of resources available to it increases during run time. Algorithms that use task-farming control are generally preemptive if the number of tasks significantly exceeds the number of available processors. Master-worker algorithms are also preemptive if the number of tasks in the task pool at any one time can be maintained at a high enough level to keep all the processors busy.

3.2.4 Grain Size

The *grain size* specifies the computational size of the "unit of work" into which the parallel algorithm can be broken. In some algorithms, there may be different ways of partitioning the work, each of which gives rise to a diferent grain size. In many cases, individual units of work can be aggregated in a single task, where the total data required to specify the task is not too much greater that that required for a single unit or work.

If information about the communications and computational properties of the metacomputing platform are available at run time, it may benefit the algorithm to determine the size of the task dynamically. Algorithms that are able to adjust the sizes of their tasks are in general more suitable for metcomputers.

3.2.5 Heterogeneity

Some algorithms have a special need for a heterogeneous computational environment (in which, for example, the processors have different speeds and different amounts of storage) and can benefit from it. Algorithms with a dynamic grain size, for example, can take advantage of differences in machine speed to obtain better load balancing.

COMMENT — Amdahl's Law type stuff here?

3.2.6 Scalability

Since metacomputing environments offer more resources in which to solve a problem, a logical question to ask is whether or not this increase in resources will result in an increase in the problem size that one is able to solve. This basic concept was formalized by Kumar and Rao [?], who introduced the concept of the *isoefficiency function.* For a given efficiency $E \in (0,1]$, the isoefficiency function $f_E(p)$ is equal to the problem size N needed to attain an efficiency of E on p processors. If $f_E(p)$ were an exponential function, the algorithm would be poorly scalable, since the problem size N would have to increase rapidly to maintain the same efficiency as new processors are added. It would be difficult to obtain good speedups on large numbers of processors unless the problem size is enormous. If $f_E(p)$ is more nearly linear, the problem size need only increase at a rate proportional to the number of processors to maintain a constant efficiency.

COMMENT — The CS people can comment more here if necessary

3.2.7 Approximability

In some optimization problems and algorithms, there is a tradeoff between the accuracy or confidence level of the computed answer and the amount of computational resources. A better answer can be obtained when more resources are used, and the relationship can be quantified to some extent. This issue is important in metacomputing, where the resource availability may be highly variable. We would in general prefer an algorithm that returns a lower quality solution when resources are scarce to one that cannot execute at all in this situation.

4 Specific Optimization Algorithm Characteristics

COMMENT — In this section, I envision "numbers" that the CS people can use as a guildline for their infrastrucure design

In the previous two sections we discussed optimization algorithms and a number of attributes of algorithms viewed from the perspective of designing effective metacomputing implementations. In this section, we give qualitative and quantative details of the algorithms from which a metacomputing implementation may be derived.

4.1 Global Optimization

COMMENT — Golbon? This one is a task-farming application, as far as I understand

4.2 Stochastic Optimization

4.2.1 Decomposition Approaches

- **State Persistancy :** Persistence of state is required to store the description of the work and a copy of the executable. It is desirable to have this as the work description does not change very much from one iteration to the next.

- **Communication :** (BD) Typically, at each forward pass, the master sends workers an array of doubles (length depending on problem size; typically 100s to 1000s). This is the altered data that the worker will commence working on. Upon ending the work, each worker will send back a cut which is again an array of doubles of similar size, typically. (LM) Here the communication is even less than the above. The amount of communication is small compared to the amount of the solution time.

- **Synchronicity :** The algorithms are synchronous.

- **Grain Size :** An LP or a scenario (a whole collection of LPs).
- **Fault Tolerance :** If a worker goes down and does not return the finished task, that task will have to be rescheduled for another worker. These algorithms do not cope with lost information and hence are not fault tolerant in that sense.
- **Heterogeneity :** Not heterogeneous, the workers have similar tasks to handle and even the master is rather similar.
- **Paradigm :** Master/Worker with fixed work cycle.
- **Load Balancing :** Naturally load-balanced, but potentially improvable.
- **Preemptivity :** Conventional BD and LM are not preemptive in the sense that the task pool size is fixed. If new workers become available and there are still tasks to be performed however they can be taken advantage of.

