D I S S E R T A T I O N

High Performance Computing in Finance—On the Parallel Implementation of Pricing and Optimization Models

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
E188
Institut für Softwaretechnik und Interaktive Systeme

eingereicht an der Technischen Universität Wien
Fakultät für Informatik
von

Dipl.-Ing. Hans Moritsch
Smolagasse 4/2/8, A-1220 Wien
Matr.Nr. 77 25 716

Wien, am 23. Mai 2006

Kurzfassung

Der Einsatz von Hochleistungsrechnern in der Finanzwirtschaft ist sinnvoll bei der Lösung von Anwendungsproblemen, die auf Szenarien und deren Eintrittswahrscheinlichkeiten definiert sind. Die Entwicklung von Aktienkursen und Zinsen kann auf diese Weise modelliert werden. Gehen diese Modelle von einem bekannten Zustand aus und erstrecken sie sich über mehrere Zeitperioden, so nehmen sie die Form von Szenario-Bäumen an. Im Spezialfall "rekombinierbarer" Bäume entstehen Gitterstrukturen. In dieser Arbeit werden zwei Problemklassen behandelt, nämlich die Berechnung von Preisen für Finanzinstrumente, und die Optimierung von Anlageportefeuilles im Hinblick auf eine Zielfunktion mit Nebenbedingungen. Dynamische Optimierungsverfahren berücksichtigen mehrere Planungsperioden. Stochastische dynamische Verfahren beziehen Wahrscheinlichkeiten mit ein und führen daher zu (exponentiell wachsenden) Baumstrukturen, die sehr groß werden können. Beispielsweise besitzt ein Baummodell dreier Anlagen über zehn Perioden, wobei die Preise durch Ausmaß und Wahrscheinlichkeit des Fallens bzw. Steigens innerhalb eines Jahres beschrieben werden, $(2^3)^{10} = 1\,073\,741\,824$ Endknoten. Die Lösung solcher Probleme auf einem Arbeitsplatzrechner kann Stunden oder Tage dauern, daher ist es wünschenswert, sie mittels Hochleistungsrechnern zu beschleunigen.

Die Parallelverarbeitung stellt den wichtigsten Ansatz zur Leistungssteigerung in dieser Arbeit dar. Es wurden Verfahren zur Preisberechnung pfadabhängiger Zinsderivate auf der Basis sowohl von Monte Carlo Simulationen als auch von Rückwärtsberechnungen parallel implementiert. Im Optimierungsteil der Arbeit wurde das Dekompositionsverfahren von Benders zur mehrstufigen stochastischen Optimierung eingesetzt und sowohl in einer synchronen als auch in einer asynchronen Variante parallelisiert. Mit diesen parallelen Implementierungen lassen sich angemessene bis sehr gute Verbesserungen der Ausführungszeiten im Vergleich zu den sequentiellen Programmversionen erzielen. Darüber hinaus dienen sie als Fallstudien in der Entwicklung von Softwarewerkzeugen für Hochleistungsrechner, die im Rahmen des Spezialforschungsbereichs F011 Aurora ("Advanced Models, Applications and Software Systems for High Performance Computing") des FWF durchgeführt wurden.

Die datenparallele Programmiersprache HPF+, mit Erweiterungen für SMP-Cluster, wurde erfolgreich bei der Implementierung von Preisberechnungsverfahren eingesetzt. Eine Notation für Pfade, die es erlaubt, parallele Algorithmen auf Gittern auf hoher Abstraktionsebene auszudrücken, wurde als Erweiterung von Fortran 95 spezifiziert. Das parallele Programmiermodell eines verteilten aktiven Baumes wurde entworfen und auf der Basis von Java Threads und RMI implementiert. Parallele Implementierungen des Benders Verfahrens in Java zeigen, dass diese Sprache zur Entwicklung von Hochleistungsanwendungen geeignet ist. Sowohl OpusJava als auch JavaSymphony und das Modell eines verteilten aktiven Baums haben sich als nützliche Werkzeuge bei der Implementierung paralleler baumstrukturierter Algorithmen erwiesen.

Zusätzlich zur Parallelisierung existierender sequentieller Verfahren, der Weiterentwicklung bekannter Parallelisierungsansätze und der Verwendung spezieller Programmiersprachen und Programmiermodelle wurden Leistungssteigerungen auch durch algorithmische Verbesserungen erzielt. Eine Verallgemeinerung der klassischen Rückwärtsberechnungsmethode ermöglicht die schnellere Berechnung, und zwar in linearer statt exponentieller Zeit, von Preisen bestimmter pfadabhängiger Produkte mit "begrenzter" Pfadabhängigkeit. Dadurch wird auch gezeigt, dass besonders effiziente Ansätze zur Leistungssteigerung die Ebene der Algorithmen mit jener der parallelen Implementierung verbinden.

Abstract

High Performance Computing is useful in the field of finance for solving problems which are defined on models of financial variables in the form of sequences of scenarios along with their realization probabilities. Both the evolution of stock prices and interest rates is frequently described in this manner. Starting from a known state, and covering a future time horizon of multiple periods, these models take the form of scenario trees. As a special case, the structure collapses into a lattice, if the tree is "recombining". This work deals with the two problem classes of determining prices of financial instruments, and of determining optimal portfolios of assets, with respect to some objective function and constraints. Dynamic optimization techniques allow for multiple planning periods, whereas stochastic dynamic optimization problems take into account also probabilities and exhibit (exponentially growing) tree structures, which can become very large. As an example, a model of ten years of three assets, each of which is described by the extents and probabilities of rising or falling of their respective prices within a year, results in a scenario tree with more than one billion $(2^3)^{10} = 1\,073\,741\,824$ terminal nodes. Computation times for solving these problems can extend to hours and days, hence high performance computing techniques of achieving speed up are desirable.

The major approach for performance improvement in this work is parallel computing. It includes the parallel implementation of Monte Carlo simulation techniques as well as of backward induction methods for pricing path dependent interest rate derivatives, in particular constant maturity floaters with embedded options. In the optimization part, the nested Benders decomposition method of multistage stochastic optimization has been parallelized in a synchronous as well as in an asynchronous version. The parallel implementations obtain speedups ranging from reasonable to excellent and demonstrate the potential of high performance computing for financial applications. In addition, they served as case studies in the development of software tools for high performance computing within the framework of the Special Research Program No. F011 AURORA "Advanced Models, Applications and Software Systems for High Performance Computing" of the Austrian Science Fund (FWF).

