A High Performance Decomposition Solver for Portfolio Management Problems in the AURORA Financial Management System*

Erwin Laure

Institute for Software Science
University of Vienna
erwin@par.univie.ac.at

Hans Moritsch

Department of Business
University of Vienna
moritsch@finance2.bwl.univie.ac.at

Abstract

Financial planning problems are formulated as large scale, stochastic, multi-period, tree structured optimization problems. An efficient technique for solving this kind of problems is the nested Benders decomposition method. In this paper we apply this technique to the problem of portfolio optimization and present a parallel, portable, asynchronous implementation. To achieve our portability goals we elected the programming language Java for our implementation and used a high level Java based framework, called *OpusJava*, for expressing the parallelism potential as well as synchronization constraints. Our implementation is embedded within a modular decision support tool for portfolio and asset liability management, the *Aurora Financial Management System*.

1 Introduction

Finding optimal investment strategies is important for long term investors, especially those who wish to achieve goals and meet future obligations. This task is called asset and liability management (ALM). Application fields of optimal asset allocation taking into account future liabilities include pension plans, insurance companies, investment conglomerates, banks, university endowments, etc. Systems to support the investor require high-level modeling and optimization tools that are able to deal with multistage decisions and realistic models, including highly complex stochastic multidimensional scenario processes.

The Aurora Financial Management System under development at the University of Vienna is such a decision support system for portfolio and asset liability management. In order to find an optimal investment plan, it solves a large scale, stochastic, multi-period, tree structured optimization problem. Realistic models taking into account many risk factors and transaction costs result in a computational complexity which requires high performance computing, in particular since the system response time is a critical factor in financial planning.

In this paper, we particularly deal with the issue of portfolio management which is a classical problem in the field of financial optimization [25]. The objective of a portfolio optimization problem is generally to maximize an investors return, cash flow, or utility subject to constraints such as the budget available or investor's preferences. It yields a decision on how to spend a budget on different assets. If asset returns are not known precisely, as it is the case in practice, a decision problem under uncertainty must be solved which can be formulated as a *stochastic programming model*. In addition, we assume that the portfolio is restructured several times within a planning horizon, i.e. decisions are taken in stages, which results in a *dynamic multistage stochastic program*. The uncertainty is modeled by means of a tree structured representation of future scenarios for asset prices (e.g. stock or bond prices), along with their probabilities.

*A short version of this paper appeared in: *Commercial Applications for High-Performance Computing, ITCom 2001*, Denver, CO, 2001.

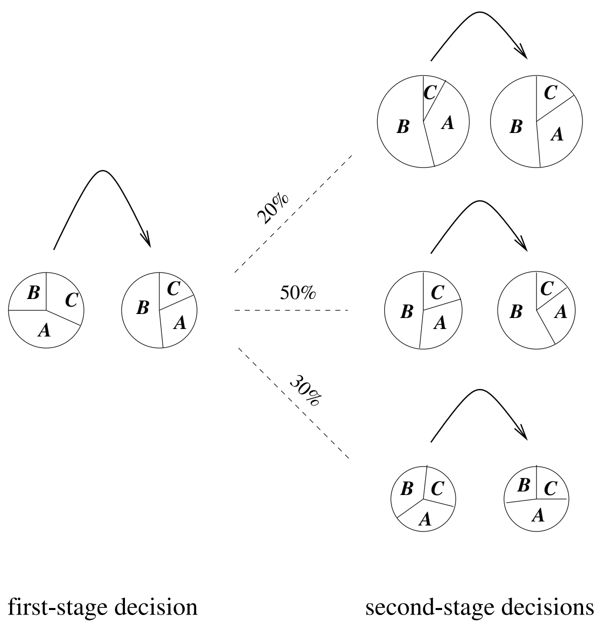

Figure 1: Two-stage Portfolio Optimization Problem

The investor has to plan all his present and future activities conditional to the realized scenario. Figure 1 illustrates a two-stage problem with a portfolio of three assets (A, B, and C). Based on the assumed future development (modeled by three scenarios for increasing, decreasing, and constant asset prices) the investor has to take a rebalancing decision now which takes future portfolio rearrangements into account. The objective is the maximization of the expected value of the portfolio at the end of the planning horizon. The circle diameters in the Figure correspond to the assumed portfolio values.

In this paper we present an asynchronous parallel solution technique for this kind of optimization problem which is based on the nested Benders decomposition method. With this technique, the large optimization problem is represented as a set of small problems defined over the nodes of a scenario tree. By iteratively solving and—through data exchange with neighbor nodes—modifying their local problems, all nodes contribute to the overall solution.

Since the local problem can be solved independently from other nodes, we are able to leverage task parallelism. In our task parallel implementation, we specifically took care of portability issues. Parallel applications in finance often run on clusters and on networks of workstations or PCs rather than on classical supercomputers. Therefore, we implemented the optimization module in the Java language, more specifically in OpusJava, a portable Java based framework for distributed and parallel computing.

The remainder of this paper is organized as follows: after an overview of the Aurora Financial Management System in Section 2 we discuss in detail the stochastic optimization module, focusing on the asynchronous nested Benders decomposition algorithm, in Section 3. Furthermore, we derive some requirements a parallel implementation has to fulfill. The OpusJava framework which is employed for our implementation is presented in Section 4. We present details of the parallel implementation in Section 5 and report on initial experiments conducted on a workstation cluster and a Beowulf system. We conclude the paper with a summarizing discussion and an outlook on future work.

2 The Aurora Financial Management System

The *Aurora Financial Management System* under development at the University of Vienna is a modular decision support system for portfolio and asset liability management (ALM). One of the features of the system is the integration of asset and liability management within a unified optimization process. Parts of the system are being developed in cooperation with an Austrian pension fund company [7, 30, 31].

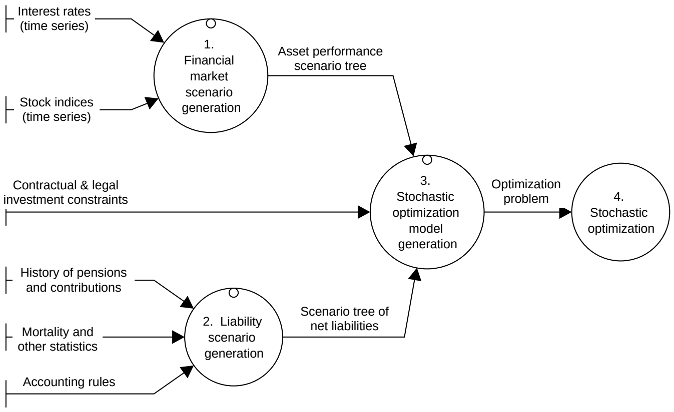

Figure 2: The Aurora Financial Management System.