4.3 Decomposition Approaches with Sampling

Here state persistence and communication is very similar to the previous case.

- **Synchronicity :** The algorithms are asynchronous.
- **Load Balancing :** Naturally load-balanced, but potentially improvable.
- **Grain Size :** An LP or a scenario (a whole collection of LPs).
- **Fault Tolerance :** Unlike the previous set of algorithms, we maintain complete fault tolerance here in the sense that it is possible to progress with results that a limited number of workers have produced. It is not necessary for every worker to complete their given task in order to proceed.
- **Preemptivity :** These algorithms are fully preemptive. For any new available worker, a new task can be spawned.
- **Heterogeneity :** Not heterogeneous, the workers have similar tasks to handle and even the master is rather similar.
- **Paradigm :** Master/Worker with variable work cycle.

4.4 Bounding and Verification Methods with Sampling

- **State Persistancy :** If the number of resources is large enough, each worker will have to complete only one task hence this won't be of too much importance.
- **Communication :** Minimal, master sends work and results may have to be collected (or they can just be written to files).
- **Synchronicity :** The algorithms are asynchronous.
- **Load Balancing :** Naturally load-balanced.
- **Grain Size :** Large grain sizes, for example a whole SLP.
- **Fault Tolerance :** Fully fault tolerant as information can be drawn from variable sample sizes.
- **Preemptivity :** These algorithms are preemptive since we can always use a larger sample size and work can usually be generated on the spot.
- **Heterogeneity :** Not heterogeneous.
- **Paradigm :** Task farm.

4.5 Branch and Bound

4.5.1 Mixed-Integer Nonlinear Programming

The parallelization strategy in the nonlinear branch-and-bound code developed by Leyffer is the same strategy as Jonathan Eckstein used for MILP branch-and-bound. The difference will be that we would solve NLP instead of LP problems. Eckstein mentioned different approaches in his papers. As far as metacomputers are concerned, it seems reasonable to reduce the amount of data stored on each processors. Therefore we will use a single pool approach (centralized) and we will even not use the features of the quasidistributed pool approach (where state information of the previous solve are kept on the local memory of the worker). The master is assumed to run on a dedicated machine.

- **State Persistancy :** We have a single pool and no quasidistributed informations stored, persistence of state is only required to store the description of the original relaxed NLP and a copy of the binary. If this information is lost, it is not expensive to regenerate it.

- **Communication :** The workers have already the description of the NLP (except for his first task), the master needs only to pass him the warm-start, bound and branching information (stored on the master level). The amount of communication is small compared to the amount of the solution time.
- **Synchronicity :** The algorithm is asynchronous.
- **Load Balancing :** Naturally load-balanced, but potentially improvable.
- **Grain Size :** One NLP.
- **Fault Tolerance :** The problem instances are not erased from the stack during the solution phase, therefore if a worker happens to disappear, the information is not lost and another worker can handle it.
- **Preemptivity :** The algorithm is preemptive. If new workers are available, they can handle problems remaining on the stack.
- **Heterogeneity :** The algorithm is not heterogeneous, the workers have similar tasks to handle. The master should preferably run on a machine with a lot of memory.
- **Paradigm :** Master/Worker.

4.5.2 Mixed-Integer Linear Programming

Control Paradigm
The branch and bound algorithm lends itself naturally to a master-worker control paradigm. A number of authors have considered branch and bound algorithms with decentralized control [?] [?], but as mentioned in Section 3.1.1, in a metacomputing environment we choose to focus on algorithms that use the master-worker paradigm.

Communication
The communicated data for a branch and bound algorithm using a master-worker control paradigm can be broken into four categories: initialization information, task information, result information, and "optional" information.

Initialization information includes the worker executable program and the linear programming relaxation of the root node of the branch and bound tree. The representation of the linear programming instance may be of size ranging from less than a kilobyte to tens of megabytes. Bixby, Ceria, McZeal, and Savelsbergh [?] introduce a library of mixed integer programming instances MIPLIB arising from real world applications and give statistics about the problem instances.

COMMENT — Should I repeat their table for the instances we consider?