The data parallel programming language HPF+, with extensions for clusters of SMPs, has been successfully employed in the implementation of pricing algorithms. A path notation has been specified as an extension to Fortran 95, allowing for the high level formulation of parallel algorithms operating on lattice structures. The parallel programming model of a distributed active tree has been designed and implemented on top of Java's threads and RMI. Parallel implementations of the nested Benders decomposition algorithm in Java demonstrate that this is a suitable language for high performance computing. The OpusJava component framework, as well as the JavaSymphony class library, and the distributed active tree model proved their usefulness as programming support environments in the implementation of parallel tree structured algorithms.

In addition to the parallelization of sequential existing algorithms, the improvement of known parallelization approaches, and the use of specialized parallel programming languages and programming models, an increase in performance has been achieved by algorithmic developments. The generalization of the classical backward induction method allows for the faster calculation, i.e., in linear instead of exponential time, of prices of a class of instruments exhibiting "limited" path dependence, demonstrating that highly effective approaches of performance improvement combine the levels of algorithms and parallel implementation.

Acknowledgments

I wish to thank my supervisor Professor Gerti Kappel for her indispensable support in accomplishing this thesis and for her continuous encouragement to integrate the diverse aspects of high performance computing in finance arisen from the work within the AURORA project.

I am grateful to my academic teacher and project leader of the AURORA subproject "High Performance Computing in Finance" Professor Georg Ch. Pflug for enabling the access to the exciting field of stochastic optimization, for lots of detailed discussions on nested decomposition methods, and for helping to optimize the "generalized" backward induction algorithm. For suggesting the term, I would like to thank Professor Engelbert J. Dockner, as well as for stimulating discussions on numerical pricing algorithms and for providing such a fine environment for becoming acquainted with finance theory. I am also grateful to my co-supervisor Professor A Min Tjoa for supporting my thesis at the Institute of Software Technology and Interactive Systems of the Vienna University of Technology.

Also, I am thankful to the speaker of the AURORA project Professor Hans P. Zima for providing the first environment to get familiar with parallel programming and for introducing me into the science of automatic parallelization. I wish to thank Professor Barbara M. Chapman for her support while gaining experience with parallel programming environments and their application, and for an initial outline of this thesis. I am grateful to Professor Stavros A. Zenios for inspiring statements on high performance computing in finance, and hints related to sampling techniques. I appreciate the opportunity to work in excellent research groups at the University of Vienna, at the former Institute for Software Technology and Parallel Systems, the European Centre for Parallel Computing at Vienna, the Department of Finance, and the Department of Statistics and Decision Support Systems.

Thanks to all the members of these groups who contributed to this work. I would like to thank Professor Andrea Gaunersdorfer for helping me to get used to the field of finance as well as Dr. Helmut Elsinger for giving me countless ad-hoc crash courses in financial mathematics. I am grateful to Professor Siegfried Benkner, Professor Thomas Fahringer, and Dr. Erwin Laure for numerous fruitful and motivating discussions on the parallelization part. I am thankful to all the participants of the AURORA project for the opportunity to work together, in particular to Professor Peter Brezany, Dr. Ian Glendinning, Dr. Ronald Hochreiter, Dr. Alexandru Jugravu, Dr. Maria Lucka, Professor Eduard Mehofer, Dr. Bernhard Scholz, Dr. Viera Sipkova, and Dr. Artur Świętanowski. Thank you to Dr. Mariusz Siomak for help with the optimization model specification. This work was supported by the Austrian Science Fund (FWF) as part of the Special Research Program No. F011 "AURORA".

I appreciate all the encouragement of Professor Robert Glück, Professor Klaus Gugler, Professor Christian Keber, Mag. Ulrike Keber-Höbaus, Professor Michaela Schaffhauser-Linzatti, Dr. Matthias Schuster, and Professor Katharina J. Srnka, who is particularly thanked for her help with making the interdisciplinary presentation comprehensible. Thanks to all mentioned who contributed to the proof reading of this thesis.

I thank my parents, Ing. Peter and Maria Moritsch for providing the opportunity to take my studies in Vienna.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The thesis deals with the application of high performance computing to problems arising in finance. This introduction lays the ground for readers coming from the both communities, economics and computer science. The first section introduces into basic concepts of finance and high performance computing. The second section gives a detailed account on related work. The third section summarizes the contribution made by the author.

1.1 Motivation and Background

Applying high performance computing techniques to problems in finance requires adequate programming support environments and represents an instructive opportunity of developing, using and evaluating these. Both high performance computing and computational finance are special fields for computer scientists as well as for economists. The work described lies in the intersection of these two fields. The aim of this section is therefore to create, for readers of both communities, an understanding of the problems tackled.

1.1.1 A Tour on Finance

The overall issue investigated in this research concerns the utilization of high performance computing techniques to solve problems in finance. "Finance can be defined as the art and science of managing money" [63]. More precisely, finance studies the allocation and use of money and other assets and the management of these, with particular focus on the risks undertaken [153].

Risk Management The term financial management under uncertainty makes clear that future events which have an effect on financial decisions and the value of investments cannot always be predicted. Risk management distinguishes several types of financial risk, e.g., market risk associated with the movements of interest rates, and currency risk caused by exchange rate fluctuations. Other types of risk are shape risk, volatility risk, sector risk, credit risk, and liquidity risk [39]. Combining different assets or asset classes within a portfolio is a classical means to diversify and thus reduce risk [84].

Portfolio Optimization In general, a portfolio can be constructed in such a way that a relation between return and risk specified by the investor is maintained. The composition of the portfolio is obtained as the solution of an optimization problem, which models the desired risk-return characteristics including exposure to specific risks via an objective function and constraints. As an example, in the classical mean-variance portfolio optimization model developed by Markovitz, risk—quantified by the variance of the portfolio return—is minimized whereas the portfolio must achieve a certain minimum return within some time period [106]. The objective function is quadratic, and the optimization problem can be solved using quadratic programming techniques. The design and definition of a useful optimization model[1] requires a portfolio manager to analyze which risks are accepted. The choice of a suitable risk measure is a research issue on its own [126]. Optimization models are employed in particular in financial engineering, a term referring to the process of designing new financial products to be added to the market, with risk characteristics fitting specific investors needs and preferences [39]. Financial innovation sees a continual introduction of products tailored, in a computer-aided design process, to the demands made by risk management [160].