The problem of dynamic asset-liability management seeks an investment strategy in which an investor chooses a portfolio of various assets or asset classes, in such a way that some risk-adjusted objective (see [30]) is maximized, subject to the uncertainty of future markets' development, future obligations, and additional constraints, related to the investor's business.

The problem requires modeling of all relevant risks (both on the asset side and the liability side and a precise statement of a coherent objective function (in the sense of [28]). The Aurora ALM model is developed within a framework of stochastic dynamic optimization methods and models. Thus the uncertainty is modeled by a number of random processes which are eventually transformed to ones discrete in time and values. Multistage stochastic optimization is by now a widely used technique within the field of portfolio planning, asset and liability management, risk management and asset allocation problems. Numerous models of these types have already been proposed [3, 8, 17, 27, 35].

The Aurora ALM model is based on a pricing model for financial instruments on the asset side and a repayment model for liabilities. The core of the system is a large scale linear or convex optimization problem, solution of which is the ultimate aim of the modeling effort.

Figure 2 depicts the structural design together with the simplified data flow in the system. The *stochastic optimization model generator* (3) creates an optimization problem derived from *financial market scenarios* (1) and *liability scenarios* (2). The latter two modules are completely independent of each other. The financial market scenarios are generated using historical records of the financial market performance (in form of stock indices, interest rates, prices of individual stocks and other financial instruments). The liability scenarios have to take into account the structure of the liabilities; for instance, the scenario for a pension fund must adhere to the accounting rules specific to the pension fund. These rules are typically based on the fund's historical records, as well as common statistical data such as mortality tables. In either case, the uncertain future is modeled by a tree of all possible futures.

As a result of joining these two models of independent aspects of the uncertain future, a multistage optimization problem is formed (3) and solved (4). After a solution has been presented to the decision maker, he may wish to adjust some of the parameters of the scenario generators, or the financial targets. In either case, some or all of the above computation will have to be repeated to produce a new set of results. In the following, we concentrate on the stochastic optimization module and refer the interested reader to [23] for a detailed discussion of the modeling modules and their parallel implementation using High Performance Fortran (HPF) [13] and to [30] for a detailed description of the complete optimization problem.

3 The Stochastic Optimization Module

The task of the stochastic optimization module is to solve a multistage decision problem based on a combination of financial market and liability scenarios as described above. Specifically, the goal of the optimization

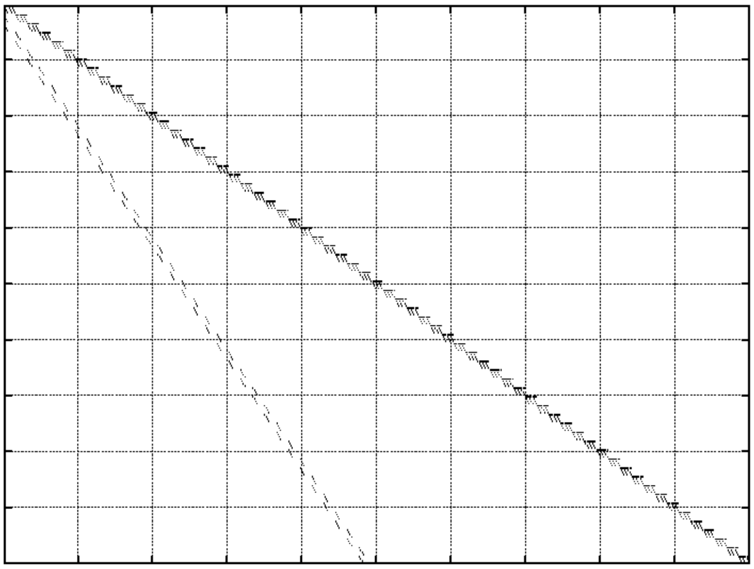

Figure 3: Constraint Matrix of a 63 Nodes LP Formulation

is to derive an optimal investment strategy for a given planning horizon which includes the possibility of dynamic re-balancing the portfolio.

Problems in which decisions are made in stages can be described by means of tree structured models. Usually, a tree defines all the possible developments of the environment in which the decisions are taken. Every node of the tree corresponds to a specific future scenario. Based on this scenario a node specific objective function is formulated. The optimization of a combination of all node specific functions is the ultimate goal:

$$\left\| \begin{array}{ll} & \text{Minimize} \sum_{n \in \mathcal{N}} f_n(x_n) \\ \forall (n \in \mathcal{N}) & \left\{ \begin{array}{l} T_n x_{pre(n)} + A_n x_n = b_n \\ x_n \in S_n, \end{array} \right. \end{array} \right. \tag{1}$$

where $\mathcal{N}$ denotes the set of nodes in the tree. A node $n \in \mathcal{N}$ is associated with a local objective function f_n wrt. decision variables x_n. A_n, b_n, and T_n describe the constraints representing the dependency on the variables of predecessor node $pre(n)$ (i.e. decision taken at the previous stage), whereas S_n describes constraints local to node n. In the case of portfolio optimization the ultimate goal is the maximization of the expected value of the portfolio expressed as the sum of the (probability-weighted) portfolio values at individual nodes. Note, that the maximization problem is converted to a minimization problem.

3.1 Nested Benders Decomposition

One possibility of solving this problem is the formulation of the whole problem (1) as a linear program which could be solved e.g. by means of the simplex method. However, this would result in a very sparse constraint matrix that is difficult to deal with. Figure 3 depicts the constraint matrix of the LP formulation of a 63 nodes problem, where every dot in the figure corresponds to a non-zero matrix element.

Decomposition methods, such as nested Benders decomposition [2, 10, 15, 18], allow to decompose the whole problem into a set of small problems that correspond to the objective functions and constraints associated with each node of the scenario tree. Thus, in a two-stage problem, the second stage decomposes into the sum of local functions $f_j, j \in succ(n)$ of all successor nodes $succ(n)$. The two stage problem can be represented as:

$$
\left\|
\begin{array}{l}
\text{Minimize } f_{n_0}(x_{n_0}) + \sum_{j \in succ(n_0)} f_j(x_j) \\
\text{s.t.} \\
T_j x_{n_0} + A_j x_j = b_j \\
x_{n_0} \in S_{n_0} \\
x_j \geq 0 \qquad \text{for all } j
\end{array}
\right.
\tag{2}
$$

where n_0 is the root node.

Using the decomposition method, the problem over the full tree can be solved in an iterative manner. Every node n makes use of *optimal value functions* $Q_{n,j}(x_n) = \min\{f_j(x_j) : A_j x_j = b_j - T_j x_n; x_j \geq 0\}$. They express the dependence of the non-local parts of the global objective (more specifically, those parts associated with all successors j) on the value of the local variable x_n.