Task information consists of incremental information needed to construct the relaxation at a node and (optionally) the fractional solution from which to warm start the subtask optimization procedure. The incremental information is generally a short vector of bounds that are imposed on a subset of the integer variables. The length of the vector is linearly related to the depth of the subtask in the search tree. For practical (solvable) problems the depth is usually not much greater than 50.

In Table 1, we attempt to quantify the usefulness of the warm start information by showing the average time required to reoptimize a node and the time required to solve the initial linear programming relaxation using the commercial MIP software XPRESS-MP [?]. For this demonstration, we have chosen the instances of MIPLIB that require a significant amount of computation time to solve the initial linear programming relaxation. From Table 1 we see that given warm start information, a linear program can be optimized quite a bit more quickly than without warm start information.

Table 1: Initial Optimization and Reoptimization Times for MIPLIB Instances

Name	#Rows	#Columns	Initial LP Time (sec.)	Avg. Reoptimization Time (sec.)
air03	125	10757	4	0.42
air04	824	8904	454	7.62
air05	427	7195	37	2.23
arki001	1049	1388	1	0.035
danoint	665	521	2	0.83
dano3mip	3203	13873	628	91.3
fast0507	508	63009	262	47.7
mitre	2055	10724	3	0.045
nw04	37	87482	23	1.34
seymour	4945	1372	50	1.52
swath	885	6805	3	0.16

Result information is the solution of the linear programming relaxation at a node, which is a vector of doubles of length equal to the number of variables in the problem.

Optional pieces of information that may be communicated include pseudocosts and cutting planes. On sharing pseudocost information, the computational studies in Linderoth and Savelsbergh [?] indicate that either the explicit initialization of pseudocosts or the sharing of pseudocosts in the early phases of the branch and bound procedure can be very important. Also, the importance of initializing and sharing pseudocost information dimishes as the algorithm progresses. Pseudocost information may be generated at every node of the branch and bound tree and is a very small amount of information.

Cutting plane information is different than pseudocost information in that it may not be generated at every node of the branch and bound tree. Inequalities from the classes used by Crowder, Padberg, and Johnson [?] and by Van Roy and Wolsey [?] can only sometimes be found that are useful. An inequality of the type descibed by Gomory [?] or Balas, Ceria, and Cornejouls [?] can theoretically *always* be found that cuts off a fractional solution. However, for reasons of efficiency and numerical stability, it is not always best to use the generated inequalities.

For a number of instances, Table 2 shows the percentage of time a cutting plane was generated at a node, as well as the average time necessary to generate a cutting plane. The statistics were generated using the code PARINO described in the thesis by Linderoth [?]. The cutting planes generated were simple variations of those described by Crowder, Padberg, and Johnson [?] and by Van Roy and Wolsey [?].

Table 2: Cut Generation Statistics for MIPLIB Instances

Name	#Nodes	Success Rate (%)	Avg. Generation Time (msec.)
blend2	3315	39.51	7.43
cap6000	5690	72.81	2190.5
gesa2_o	98465	2.39	20.00
gesa3_o	2144	11.54	20.00
l152lav	1790	19.16	12.29
mitre	672	100.00	8.71
mod011	2882	11.20	322.57
modglob	285	63.81	5.57
p2756	896	75.30	22.00
pp08a	463834	3.79	3.00
set1ch	261215	1.00	11.00
vpm2	94473	6.56	2.00

Synchronicity

Almost no synchronization is required. The nodes may be evaluated in any order. Also, any valid bound information, cut information, and pseudocost information can be used to control the algorithm, so there is no need to wait for the most recent information.

Load Balancing

COMMENT — Hmmmmm. Don't know what to say here really. Ideas?

State Persistance

A number of issues relating to the usefulness of state persistance for branch and boun algorithms for MIP were discussed in the **Communication** section. In particular, state persistance might be useful both because a MIP subtask can be specified incrementally and because information such as warm-starts, pseudocosts, and cutting planes could be stor4ed and used locally on a processor.

Fault Tolerance

Theoretically, if a feasible solution exists, then all nodes of the branch and bound tree must be explored in order to ensure that it is found. For practical purposes, it may be that a branch and bound algorithm is able to cope with the loss of a few of its processors (and hence nodes of the branch and bound tree).