Operations Research Risk management, portfolio optimization, and financial engineering are important issues in operations research (OR), a field dealing with the interdisciplinary solution of planning problems using developments in computer science and mathematics [57]. Systems are described using mathematical models, which allow for the comparison of decisions

[1]The terms *model* and *problem* are synonymous in this context, as the mathematical problem formulation inherently represents a model.

and strategies taking into account trends and uncertainty [4]. OR provides methods to describe complex decision problems in a quantitative manner, as well as algorithms to solve these problems on a computer, e.g., the classical Simplex algorithm by Dantzig for solving linear programs [40].

Multistage-Optimization Portfolio optimization models are extensively used by banks and companies offering financial services. In the Markowitz model, an investor plans portfolio decisions only for a single time period, with future consequences of the current decisions not taken into account. A more general approach includes a multi-period planning horizon allowing for rebalancing, i.e., altering the composition of the portfolio in every time period. The uncertain future returns of the individual assets are modeled as a set of different scenarios jointly with their realization probabilities. The resulting financial planning model is multistage and stochastic in nature, and can be solved using stochastic optimization techniques such as multistage stochastic programming. Dynamic portfolio management problems can also be formulated as stochastic optimal control models [25]. An alternative approach defines parameterized decision rules and optimizes their parameters [119].

Tree Structured Problems Linking every scenario in a multi-period stochastic model to its successors yields a tree structure. Tree structured models for problems in which decisions are made in stages have been employed, besides their application in portfolio management [23, 45, 78, 89], in resource acquisition [14] and natural resource management [2, 124, 143]. For the generation of scenario trees, future probability distributions of asset returns have to be represented in a discrete way with a small number of tree nodes. Various methods exist, based on historic data and expert knowledge, such as random sampling, matching of the moments of a distribution, and minimizing special metrics [88, 73, 127]. A large number of scenarios allows for more accurate models, but at the same time increases the computational effort of solving the problem. In general, the tree size grows exponentially with the number of time periods. Multiple decision dates and sufficient approximations of the return distributions easily result in large scale models with millions of scenarios [65].

Solution Techniques As far as its solution is concerned, a stochastic problem can be formulated as its so called deterministic equivalent problem, a linear program which could be solved e.g. by means of the Simplex method [85]. However, because of the huge and extremely sparse constraint matrix arising from a large tree structured problem, this approach is rather inefficient. Model decomposition algorithms, e.g., Benders decomposition, provide for an alternative [5, 41, 138]. The whole optimization problem is defined as a set of relatively small node problems, each associated with a node of the scenario tree. "Linking" constraints of the local problem define dependencies on other nodes, in particular on the predecessor and successors. The local node problems can be solved autonomously, though requiring data from other nodes in order to satisfy the linking constraints. By repeatedly solving and updating their local problem, all nodes contribute to the overall solution.

Asset Price Models The evolution of asset prices can be described by mathematical models in terms of probability distributions. According to the observation of unexpected changes in the market, asset prices are seen as variables whose values change randomly over time. Such variables are said to follow a stochastic process [72]. Finance theory has developed—in the so called "age of quants"— a variety of models of asset prices, e.g., the classical model by Black and Scholes, who made the assumption that asset prices follow a geometric Brownian motion [75]. The model can be formulated as the stochastic differential equation $dS = \mu S dt + \sigma S dz$, where dS represents the change of the asset price within a small interval of time dt, and μ is a constant quantifying a locally deterministic proportional increase (the "drift") of the price. Random proportional deviations have the form of Gaussian noise described by a constant σ and increments dz of a Wiener process, which are normally distributed with mean zero and variance dt. The parameters σ and μ are determined via a calibration process such that the model fits observed market data.

Derivatives Models of asset prices and other economic variables, such as interest rates, are in particular needed in case of derivative products. A financial derivative is an instrument whose value depends on other, more basic underlying variables, e.g., a bond with its coupon payments depending on current interest rates (a "floater"), or an option. An option is a contract whereby the holder has the right to buy (or sell) an underlying asset by a certain date for a certain

price [75]. The holder has to pay a price for this right which reflects expected profits through exercising it, which in turn depend on the future evolution of the underlying asset price.

Pricing Models In general, the derivatives pricing problem can be stated as follows: what is the price today of an instrument which will a pay cash-flow in the future, depending on the value of the underlying ? The widely used non-arbitrage principle states that two financial contracts that deliver identical future cash-flows must have the same price today. The relation between the price of the derivative and the underlying is again described by mathematical pricing models. In certain cases, analytical pricing formulas can be derived, using stochastic calculus. For example, the Black-Scholes option pricing model leads to a (deterministic) partial differential equation, whose solution is the famous Black-Scholes formula [19]. However, for many complex "exotic" products, in particular if they are path dependent, i.e., their payments depend on past values of the underlying, analytical solutions do not exist.

Numerical Pricing Techniques Numerical solution methods are applied which employ approximations of the mathematical models, e.g., finite difference methods replace partial differentials with finite differences [140]. Using Monte Carlo simulation, a stochastic process is sampled by randomly drawing paths of the underlying, enough to represent the future evolution. The prices corresponding to each path are then averaged [21]. Binomial or trinomial lattice methods discretize the distribution of the underlying and describe the future as a finite set of states with transition probabilities. Using backward induction, the prices at all states are determined backwards in time, i.e., from the future to the present [37]. The computational effort of these methods is increasing with the desired accuracy of the solution. They principally discretize time, and the number of paths to be processed for the pricing of path dependent products grows exponentially with the number of time steps. Backward induction shows a much lesser polynomial effort but cannot be employed for path dependent products. The prices calculated by numerical methods already represent meaningful information, e.g., in the design and marketing of new products, but in particular they are needed for the definition of scenarios in a stochastic optimization model.

Computational Finance Advances in computing and algorithms have established a new interdisciplinary area combining finance and computer science. Computational finance aims at gaining better understanding of financial markets strongly relying on the use of computer programs to simulate market phenomena. Among other things, it studies artificial markets based on game theory and employs neural networks and genetic algorithms in forecasting markets [149]. The effective exploitation of new computational methods helps financial institutions to improve their decisions and reduce risk via portfolio optimization. The authors in [95] state that it is desirable to support decision makers with computerized decision support systems that are on the one hand able to cope well with the huge amounts of data and on the other hand model the uncertainty aspects. Realistic models, taking into account a lot of risk factors, result in a computational effort which requires high performance computing.