In each iteration step s, node n uses approximations of these functions, denoted by $Q^{(s)}_{n,j}(x_n)$, which are represented as a set of linear constraints called *optimality cuts*. At the beginning of an iteration cycle, a node receives from its predecessor node (the *master*) the solution $x_{pre(n)}$ together with the actual approximation value of Q. It solves

$$
\left\|
\begin{array}{l}
\text{Minimize } f_n(x_n) + \sum_{j \in succ(n)} Q^{(s)}_{n,j}(x_n) \\
\text{s.t.} \\
T_n x_{pre(n)} + A_n x_n = b_n \\
x_n \in S_n \\
x_n \geq 0 \\
x_n \in R^{(s)}_n
\end{array}
\right.
\tag{3}
$$

where $R^{(s)}_n$ are feasibility constraints (see below). In the case of linear objective functions, the problem can be solved by means of a Linear Program (LP) solver based on the simplex method.

If the LP is feasible, the node sends the solution x_n and the approximation values $Q_{n,j}(x_n)$ to its successors (the *slaves*). In addition, it compares its newly computed objective value with the master's approximation. If the local objective value matches the master's approximation the master is notified via sending a signal called *null cut*. Otherwise, an additional constraint to the master's problem (an optimality cut) is computed and sent to the master. This constraint will improve the master's approximation in the next iteration.

If the LP is infeasible, a new constraint to the master's LP called *feasibility cut* is computed and sent to the master. This constraint ensures that the local LP in the next iteration, which is based upon the master's solution taking this feasibility cut into account, has a feasible solution.

If a node receives null cuts from each of its slaves, i.e. its current approximation is correct, it enters a *sleeping* state from where it can resume only by receiving a new solution from its master or new cuts from its slaves. Once the root node of the tree enters the *sleeping* state and all the other nodes of the tree are also sleeping, the optimal solution of the whole problem has been found and the algorithm terminates.

For a more detailed discussion of nested Benders decomposition see for instance [2, 10, 15, 18].

3.2 Synchronous vs. Asynchronous Algorithm

In the synchronous version of the method, the master and the slaves are tightly coupled: each node (acting as slave) receives the solution from its master, solves its current local problem and (acting as master) sends its current solution to its slaves. Subsequently it waits for receiving a complete new set of cuts from them. This coupling results in forward and backward sweeps over the whole tree. Due to the computational complexity of the algorithm, parallel decomposition methods [1, 6, 29] have been developed. However, only neighboring nodes may be simultaneously active in the tightly synchronized version.

In the asynchronous version (see also [26]), the local problem is iteratively solved using only the best information obtainable so far. This means that a node does not have to wait for all slaves to send their

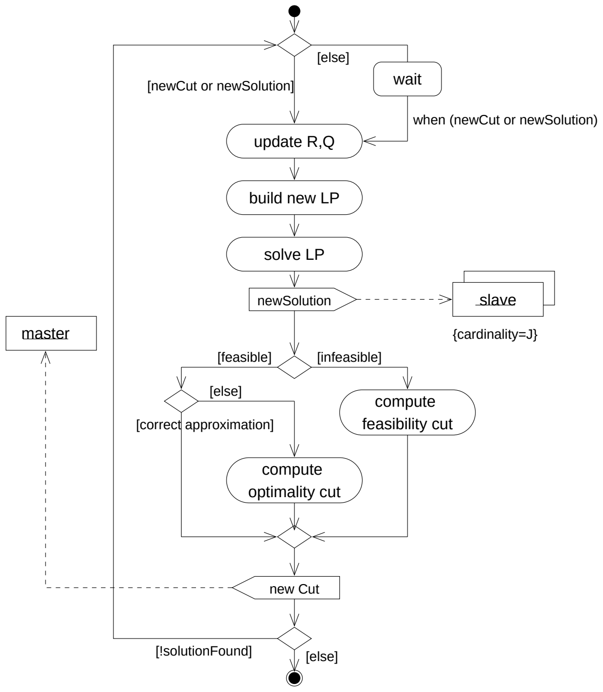

Figure 4: Node Algorithm for Asynchronous Nested Benders Decomposition

cuts and/or for its master to send a new solution prior to start a new iteration. Rather, it is sufficient to receive any new data. Since in the asynchronous version there is no tight coupling between sending and receiving nodes, the information received has to be stored in communication buffers. At each new iteration cycle, a node reads the buffers and uses this information to create a new local optimization problem. It may therefore happen that the master's solution did not change from the last iteration, since the master did not yet finish his optimization. It may also happen that not all slaves sent new cuts, in this cases the master works with the same cuts as in the previous iteration. Figure 4 depicts the asynchronous algorithm using UML notation.

Asynchronous algorithms exhibit a new challenging class of algorithms since their computation graph depends on the execution. But since the whole procedure is an iteration with increasing accuracy, it does not matter if some information is lacking at any stage provided that this information is at least available for sure in some later iteration.

Experiences with asynchronous algorithms of the described type do not exist so far. Therefore, we built a prototype implementation that allows to experiment with this class of algorithms.

3.3 Implementation Requirements

Prior to the assessment of a suitable tool for building our parallel implementation of the asynchronous nested Benders decomposition algorithm, we postulated a set of requirements the resulting software has to obey:

Portability: The code should be easily portable across a wide range of platforms.

Heterogeneity: It has to be assumed that any application for financial institutions will have to cope with heterogeneous machines of any kind that are typically present in such institutions.

Modularity: A modular structure of the application should facilitate extensions to the application as well as the integration and replacement of modules, such as the LP solver.

Parallelism and synchronization: If possible, both shared memory as well as distributed memory machines (especially loosely coupled networks of PCs and workstations) should be efficiently exploitable by the application. In order to ease maintenance, parallelism and synchronization should be expressed at a high level.

Modularity is e.g. provided by object oriented approaches. A recent object oriented language that also supports heterogeneous systems and that is easily portable is the Java [11] language. However, Java lacks a high level parallel model.

Therefore, we decided to implement the optimization module using a high level Java framework called *OpusJava* which fulfills all of our requirements. Moreover, OpusJava is seamlessly interoperable with HPF which facilitates the integration of the already existing HPF modules for the financial market and the liability scenarios into the whole Aurora System.