COMMENT — I am running the following experiment. For a number of instances, I am randomly "throwing away" nodes of the branch and bound tree. Specifically, for each $\alpha \in \{0.5, 1.0, 5.0, 10.0, 25.0\}$ and depth $d \in \{0, 5, 10, 50\}$ a node is thrown away with probability α if its depth is greater than d. This will take quite a bit of time to complete, especially since many trials should be performed

.

For a master-worker control paradigm, a fault-tolerant branch and bound algorithm can be implemented by

- ensuring that the master process resides on a reliable/dedicated machine,
- only deleting a subproblem from the work queue at the master upon confirmation that the subproblem has been optimized.

Preemptivity
As seen in Table 2, some instances of MIP require many tens of thousands of nodes of computation. For these instances, an implementation of the branch and bound algorithm should easily be able to make use of resources that become available.

Grain Size
The grain size could easily be as small as solving one node. However, for many MIP instances, the time required to solve one node is very small. Table 1 only showed average node solution times for the larger instances in MIPLIB. Table 3 shows the average node solution time for smaller instances. From this data, it seems that allowing the grain of computation to be many nodes of the branch and bound tree should be useful. This is especially true for metacomputing implementations, since in these enviornments the time to communicate the result of a subtask optimization could certainly dwarf the computation time. Also, if a master-worker paradigm is used, then increasing the grain size will also reduce contention effects at the master.

Table 3: Average Node Solution Times for MIPLIB Instances

Name	#Rows	#Columns	Avg. Node Solution Time (msec.)
cap6000	2177	6000	64.1
dcmulti	291	548	3.64
fiber	364	1298	1.19
fixnet6	479	878	5.62
flugpl	19	18	0.347
markshare	8	67	5.36
mas76	13	151	5.57
misc07	213	260	11.6
mod010	147	2655	39.3
modglob	292	422	6.27
p0548	177	548	6.84
set1ch	493	712	6.67
stein45	332	45	5.73

Heterogeneity

In the master-worker control paradigm, the master process and the worker processes perform different tasks, so ... The deviation of computation times among nodes

Scalability
MINLP and MIP are NP-Complete, meaning that all known algorithms (including branch and bound) require in the worst case an amount of work exponential in the problem size in order to find a solution. Therefore, for a reasonable implementation, a linear increase in the number of processors requires only a logarithmic increase in the problem size in order to maintain the same efficiency. At first glance, the sublinearity of the isoefficiency function seems to be a good thing, since it implies that branch and bound should scale well. However, it also means that increasing the number of processors by a small (or linear) amount will not lead to a similar growth in the size of the problem that we are able to solve.

Approximability

Unless $\mathcal{P} = \mathcal{NP}$, then it is impossible to approximate the solution of a general MIP instance in polynomial time with any constant factor [**?**]. For certain classes of MIP problems, such approximability is possible, see Arora [**?**] for a survey.

COMMENT — Perhaps the following experiment: Run a variety of (hard) problems for a long time, outputting the closeness to optimality after a specified time interval

5 Conclusions

COMMENT — Something obviously needs to go here. Also, the place where the table best fits into the paper needs to be determined and MILP needs to be entered into the table

SP Alg.	*state Persistancy*	*comm*	*synch*	*load balancing*	*gran.*	*fault tolerance*	*preemptivity*	*heterogeneity*	*paradigm*
standard decomposition approaches as they stand	best if we have	vertical, small amnt.	synch.	naturally load balanced	LP or scenario	no	no	no	MW w/ fixed WC size
decomposition approaches with sampling	best if we have	vertical, small amnt.	asynch.	naturally load balanced	LP or scenario	yes	yes	no	MW w/ variable WC size
bounding and verification with sampling	no concern	no communication	asynch.	naturally load balanced	large (e.g. SP)	yes	yes	no	task farming
nonlinear B&B	yes (static)	restart, branching, bounds	asynch.	naturally load balanced	large (e.g. NLP)	yes	yes	no	MW

Table 4: An Approximate Taxonomy of Optimization Algorithms