1.1.2 A Tour on High Performance Computing

High Performance Computing Systems Whereas high performance computing (HPC) cannot be defined precisely in terms of quantitative parameters, the use of high performance computer hardware is essential [67]. A high performance computing system provides more computing power than is "generally available" [35], in particular it goes beyond the performance of a desktop machine [67, 35]. Due to advances in hardware technology, the threshold for "high performance" is steadily increasing. Today a performance in the range of TeraFLOPS—10^{12} floating point operations per second—can be achieved. High performance computing systems are either computer clusters or highly parallel supercomputers. The term supercomputer refers to computing systems—including hardware, operating system and application software—that provide "close to the best currently achievable" [67] sustained performance on demanding computational problems, or simply to "the most powerful machines that exist at any given point in time" [163]. High performance computing systems have been used in various application areas, such as climate modeling, weather forecasting, computational fluid dynamics, molecular dynamics, image processing, and computer vision.

Parallelism Besides higher clock speeds and shorter memory access times provided by technology, parallelism is the dominating concept for performance improvement. In a parallel

multiprocessing system, many processors are simultaneously executing a program, hence the overall execution time may be reduced by orders of magnitudes. Program development on these systems is clearly more complex than on sequential machines, because it requires to take care of the additional dimension of parallelism. Apart from trivial cases, the individual processors need to be synchronized and have to exchange data in order to efficiently cooperate.

Parallel Architectures In a so called shared-memory multiprocessor, the processors are "tightly" coupled via a single memory, thus sharing all the data and directly providing it to other processors. Symmetric multiprocessor (SMP) systems, workstations or PC's containing a relatively small (up to 32) number of processors with uniform access to a shared memory, follow the trend. In a so called distributed-memory system, each processor has its own exclusive memory. Data needs to be explicitly transfered to other processors via message passing using special communication operations, resulting in an ever increased programming effort. However for technical reasons, much more processors can be coupled in this "loosely" way (up to $n\,10^3$). Single processors as well as multiprocessor systems, e.g., SMPs, can be part of a distributed-memory machine. Both cases are subsumed under the more general term "compute node". Combining a number of workstations or PC's of the same or different type, permanently or temporarily, in a network or cluster allows for building a low budget distributed-memory system. The popular Beowulf cluster consists of compute nodes and an interconnection network which are mass produced, relatively inexpensive standard components, and runs open source software [145]. Shared-memory parallelism within SMP compute nodes can be combined with distributed-memory parallelism in a cluster of SMPs, a hybrid architecture becoming increasingly important.

Parallel Programming Support In any case of parallel architecture, the task of programming needs some kind of high level support, allowing the programmer to focus on the algorithm to be implemented rather than on low level details of synchronization and communication. The availability of this support, in the form of special programming languages, automatic parallelization [60, 61], program libraries, and programming tools, is seen even as crucial for the broad development of software for high performance computing systems. A lot of research has been devoted to this issue during the last decades.

Parallel Programming Models An interface separating higher and lower level issues in a parallel program is termed a model of parallel computation or a parallel programming model, respectively [142]. It represents a means to express parallel algorithms and simplifies the program structure through providing operations to the programmer at a higher level than the underlying (hardware) architecture.[2] For example, the classical Single-Program-Multiple-Data (SPMD) model of computation restricts the processors in a parallel system to execute the very same program. Every processor is assigned a portion of the whole data, and, according to the so called owner-computes rule, it performs—in parallel with the rest—just the computations on its "own" data. In this way, the data as well as the computational work is divided and "distributed" amongst all processors. This kind of parallel computation is also referred to as data parallel. A suitable parallel programming model, in the form of a programming language or program library, significantly reduces the complexity of program development on parallel systems. Among others, the parallel programming languages Vienna Fortran [162] and High Performance Fortran [69] follow the SPMD model.

1.1.3 Use of High Performance Computing in Finance

The study of large scale financial planning problems requires a combined approach of modeling techniques, algorithms and high performance computing [65, 160]. Significant shorter computation times can make previously intractable problems really tractable. In general, faster responses of a system are seen as a benefit (and nothing less than a driving force for technical progress). They can in particular enable new qualities of using software tools, if the effect of changes to model parameters can be displayed to the user sufficiently fast to allow for interactively exploring parameter spaces, e.g., in the computer-aided design of financial products and in research. Finally, as an economic aspect in the area of managing money, both the early availability of data and its reliability, by reducing modelling errors, represent a profit potential of investment strategies and justify the hypothesis of a performance improvement of financial operations through high performance computing.

[2] For this reason, a parallel programming model can also be viewed as an abstract machine.

1.2 Related Work

The presentation of related work is divided into three parts, each of which corresponds to an area of contribution (see Figure 1-1).

1.2.1 Pricing

The book [75] describes models of asset prices and interest rates as well as the numerical pricing of derivative instruments. Interest rate models and their application to the pricing of interest rate dependent products are further described in [18, 36, 76, 161]. Monte Carlo methods for pricing financial instruments are presented in [21, 22] and in the book [64]. The paper [62] presents a method to accelerate the calculation of hedge sensitivities by Monte Carlo simulation and applies it to the LIBOR market model. A description of a wide range of pricing algorithms is given in the book [32]. Polynomial algorithms for pricing path dependent instruments with special cash-flow structure are presented in the report [116]. Numerical pricing methods are becoming increasingly important due to the continuing introduction of exotic instruments with complex cash-flows [122], and due to the integration of optimization and simulation models for the design of financial products [34] and for the tracking of fixed-income indices [154].

Parallel algorithms have been applied to the numerical pricing of derivatives, in particular of path dependent fixed income securities [77, 159]. The shared- as well as distributed-memory parallelization of multinomial lattice methods for pricing European- and American-style options based on hidden Markov models is presented in [26]. Parallel algorithms for pricing multi-asset American-style Asian options employing the binomial method are discussed in [74]. An architecture independent parallel approach of option price valuations under the binomial tree model is described in [59]. The paper [125] presents an approach of optimizing both sequential and parallel option pricing programs through the use of performance models. The paper [103] compares the performance of the standard Monte Carlo method with a method approximating high dimensional integral problems through (t, m, s)-nets generated by a parallel algorithm. The parallelization of Monte Carlo methods with focus on the underlying parallel pseudo-random number generation is discussed in [123]. A parallel financial engineering C++ library based on MPI is described in [101].