4 OpusJava

OpusJava [19, 21] is a Java [11] component framework that provides high level means for interaction among distributed active objects in a heterogeneous and multi-lingual environment. In particular, active objects may be dynamically created, migrated, and destroyed where the user controls the resources (i.e. the physical machines and processors) onto which the object resides. The development of OpusJava is based upon the *Coordination Language Opus* [22, 24], a task parallel extension to HPF [13] which was specifically designed to support multi-disciplinary applications. OpusJava introduces the concepts developed in Opus to the Java world and is seamlessly interoperable with Opus and thus with HPF. Thereby, the Opus system, comprising the coordination language Opus and OpusJava, provides a high level framework for distributed high performance computing in a multi-lingual environment [20]. In the remainder of this section we give an overview on the salient features of OpusJava. The interested reader is referred to [20, 22, 24] for more in-depth discussions.

4.1 OpusJava at a Glance

The OpusJava distributed object model is realized by introducing a new class called *ShareD Abstraction (SDA)* which is a container that allows to instantiate, manipulate, migrate, and terminate remote objects.

In contrast to Java's remote object model (RMI [33]), OpusJava does require SDA objects neither to implement specific interfaces nor to extend specific classes. Hence, any Java class, even in pre-compiled byte-code representation, may be created as an SDA object. The OpusJava framework provides generic functions for SDA creation and interaction and hides all low level details such as socket communication, RMI, JNI, and synchronization.

Methods of SDAs may be invoked in a synchronous or asynchronous manner. Asynchronous method executions are bound to *events* with which explicit synchronization is possible. Method executions may be guarded by *condition clauses* a method may be associated with. A condition clause is a boolean function which must evaluate to `true` before the guarded method can be executed. Condition clauses are specified using a special naming convention: their name must be the name of the guarded method postfixed with "`_cond`". With help of condition clauses a consistent state of the data of an SDA can be guaranteed independently of the order the methods in an SDA are invoked. For instance, an SDA implementing a bounded buffer may guard its put and get methods to avoid over- and underflows of the buffer, respectively.

The OpusJava framework provides for mutual exclusive execution of SDA methods, thus ensuring a consistent state of the SDA data. Arguments to SDAs are passed with standard RMI semantics; i.e., standard objects are passed by value, remote objects (including SDAs) by reference. The data of SDA objects may only be accessed through the invocation of methods within the object; thus, it is implicitly private. In contrast to standard Java RMI, which does not provide for an interference-free execution of multiple concurrent method invocations within an object and leaves all synchronization burden to the user, OpusJava provides high level synchronization by means of condition clauses and events and furthermore ensures a consistent state of a remote object by imposing mutual exclusive method executions. Moreover, the OpusJava framework is able to distinguish dynamically between method invocations in remote and local objects which allows a more efficient implementation of the latter ones by by-passing the heavy-weight RMI mechanism and invoking the method directly in the local object.

Apart from the high level object manipulation and synchronization features, OpusJava provides *persistent objects* in that an SDA may be saved to persistent storage and re-loaded later on. This feature facilitates, among others, fault-tolerance strategies such as checkpoint-restart. In contrast to standard Java objects OpusJava SDAs can be destroyed explicitly; this feature is required since SDAs may not only be Java objects accessible to the garbage collector, but also HPF objects which need to be destroyed explicitly.

To support dynamic task re-location, which is a crucial feature in heterogeneous environments, a special `migrate` method is provided to any SDA. An activation of this method results in a re-location of the SDA object to the new resources.

By providing these features OpusJava enriches the standard Java RMI model with asynchronous method invocation, dynamic object creation/migration on user specified resources, and high level coordination means. These features, which are (at least partially) commonly found in similar Java systems [4, 9, 14, 32], are indispensable for distributed high performance applications. It is worth noting that OpusJava may not only be employed within distributed environments but also facilitates an efficient exploitation of shared memory architectures (SMPs). Specifically, any OpusJava SDA represents a separate thread of control; thus, creating multiple SDAs on an SMP allows the exploitation of shared memory thread parallelism together with efficient method invocation among local objects as discussed above.

The OpusJava framework has been implemented as a pure Java library without any modifications to the language and the Java Virtual Machine. Advanced features of Java such as as multi-threading, remote method invocation, and reflection are exploited within the framework. The decision to implement OpusJava as a pure Java framework makes OpusJava easily portable to any platform supported by Java; moreover, Java's support for heterogeneous systems is leveraged.

OpusJava introduces four new classes visible to the user: class `SDA`, which is a stub for a remote object, class `Event`, which is used to reference asynchronous method invocations, class `Resource` describing an SDA's resources (cf. next subsection), and class `Request` which is primarily used as wrapper for requests posed to HPF SDAs (see e.g. [20]). In the remainder of this section we provide a more in depth discussion of two crucial SDA features: *SDA creation* and *method invocation*. For more details about the internals of OpusJava the interested reader is referred to [20].

4.2 SDA Creation

The instantiation of a new `SDA` object results in the creation of a new remote object of user specified type that is under control of OpusJava on the specified resources. The (local) `SDA` object serves as a stub through which the remote object can be accessed. There are two ways of creating an SDA. First, a new remote object may be instantiated, in which case the constructor has 3 arguments: the name of the class of the remote object, an array of input parameters for the remote constructor, and a `Resource` object. Second, a *persistent* SDA which was previously stored to some persistent memory may be *loaded*. In this case, the constructor has other arguments: the file name of the persistent SDA, the code-base, i.e. the location of the file, and a `Resource` object. OpusJava requires all input arguments to constructors as well as to all other methods to be provided as an array of Java objects. Moreover, OpusJava's call-by-value semantics require all arguments to implement Java's `Serializable` interface.

A resource object contains the name of the machine on which the SDA should be allocated (typically an IP-address), and the number of processors to be used. Processors of a distributed memory multi-processor machine are currently allocated in a system dependent manner. We plan to provide the user with possibilities of extending the resource object allowing a more precise resource request formulation. The JavaSymphony framework [9], for instance, provides high level means for specifying abstract topologies on top of cluster systems in conjunction with static and dynamic system properties such as CPU load, available memory, network bandwidth, etc. Similar features are provided by the Globus resource management system [5] that comprises intelligent resource brokers, resource managers local to physical machines, and an information service that provides information about the current capabilities of resources.

4.3 Method Invocation

Two methods are provided for invoking methods within an SDA object: `call` for synchronous invocations and `spawn` for asynchronous ones. The arguments of these methods are the name of the method to be invoked and an array of input parameters (marshaled as an object array) for the SDA method. Note that all arguments passed to SDA methods have to implement Java's `Serializable` interface.

The `call` method blocks the caller until the results are available. Results of a synchronous method invocation are returned as a Java object exploiting the regular Java result mechanism.