1.2.2 Optimization

An overview of stochastic programming and tree structured models is given in [43, 48, 49, 85]. Implementations of Simplex based methods for solving the deterministic equivalent are described in detail in [38, 79, 146]. Interior point methods are presented in [105, 151, 155], and augmented Lagrangian methods in [13, 44, 136]. Decomposition techniques are discussed in [12, 138]. Benders introduces the method in [5], as the dual form of the Dantzig-Wolfe algorithm [41]. Nested Benders decomposition extends to multistage, i.e., tree structured, problems [16], an early implementation is [58].

Parallel implementations are described in [10, 20, 102, 156], parallel Benders decomposition in [3, 17, 120]. A combination of parallelization with the aggregation of decision stages is presented in [46]. The papers [128, 152] review parallel algorithms for large-scale stochastic programming and raise the issue of asynchronous decomposition techniques. An implementation of a nested decomposition method based on regularized decomposition [135] on a (shared-memory) multiprocessor system, both in a synchronous and in an asynchronous manner, is discussed in [137]. This work deals with implementation techniques for parallel tree structured optimization algorithms, including varying coordination mechanisms and scheduling strategies. As a tool, it provides a distributed active tree framework for a high level, object-oriented specification of parameters (see Section 1.2.3). Within the AURORA project (see Section 1.3), the nested Benders decomposition method has also been implemented using OpusJava [98], and a distributed active tree with its coordination layer on top of JavaSymphony [54]. The book [29] describes a large number of parallel optimization algorithms in detail, and the book [1] deals with the field of parallel metaheuristics.

1.2.3 Programming Support

The advantages of a hybrid programming model based on MPI and OpenMP as opposed to a unified MPI-only model are investigated in [27, 68]. On the Origin2000 system, data placement directives form a vendor specific extension of OpenMP [141]. Compaq has extended Fortran for Tru64 UNIX to control the placement of data in memory and the placement of computations that operate on that data [15]. A set of OpenMP extensions, similar to HPF mapping directives, for locality control is proposed in [31], and a high-level programming model that extends

the OpenMP API with data mapping directives, is proposed in [100]. This model allows for controlling data locality with respect to the nodes of SMP clusters.

A classification of Java environments, language extensions, libraries, and JVM modifications for high performance computing in Java is given in [104]. Several language extensions have been defined for writing high performance applications in Java. Spar [150] provides extensive support for arrays such as multidimensional arrays, specialized array representations, and tuples. It supports data parallel programming and allows for an efficient parallelization via annotations. Titanium [157] is a Java dialect with support for multidimensional arrays. It provides an explicitly parallel SPMD model with a global address space and global synchronization primitives. A high-performance numerical Java library as well as programming techniques for numerical Java codes are described in [110]. A parallel random number generator Java library is presented in [33]. HPJava [28] adds SPMD programming and collective communication to Java. The Java implementation of the distributed active tree model is specifically targeted at a high level formulation of tree structured iterative algorithms, a shared address space, collective communication, and a highly modular architecture.

The papers [51, 52, 55, 56, 134, 148] are dealing with the important aspects of evaluation and estimation, respectively, of the performance of parallel programs and present performance tools developed within the AURORA project. The paper [71] describes the utilization of grid computing for financial applications. A method for specifying concurrent programs at the level of UML is presented in [109]. The paper [87] discusses model checking for UML state diagrams and automatic code generation techniques.

1.3 Contribution

This work describes parallel implementations of numerical pricing methods as well as of large scale stochastic optimization methods. These implementations form components of the AURORA *Financial Management System*, a decision support system for asset and liability management under development within the framework of the special research program AURORA "Advanced Models, Applications and Software Systems for High Performance Computing". The research agenda of AURORA [144]

> "focuses on high-level software for HPC systems, with the related research issues ranging from models, applications, and algorithms to languages, compilers, and programming environments. The major goals include pushing the state-of-the-art in high-level programming paradigms, languages, and programming environments for HPC systems, the study and development of new models, applications, and algorithms for HPC, ...
>
> AURORA is a distinctively interdisciplinary project, based upon a highly synergetic cooperation of sub-projects ranging across a broad range of disciplines. The explicit common goal that unites these projects is to push the state-of-the-art in the field of software for HPC systems. At the heart of the project lies the cooperation between language and tool developers on the one hand, and application designers on the other hand. Synergy is at the core of AURORA—none of the participating institutions alone has the full expertise required to make a real contribution to this difficult area of research and development."

As stated in [95], the AURORA Financial Management System is intended to support a decision maker in finding portfolio allocations that will ensure meeting future obligations, safety of the investment, and a reasonable profit. An investor chooses a portfolio of various assets or asset classes, in such a way that some objective, including a risk measure, is maximized, subject to the uncertainty of future markets' development and additional constraints such as the investor's preferences and budget restrictions [129]. The system contains pricing models for financial instruments such as stocks, bonds, and options, implemented within a pricing module, and repayment models for liabilities. The core of the system is a large scale stochastic multi-period optimization problem, which is solved by an optimization module. According to the research agenda, focus was not only on the development of a high performance application as such, but also on the development process, including the software tools to support it.

Figure 1-1 shows how the contribution of this work—displayed in shaded boxes— fits into the domain of a high performance financial management system, regarding the structure and development of both the system and suitable programming tools. Models of prices of underlyings as well as of derivatives are the starting point for the development of pricing algorithms. As far as the implementation on a computer is concerned, it is desirable that it is parameterized

with respect to a variety of instruments. A suitable notation for the specification of financial instruments has to be chosen. As of any software exceeding a certain size, the development of a parallel pricing tool requires a design. Typically, the parallel implementation employs some kind of programming support, either by specialized languages or libraries. Parameters of pricing models, after being calibrated to market data, are passed to the pricing kernel. The resulting prices serve as an input to the scenario tree generation.[3] The design of an optimization model yields a formal model specification describing the objective function and constraints. A problem generation procedure combines the specification with the scenario tree and produces the complete tree structured description of the optimization problem to be solved. A parallel solver implements (based on a software design) an optimization algorithm and calculates the optimal solution. A parallel programming model is designed and specified, then implemented either as a programming language (extension) or a program library (extension), and finally employed as a tool in parallel software development.

In this work, parallel pricing procedures for interest rate derivatives on the basis of the Hull and White trinomial interest rate lattice model have been implemented in High Performance Fortran, adopting classical vectorization and data parallel approaches. The parallel pricing kernel developed includes the classical backward induction and Monte Carlo simulation methods, and provides for the specification of path dependent interest rate derivatives in terms of their cash-flow behavior depending on a history state. Constant maturity instruments with embedded options are priced via a nested simulation approach. In addition, a hybrid implementation has been developed, employing language extensions for clusters of SMPs, which combines distributed- and shared memory parallelism.

The backward induction method has been generalized in order to allow for pricing a subclass of path dependent instruments exhibiting a restricted form of path dependence identified and termed "limited", with a computational effort growing linearly with the number of time steps. The generalized version includes the special cases of path independence and full path dependence. The algorithm is formulated in a parallel path notation which has been designed, as an extension of Fortran 95, for the clear description of algorithms operating on paths in lattice structures.

[3] In addition, the scenario tree generation may directly utilize market data.

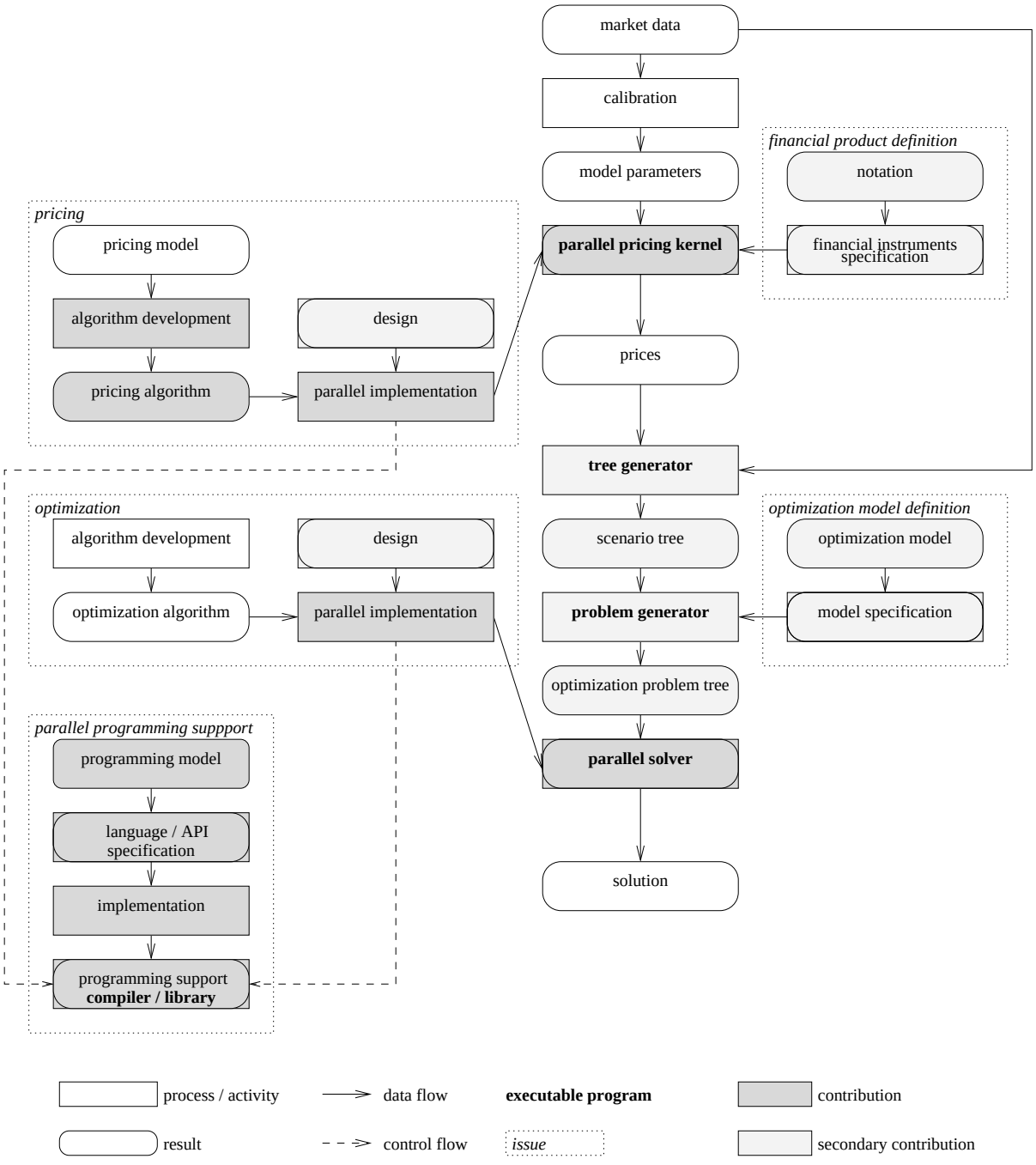


Figure 1-1: Parallel implementation of a financial management system

The nested Benders decomposition algorithm for solving multistage stochastic optimization problems has been parallelized by means of Java, according to the object-oriented paradigm. A new version of the algorithm has been implemented, based on an asynchronous computation scheme, and compared with the synchronous one.

The parallel implementation follows the programming model of a distributed active tree, which has been designed to support the parallel and distributed implementation of a whole class of tree structured algorithms with varied synchronization requirements. It provides high level operations for the coordination of simultaneously active tree nodes allocated on the compute nodes of a parallel system. An application programming interface has been specified, implemented as a Java library using threads and Java remote method invocation (RMI), and employed in the parallelization of the decomposition algorithm. In addition, the latter has been parallelized using OpusJava and a distributed active tree implemented on top of JavaSymphony. An optimization problem generator has been developed combining the generation of both the scenario tree and of local node problems for wealth maximization and index tracking optimization models.

1.4 Structure of the Thesis

Chapter 2 describes programming languages, libraries, and programming models employed as support in the parallel program developments undertaken. Chapter 3 presents financial application problems, found in the areas of pricing and optimization, to be solved by means of high performance computing techniques. Chapter 4 describes the (sequential) algorithms applied. Chapter 5 discusses in detail the parallelization of these algorithms and presents performance results. Chapter 6 draws conclusions of this work and lists a selection of open issues.

Chapter 2

Programming Support Environments for Parallel Systems

Program development on parallel architecture is more complex than on sequential machines, therefore an adequate support of the programming task is required. The parallel implementations undertaken in this work employ a number of parallel programming models, languages, and libraries. In this chapter, relevant features of the programming language HPF+ (see Section 2.1), the Vienna Fortran Compiler (see Section 2.2), HPF extensions for Clusters of SMPs (see Section 2.3), as well as of the Java frameworks OpusJava (see Section 2.5) and JavaSymphony (see Section 2.6) are presented, based on the overviews given by the authors in [54, 95, 98, 113]. The aim of efficient program development suggested the proposal of Fortran extensions for path constructs (see Section 2.4), as well as of a programming model for tree structured parallel algorithms (see Section 2.7.1) and its implementation in Java (see Section 2.7.2).