An invocation of the `spawn` method returns an `Event` object immediately and allows the caller to resume its execution. In the Opus terminology, an event is a pointer to an asynchronous method invocation and can be used for synchronization later on. Event objects provide two methods for enquiring the execution status and for synchronizing with asynchronous method executions: `testEvent` and `waitEvent`. The former one immediately returns the current status of the method execution the event is bound with (i.e. *finished* or *not finished*), while the latter one blocks the caller until the referenced method has finished its execution and the results are available.

The results of an asynchronous method invocation are also returned as a Java object, however, the regular Java result mechanism cannot be exploited because of the asynchronous execution of both the caller and the callee. The event object which is returned as result of executing the `spawn` method is not only used as a synchronization means, but also as a container for outstanding results. In particular, an event object has a private field of type `Object` which is used for storing the method results. The public method `getResults` may be used for obtaining the results from the event. This method is blocking, i.e. it waits until the asynchronous method execution has finished and all of its results are available. Hence, any invocation of `getResults` implicitly synchronizes the caller with the asynchronous method invocation.

Events not only provide *data based* synchronization (i.e. implicit synchronization when results of an asynchronous request are being accessed), but also *control based* synchronization that is independent of any data access (by *testing* or *waiting* on an event). In that respect, events are comparable to Multilisp's *Futures* [12], but differ from *Futures* in similar Java systems [4, 14] that only provide data based synchronization.

5 Implementation and Experiments

5.1 Implementation

As outlined in Section 3, in the asynchronous nested Benders decomposition, every node of the scenario tree is associated with compute activity that is performed in parallel with all other nodes, subject to the synchronization constraints described in Section 3.2. Therefore, we decided to implement every node of the tree as an SDA object such that this parallelism can be achieved.

The activity of each node object consists of a loop in which a linear program is repeatedly solved. After having solved the linear program, the node sends the solution to its slaves and provides a cut to its master. A node may only start a new iteration if it has received new data, either new cuts or a new solution. If no new data is available, the activities of the node are suspended and will be resumed once data has been received.

Since all communication takes place in an asynchronous manner, any node may receive data at any point in time, independently of its current activity. Of course, it is not desirable that data which arrives during the course of the solution process overwrites the data that is currently used in the optimization and thereby violates the integrity of the data. To circumvent this problem, all data arriving at a node needs to be stored in a temporary buffer, associated to a node, from where it is retrieved at clearly defined points during program execution.

As a consequence, the endpoint of communications is the buffer of a node rather than the node itself. Therefore, we realized the buffer, which holds both cuts and solutions, as an SDA as well. Moreover, by implementing buffers as SDAs we can take advantage of the high level synchronization means provided by OpusJava:

- SDAs have monitor semantics; thus we do not need to synchronize the accesses to the buffer.
- Methods may be guarded by condition clauses; hence we are able to formulate the necessary synchronization conditions by means of a condition clause that is associated to the method for retrieving the data out from the buffer. Again, no synchronization is needed within the nodes themselves.

```
public class Buffer {

    // Buffer for predecessor results
    private Solution masterData;

    // Buffer for successor cuts
    private Vector   slaveData[];

    // constructor omitted

    // data from a successor
    public void putSlaveData(Cut cut, ...) {
       ...
    }

    // data from the predecessor
    public void putMasterData(Solution s) {
       ...
    }

    // read all data out from the buffer
    public Object[] getData() {
       ...
    }

    // condition clause for getData
    public boolean getData_cond() {
       // returns true if new data has been received
       // from the predecessor and/or at least from
       // one of the successors.
    }
}
```

Figure 5: The OpusJava Buffer

Figure 5 sketches the OpusJava implementation of a buffer: apart from the internal data for storing the actual solution from the predecessor as well as the cuts from the successors, methods for writing and reading this data are provided. Note the function `getData_cond` which implements the condition for executing method `getData`. A request for executing this method will only be fulfilled if the condition evaluates to true, which is the case if new data (either from the master or the slaves) hast been received.

A node in the tree is realized through the SDA `Node` whose structure is given in Figure 6. A node holds references to several other SDAs with which is has to interact: the successor nodes, its own buffer, as well as the buffers of both the successor and predecessor nodes. The constructor of a node is parameterized and requires the caller to pass a reference to an SDA object, specifically to the buffer. Thereby, any node is able to instantiate its successors passing its own buffer as parameter. The tree construction is implemented within method `init`: first, the local buffer of a node is instantiated and subsequently the successor nodes are set up. A resource request is associated with the call for instantiating the successors, thus the nodes may be distributed over the available processors according to some pre-defined strategy. Subsequently, the `init` method of the successors is invoked in an asynchronous way thus allowing a dynamic parallel growth of the tree. The result of the `init` method is a reference to the local buffer of a node which is stored in array `successorBuffer`. Since the `init` methods of all successors execute in parallel, each of these invocations is associated with an `Event` which acts as a container for the result. With a call to the `getResults` method of such an event the caller synchronizes with the callee and retrieves the results of the asynchronous call, eventually.

The main computational routine within a node is the method `iterate`. Once it is invoked within a node the `iterate` request is dynamically and in parallel propagated to the successor nodes, analogous to the tree creation procedure. After having initiated the computation in the successor nodes, a node enters the main computation loop where it repeatedly retrieves new data from its buffer (remember that the buffer itself takes care of the required synchronization), performs the local computation, and sends the results to its master and slaves. This loop is repeated until a the solution of the whole problem has been found.

5.2 Experiments

To prove the efficiency of our implementation we conducted a number of experiments on a non-dedicated cluster of four SUN Ultra 10 workstations and a dedicated Linux Beowulf cluster consisting of 16 quad-board PCs, each equipped with 4 Pentium III Xeon processors, thus comprising a total of 64 processors. The interconnection network is fast ethernet in both cases.