2.1 HPF+

The high-level programming language HPF+ [6, 11] has been used in the parallel implementation of pricing algorithms, because low-level explicit parallel programming with MPI [107] or threads [80] tends to be cumbersome and error-prone. High Performance Fortran (HPF) [69] is the standard language for data parallel programming on shared- and distributed-memory parallel architectures. HPF is a Fortran 90 language extension [81] and utilizes the array handling

features of Fortran 90 for expressing parallel array operations. It provides compiler directives for distributing data and for specifying parallel loops. HPF+ is an extended version of HPF, designed in cooperation with application developers from industry. It improves the data distribution model of HPF and includes additional features that facilitate the efficient compilation of irregular applications. In the following, relevant HPF+ language features are summarized.

Using the `BLOCK` data distribution, data is distributed in equally sized chunks onto the available processors. The block-wise data distribution guarantees both data locality and load balancing. Since loops represent the main source of parallelism and static compiler analysis cannot always detect parallelism, the user guides the parallelization via directives. The `INDEPENDENT` directive states that a loop can be executed in parallel. It can be combined with other constructs. The `ON HOME` clause specifies a work distribution. Every processor which owns a copy of the variable referred to in the `ON HOME` clause executes the associated loop iteration. The `RESIDENT` clause asserts that data is distributed in such a way that no communication is required during the execution of the loop.

The HPF+ `REUSE` clause indicates that both loop-iteration- and communication-schedules do not change from one instantiation of the loop to the next. The parallelization overhead is reduced significantly, because the schedules are computed only once and reused later on. Reduction operations, for example, summations, are marked explicitly with the `REDUCTION` keyword. The compiler substitutes optimized library calls for these operations. In HPF, procedures called from within parallel constructs must have the `PURE` attribute to assert that no side effects prevent parallelization. However, `PURE` does not provide any information about data accesses, which in general result in communication. HPF+ introduces the `PUREST` directive, which also asserts that the invocation of a procedure does not require any communication.

2.2 The Vienna Fortran Compiler

The Vienna Fortran Compiler (VFC) is a parallelizing compilation system which translates HPF+ programs into explicitly parallel message passing programs [8]. The parallelization techniques are based on the SPMD model, where each processor executes the same, indeed parameterized, code. The generated SPMD program calls MPI routines for data transfers among

the compute nodes of a distributed-memory parallel system. The data distribution directives `GENERAL_BLOCK` and `INDIRECT` of the VFC support in particular the parallel implementation of irregular applications.

Emphasis for both language and compiler development has been on the parallelization of irregular loops. Since it is in general impossible to determine access patterns for irregular loops at compile-time, neither independence of data accesses within a loop can be proved, nor required communication operations can be derived statically. The first problem is tackled by the `INDEPENDENT` directive, which allows for marking a loop as parallelizable, and the latter by a runtime parallelization strategy following the Inspector/Executor paradigm [139].

The execution of an irregular loop is performed in several phases. In the work distribution phase, the iteration space of a loop is distributed onto the compute nodes. In the inspector phase, access patterns and the communication schedule are derived. In the final executor phase, the loop is actually executed, including communication according to the communication schedule. Since the computation of communication schedules can be very costly, the reuse of loop-invariant schedules helps in achieving high performance. If, via the `REUSE` clause, the access patterns and array distributions are asserted to remain invariant, the communication schedule can be reused in subsequent executions of the loop [6].

2.3 HPF Extensions for Clusters of SMPs

HPF extensions for clusters of SMPs enhance the functionality of the HPF mapping mechanism such that the hierarchical structure of SMP clusters can be exploited and an HPF compiler can adopt a hybrid distributed-/shared-memory parallelization strategy [9, 24]. It is achieved through combining distributed-memory parallelism based on message passing, e.g. MPI [107, 108], across the compute nodes of a cluster, and shared-memory parallelism, based on multi-threading, e.g. OpenMP [121], within compute nodes.

A number of extensions allow for optimizing existing HPF codes for clusters of SMPs without the need for modifying existing mapping directives, by introducing the concept of abstract node arrays. A *processor mapping* from an abstract node array to an abstract HPF processor array represents a simple means for describing the topology of SMP clusters (see Figure 2-1).

Alternatively, hierarchical data mappings can be specified. In a first step, data arrays are distributed onto the compute nodes of a cluster by means of standard HPF data distributions. These mappings are referred to as *inter-node data mappings*. In a second step, an *intra-node data mapping* of the local data at a compute node may be specified, for use by a compiler in order to derive a work distribution of the set of threads running concurrently on the processors of a compute node.

2.3.1 Abstract Nodes Arrangements and Processor Mappings

The hierarchical structure of an SMP cluster is described by mapping abstract processor arrangements to abstract node arrangements. The `NODES` directive declares one or more rectilinear *node arrangements*. For the specification of processor mappings, HPF data distribution mechanisms as provided by the `DISTRIBUTE` directive are employed. The intrinsic function `NUMBER_OF_NODES()` yields the total number of compute nodes available for executing the program and may be combined with the `NUMBER_OF_PROCESSORS()` intrinsic function in order to specify the topology of a cluster in a parameterized way.

```
!HPF$ PROCESSORS p(16)
!HPF$ NODES n(8)

!HPF$ DISTRIBUTE p(BLOCK) ONTO n

      REAL a(na)
!HPF$ DISTRIBUTE a(BLOCK) ONTO p
```

(a)

```
!HPF$ PROCESSORS p(4,8)
!HPF$ NODES n(4)

!HPF$ DISTRIBUTE p(*, BLOCK) ONTO n

      REAL b(nb1,nb2)
!HPF$ DISTRIBUTE b(CYCLIC, BLOCK) ONTO p
```

(b)

Figure 2-1: Node arrangements and processor mappings

Example (a) in Figure 2-1 specifies the hierarchical structure of an SMP cluster consisting of eight compute nodes, with two processors each. Example (b) defines an SMP cluster with four compute nodes, each of which comprises 4×2 processors. Using processor mappings, the hierarchical structure of SMP clusters may be explicitly specified without changing existing HPF directives. Based on a processor mapping, a compiler can apply a cluster specific parallelization strategy in order to efficiently exploit both distributed- and shared-memory parallelism.