We solved decision problems based on scenario trees with five to eight stages, i.e. 63 to 511 tree-nodes in case of binary trees. Scenario trees with 8 stages are reasonable examples for real-world problems, however, we had to restrict the number of financial instruments within the portfolio to two. The reason for this restriction

```java
public class Node {
 // successor Nodes;
 SDA successors[];
 // predecessor, local, and successor buffers
 SDA myBuffer, predecessorBuffer, successorBuffer[];
 // problem specific data
 ...

 // constructor
 public Node (SDA buffer) {
  predecessorBuffer = buffer;
  ...
 }

 // initialization
 public SDA init(...) {
  // create local buffer
  myBuffer = new SDA("Buffer", ...);

  successors = new SDA[nrOfSuccessors];
  Event e[] = new Event[nrOfSuccessors];

  if (!leaf()) {
   // create successors passing the local buffer
   // as input arg
   for (int succNr=0;succNr<nrOfSuccessors;succNr++) {
    // compute distribution of nodes
    Resource r = new Resource(...);
    successors[succNr] = new SDA("Node",
                     new Object[] {myBuffer}, r);

    // spawn initialization of successors
    e[succNr] = successors[succNr].spawn("init",...);
   } // for

   // synchronize with successors and link buffers
   for (int succNr=0;succNr<nrOfSuccessors;succNr++)
    successorBuffer[succNr] =
                (SDA) e[succNr].getResults();
  } // !leaf
  // do other initialization
  ...
  // return the local buffer to the caller
  return myBuffer;
 } // init

 // main solution routine
 public void iterate() {
  Event e[] = new Event[nrOfSuccessors];
  if (!leaf()) {
   // invoke solution routine within successors
   for (int succNr=0;succNr<nrOfSuccessors;succNr++)
    e[succNr] = successors[succNr].spawn("iterate");
  } // !leaf

  // main loop
  do {
   // retrieve data from buffer
   Object[] data =
           (Object[]) myBuffer.call("getData");

   // solve the local problem

   if (!leaf()) {
    // send solution to successors
    for (int succNr=0;succNr<nrOfSuccessors;
                                        succNr++)
     successorBuffer[succNr].call("putMasterData",
                                            ...);
   } // !leaf

   if (!root()) {
    // send cuts to predecessor
    predecessorBuffer.call("putSlaveData",...);
   } // !root
  } while (!solutionFound); // do
 } // iterate
} // class Node
```

Figure 6: The OpusJava Node

is the numerical instability of the LP-solver employed; we are currently incorporating other solvers in order to deal with larger portfolios.

The tree-nodes have been distributed onto the available processors[1] according to two different strategies: *minimum communication* and *optimal load balancing*. The *minimum communication* strategy maps complete sub-trees onto a processor such that the number of edge cuts is *number_of_processors*-1. This strategy results in a non-optimal load-balancing, though. The *optimal load balancing* strategy distributes the nodes in almost equal parts at the cost of a higher communication overhead. We employed the multilevel recursive bisection method of the MeTiS tool [16] for this distribution strategy. Figure 7 illustrates both mapping strategies for a 15-node tree distributed onto 4 processors. The numbers in the tree nodes represent the processor a node is mapped onto. The minimum communication distribution has only 3 edge cuts wrt. 4 for the optimal load balancing, however 6 nodes are mapped to processor 0 and only 3 to all other processors.

Table 1 lists the elapsed execution time for our experiments together with the speedup obtained and the sequential scaling behavior with respect to an increase in the problem size (i.e. in the number of tree-nodes). The basis for our scaling comparison is a small tree of 31 nodes. Experiments annotated with an "*" have been conducted with the optimal load balancing mapping strategy of MeTiS; all other measurements employ

[1] On the Beowulf system, the nodes are distributed onto the available SMP PCs. In the following, we will use the term *processor* synonymously for both, a SUN Workstation and a four-CPU PC.

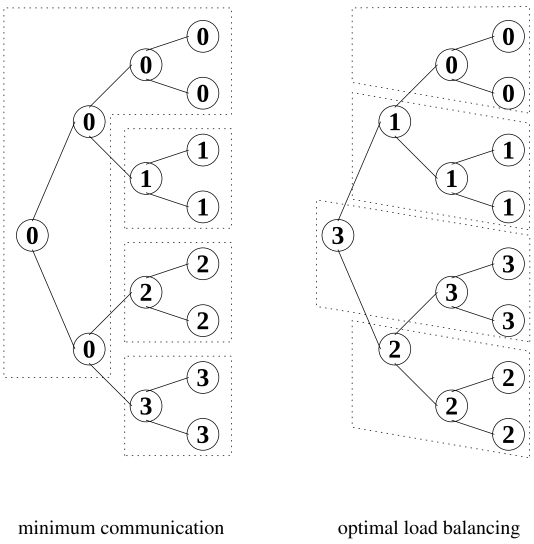

Figure 7: Tree Mapping Strategies

the minimum communication strategy.

Let T_n^p denote the execution time of a n-node problem on p processors. The scalability factor $scal_n$ for a n-node problem is computed by

$$scal_n = T_n^1 / T_{31}^1. \tag{4}$$

Figure 8 depicts the scalability of our implementation wrt. increases in problem size. We notice a superlinear increase in execution time when doubling the number of tree nodes. This effect is mainly due to the additional work the system-level thread scheduler has to perform when increasing the number of threads. As discussed above, every tree-node is represented as a separate SDA, and thus as a separate thread. While on Solaris machines this effect is minimal (e.g. the execution time for the 255 nodes problem is only 11.77 times larger than that of the 31 nodes problem) the Linux thread scheduler has severe problems in handling a large amount of threads. This results in a scalability factor of 60.66 for the 511 nodes problem although the problem size increased only by a factor of 16. As a consequence, strategies need to be devised that reduce the number of threads. For instance, we could allow an SDA to deal with multiple tree-nodes according to a multiplexing approach. Such an implementation is subject to ongoing work.

The speedup S_n^p for a n-node problem obtained on p processors is computed by

$$S_n^p = T_n^1 / T_n^p. \tag{5}$$

The comparison of the speedup results, which are depicted in Figure 9, reveal that in most cases the optimal load balancing node distribution strategy is favorable. Moreover, we see excellent speedups, which are even superlinear in some cases, on the Linux cluster. This effect is in part due to the reduced thread scheduling overhead when the tree nodes are distributed onto multiple PCs. Recall from Figure 8 that the thread scheduler has a significant impact on the execution time. On the Solaris system, this impact is much smaller, hence also the speedup obtained is not that dramatic. However, we are able to achieve a speedup of 3.48 on 4 processors if the tree is large enough. Obviously, rather small trees (such as the trees with 63 or 127 nodes) do not show a very satisfactory speedup because of the reduced computation/communication ratio.

Moreover, it can be recognized that performance might degrade with a large number of processors (64 in the case of 16 PCs) used. This is in some part due to the rather small local problems to be solved at each node. We expect the scaling behavior to improve significantly if larger portfolios can be analyzed. To prove our claim we augmented the local problems of the 255-nodes tree with some dummy work in order to simulate the work needed to analyze larger portfolios. Figure 10 depicts the results obtained. While the speedup on smaller numbers of PCs is comparable to that obtained with the small problem, on 16 PCs we are

		SUN cluster			Beowulf cluster		
p	n	$msec$	S	$scal$	$msec$	S	$scal$
1	31	646	1	1	298	1	1
1	63	1542	1	2.38	742	1	2.48
1	127	3326	1	5.14	2302	1	7.71
1	255	7611	1	11.77	6658	1	22.31
1	511	14619	1	22.60	18099	1	60.66
2	63				476	1.56	
2	63*	1164	1.33		469	1.58	
2	127				1211	1.90	
2	127*	1926	1.72		1203	1.91	
2	255				3267	2.04	
2	255*	4154	1.83		3237	2.05	
2	511				9182	1.97	
2	511*	7729	1.89		9844	1.83	
4	63				375	1.98	
4	63*	642	2.40		366	2.02	
4	127				671	3.43	
4	127*	1119	2.97		642	3.58	
4	255				1623	4.10	
4	255*	2215	3.43		1571	4.24	
4	511				3596	5.03	
4	511*	4195	3.48		3738	4.84	
8	63				359	2.07	
8	63*				362	2.05	
8	127				532	4.32	
8	127*				495	4.65	
8	255				854	7.80	
8	255*				810	8.22	
8	511				1979	9.15	
8	511*				1887	9.59	
16	63				664	1.12	
16	63*				354	2.09	
16	127				673	3.40	
16	127*				381	6.02	
16	255				885	7.52	
16	255*				594	11.21	
16	511				1584	11.42	
16	511*				977	18.51	

Table 1: Elapsed Execution Time
p...number of processors(PCs); n...number of tree nodes
msec...milliseconds; S...speedup; scal...scalability factor
n^*...MeTiS mapping

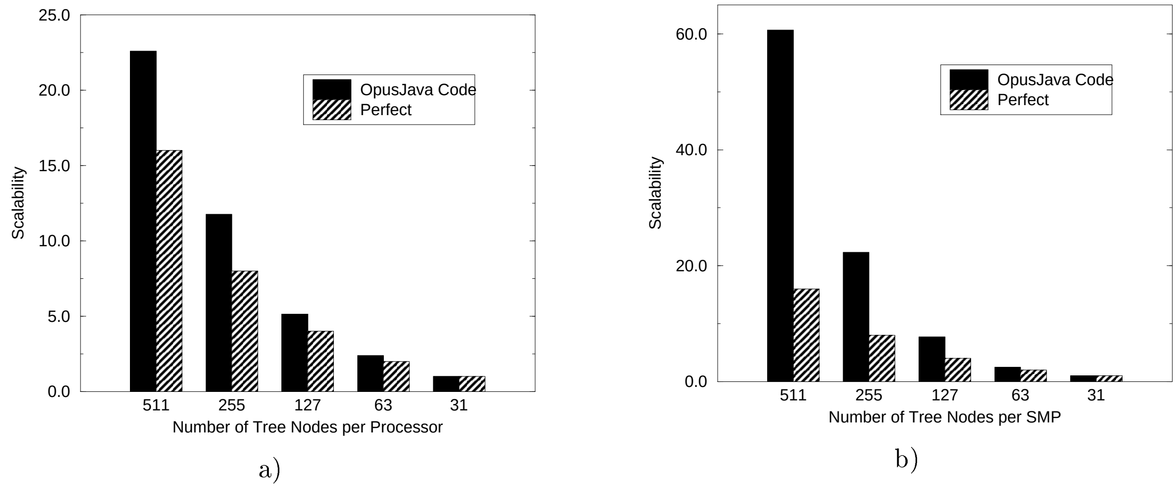

Figure 8: Scalability on a) Solaris and b) Linux Machines

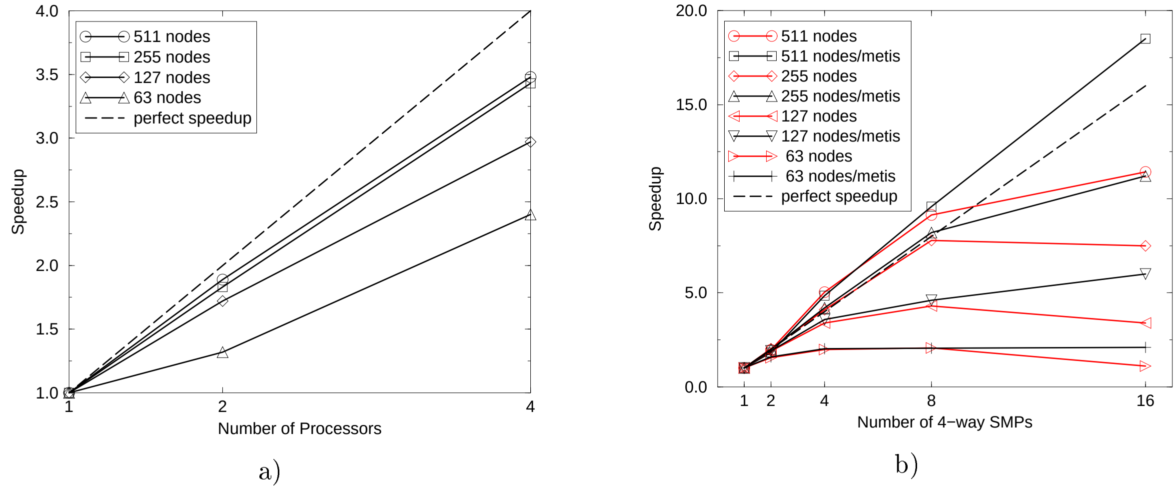

Figure 9: Speedup on a) Solaris and b) Linux Machines

now able to obtain a speedup of 15 rather than 11. With these promising results we belief that even larger clusters can be effectively exploited by our module, given that the tree and portfolio sizes are sufficiently large.

6 Conclusions

In this paper we have presented a parallel, portable implementation of the nested Benders decomposition technique for the purpose of portfolio optimization. The optimization module we developed is part of a modular decision support tool for portfolio and asset liability management, the *Aurora Financial Management System*. The solver module developed is also generic enough to be applicable to problems other than classical portfolio optimization, such as index tracking [34].

For implementing the module we used the programming language Java in order to leverage Java's portability features. Moreover, we employed a high level Java component framework, OpusJava, to express the parallelism and synchronization within our application at a high level. By using OpusJava, the coordination

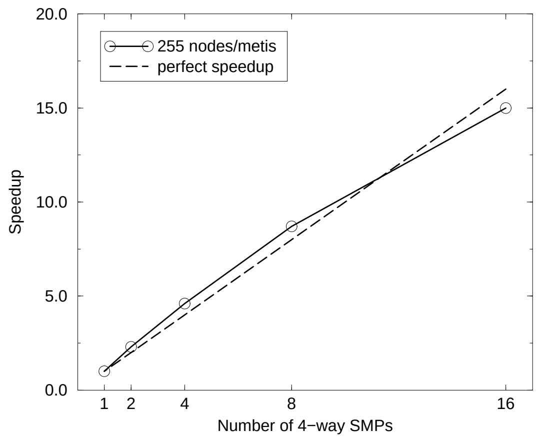

Figure 10: Speedup of Dummy Problem on Linux Machines

constraints imposed by the nested Benders decomposition algorithm can be expressed at an abstract level without sacrificing performance. Moreover, OpusJava provides means for dynamically instantiating and migrating objects which is of particular importance for future improvements such as dynamic load-balancing and tree pruning.

The efficiency of our implementation was demonstrated by a number of experiments we conducted on a cluster of Solaris workstations as well as on a heterogeneous Beowulf system. Although these experiments have been performed with small portfolios due to insufficient capabilities of the underlying systems available, the results obtained are encouraging and we expect to report on experiments with larger portfolios in the near future.

In future versions of our implementation we will specifically deal with multiplexing techniques that allow to map multiple tree nodes to a single thread of control. Thereby, the thread scheduling overhead observed should be reduced drastically. Moreover, stochastic branch-and-bound algorithms will be explored in order to dynamically prune parts of the tree that do not further contribute to the solution process. Finally, the optimization module will be integrated within the whole Aurora Financial Management System. In particular, the two modeling components implemented in HPF will be integrated with the optimization module. This task is also facilitated by OpusJava which is seamlessly interoperable with HPF.

Acknowledgments

The authors would like to thank G.Ch. Pflug and E. Dockner for numerous discussions on asynchronous nested Benders decomposition and its application to financial optimization.

The work described in this paper was supported by the Special Research Program SFB F011 “AURORA” of the Austrian Science Fund.

References

[1] K. A. Ariyawasana and D. D. Hudson. Performance of a benchmark parallel implementation of the Van-Slyke and Wets algorithm. *Concurrency: Practice and Experience*, 3:109–128, 1991.

[2] J.F. Benders. Partitioning procedures for solving mixed–variable programming problems. *Numerische Matematik*, 4:238–252, 1962.

[3] D.R. Cariño, T. Kent, D.H. Myers, C. Stacy, M. Sylvanus, A.L. Turner, K. Watanabe, and W.T. Ziemba. The Russel-Yasuda Kasai model: an asset / liability model for a Japanese insurance company using multistage stochastic programming. *Interfaces*, 24(1):29–49, 1994.

[4] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[5] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[6] M.A.H. Dempster and R.T. Thompson. Parallelization and aggregation of nested benders decomposition. *Annals of Operations Research*, 81:163–187, 1998.

[7] E. Dockner, H. Moritsch, G.Ch. Pflug, and A. Świętanowski. The AURORA financial management system. Technical report AURORA TR1998–08, Vienna University, 1998.

[8] J. Dupacova. Stochastic programming models in banking. Working paper, International Institute for Applied Systems Analysis, 1991.

[9] T. Fahringer. JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000)*, Chemnitz, Germany, November 2000. IEEE Computer Society.

[10] H.I. Gassmann. MSLiP: A computer code for the multistage stochastic linear programming problem. *Mathematical Programming*, 47(3):407–423, 1990.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesely, 1996.

[12] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[13] High Performance Fortran Language Specification Version 2.0, January 1997.

[14] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 1999.

[15] J.G. Kallberg and W.T. Ziemba. An algorithm for portfolio revision: Theory, computational algorithm and empirical results. In R. Shultz, editor, *Applications of Management Science*, pages 267–291. JAI Press, 1981.

[16] G. Karypis and V. Kumar. *MeTis: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrix, Version 4.0.* Department of Computer Science, University of Minnesota, September 1998.

[17] M.I. Kusy and W.T. Ziemba. A bank asset and liability management model. *Operations Research*, 34:356–376, 1986.

[18] L.S. Lasdon. *Optimization Theory for Large Systems*. Macmillan Series in Operations Research. Macmillan, New York, 1970.

[19] E. Laure. Distributed High Performance Computing with OpusJava. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99, 17-20 August 1999, Delft, The Netherlands*, pages 590–597. Imperial College Press, April 2000.

[20] E. Laure. *High Level Support for Distributed High Performance Computing*. PhD thesis, Institute for Software Science, University of Vienna, February 2001.

[21] E. Laure. OpusJava: A Java Framework for Distributed High Performance Computing. *Future Generation Computer Systems*, 18(2):235–251, October 2001.

[22] E. Laure, M. Haines, P. Mehrotra, and H. Zima. On the Implementation of the Opus Coordination Language. *Concurrency: Practice and Experience*, 12(4):227–249, April 2000.

[23] E. Laure, E. Mehofer, H. Moritsch, V. Sipkova, and A. Swietanowski. HPF in Financial Management Under Uncertainty. Technical Report 1999-21, SFB Aurora, October 1999.

[24] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous Computing With Data Parallel Tasks. *Parallel Processing Letters*, 9(2):275–289, June 1999.

[25] H. Markowitz. Portfolio Selection. *Journal of Finance*, 7(1):77–91, 1952.

[26] H. Moritsch, G.Ch. Pflug, and M. Siomak. Asynchronous nested optimization algorithms and their parallel implementation. In *Proceedings of the International Software Engineering Symposium*, Wuhan, China, March 2001.

[27] J.M. Mulvey. Integrative asset-liability planning using large-scale stochastic optimization. Technical report, Statistics and Operations Research series, School of Engineering and Applied Science, Princeton University, 1992.

[28] G. Ch. Pflug. How to measure risk. In U. Leopold-Wildburger, G. Feichtinger, and K.-P. Kistner, editors, *Modelling and Decisions in Economics*, pages 39–59. Physica-Verlag, 1999.

[29] G.C. Pflug and A. Swietanowski. Selected parallel optimization methods for financial management under uncertainty. *Parallel Computing*, 26:3–25, 2000.

[30] G.Ch. Pflug and A. Świętanowski. Dynamic Asset Allocation under Uncertainty for Pension Fund Management. *Control and Cybernetics*, 28(4):755–777, 1999.

[31] G.CH. Pflug and A. Swietanowski. Asset-liability optimization for pension fund management. In *Operations Research Proceedings 2000*, pages 124 – 135. Springer Verlag, 2000.

[32] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[33] SUN Microsystems. *Java Remote Method Invocation Specification*, 1998.

[34] K.J. Worzel, C.V. Vassiadou-Zeniou, and S.A. Zenios. Integrated simulation and optimization models for tracking fixed-income indices. *Operations Research*, 42(2):223–233, 1994.

[35] S. Zenios and R.L. D'Ecclesia, editors. *Operations Research Models in Quantitative Finance*. Springer-Verlag, 1994.