2.3.2 Distributed-Memory versus Shared-Memory Parallelism

In a processor mapping, the combination of distributed- and shared-memory parallelism is specified for each dimension of a processor arrangement. There are two extreme cases. If for a dimension of a processor arrangement the character "`*`" is specified as distribution format, all processors in this dimension are mapped to the same node of a node arrangement, and thus only shared-memory parallelism may be exploited. On the other hand, if the block size of a processor mapping is one, only distributed-memory parallelism will be exploited across the corresponding processor array dimension. In all other cases, distributed- and shared-memory parallelism may be combined. In Example (a) in Figure 2-1, both types of parallelism may be exploited for array `a`. In Example (b), shared memory parallelism may be exploited across the first dimension of array `b`, whereas across the second dimension both types of parallelism may be exploited.

```
!HPF$ PROCESSORS p(NUMBER_OF_PROCESSORS())
!HPF$ NODES n(NUMBER_OF_NODES())                    ! node arrangement

!HPF$ DISTRIBUTE p(BLOCK) ONTO n                    ! processor mapping

      DOUBLE PRECISION, DIMENSION(m) :: a, b, x, y
!HPF$ DISTRIBUTE(BLOCK) ONTO p       :: a, b, x, y  ! original HPF distribution
      ...

!HPF$ INDEPENDENT, ON HOME(a(y(i))), REUSE          ! INDEPENDENT loop, schedule reuse
      DO i = 1, m
          a(y(i)) = ... b(x(i)) ...
      END DO
```

Figure 2-2: HPF+ directives for clusters of SMPs

An HPF data mapping directive, e.g., `DISTRIBUTE a(BLOCK) ONTO p`, determines for each processor `p(i)` of the processor array `p` those parts of the data array `a` that are owned by `p(i)`. If, in addition, a processor mapping for `p` with respect to a node array `n` is specified, an inter-node data mapping may be automatically derived according to the following rule. The part of the array which is owned by the compute node is allocated as a whole in its shared memory, while data is distributed across the local memory of compute nodes. Also an intra-node data

mapping may be derived from a data mapping and a processor mapping. Intra-node data mappings assign portions of the data allocated at a compute node of a cluster to the processors of the compute node. They control shared-memory parallelization within compute nodes, while inter-node data mappings control distributed-memory parallelization across the compute nodes of a cluster.

2.3.3 Execution Model for Clusters of SMPs

```
      IF (first) THEN
          < inspector phase >
      ENDIF

      < MPI gather communication >                  ! inter-node communication

!$OMP PARALLEL DO                                   ! OpenMP parallelization
      DO i = 1, niters(me)
          a(loc_y(i)) = ... b(loc_x(i)) ...
      END DO
```

Figure 2-3: Code generated by VFC under the cluster execution model

The VFC compiler exploits shared memory parallelism through inserting OpenMP directives, e.g., `PARALLEL DO`. Figure 2-3 shows the code resulting from the program in Figure 2-2. The code generated by the VFC compiler is compiled with an OpenMP Fortran compiler and linked with the MPI library. The resulting binary is executed as an MPI program, i.e., as a set of parallel processes, each of which runs, within its own local address space, on a separate compute node of a cluster. Each process employs a set of concurrent threads running in the shared address space of the compute node. Data partitioning and message-passing takes place among the compute nodes of a cluster, while within a compute node additional parallelism is exploited by means of multiple threads.

2.4 Fortran 95 Extensions for Path Constructs

Pricing algorithms on lattices (see Section 3.1) typically are composed of building blocks which operate on sets of either nodes or paths. In the conventional programming style, the use of

multi-dimensional arrays results in nested loops in which variables are accessed via index tuples, including indirect accesses, whereby the readability of the resulting program is impaired. Abstractions at a higher level, in particular including the notion of paths, allow for a more natural and concise formulation. The notion proposed in this section is also capable of expressing parallelism and serves as a means in the high level parallel formulation of lattice based algorithms (see Section 5.3). In the following, the data types and operations of an abstract lattice are introduced. Subsequently, they are mapped onto a programming language notation based on Fortran 95.

2.4.1 Data Types

Lattice The lattice defines a two-dimensional discrete state space. Each state is associated with a time value and values of financial variables. At least one such variable exists, the short term interest rate, which is needed in discounting operations. States are represented by nodes. Arcs are labeled with transition probabilities. The time increment is constant over time and returned by getΔt(). The operation getLength() returns the number of time steps.

Node A node n is identified by the pair (time index i, value index j) and accessed via getNode(i,j). The set of all nodes at time index i is addressed by getNodes(i). The indices of the node are accessible via timeIndex(n) and valueIndex(n). The operation getRate(n) returns the short term interest rate.

Path A path π is specified by its starting node and the sequence of value indices defining the remaining nodes. The operation getProbability(π) returns the conditional probability of the path. The k-th node of a path is accessed through getNode(π,k), where $k = 0$ identifies the starting node. The operation getLength(π) returns the length of a path. A path of length 0 is a single node, and vice versa, and has a conditional probability of 1. The set of all paths of length ℓ emanating from node n is specified by getPaths(n,ℓ).

Array Nodes as well as paths can build array index domains, thus allowing for the declaration of arrays of nodes and paths. An element of array a is accessed through getElement(a,n) and getElement(a,π), respectively. The function sum(a) returns the sum of all elements of array a.

2.4.2 Operations

Concatenation The operation $\mathsf{concat}(\pi_1,\pi_2)$ returns the concatenation of two paths.

Cash-flow The operation $\mathsf{cashflow}(\pi)$ returns a (path dependent) cash-flow as a function of the interest rate values at the nodes of path π.

Discount The operation $\mathsf{discount}(c,\pi)$ returns

$$c \prod_{i=0}^{\mathsf{getLength}(\pi)} e^{-\mathsf{getRate}(\mathsf{getNode}(\pi,i))\,\mathsf{get}\Delta t()}, \tag{2.1}$$

the cash-flow c discounted on the basis of the interest rates at the nodes of path π.

Parallel Array Assignment The Fortran 95 `FORALL` construct assigns values to the elements of an array indexed by a given set [82]. It specifies one or more assignment statements, each of which is completely executed, i.e., it is executed for all array elements specified, before the execution of the next statement. Within a statement, the array elements may be assigned in arbitrary order, in particular in parallel. All expressions are evaluated before any array element is changed, and no array element is assigned more than once. The `FORALL` construct applied to arrays of nodes or paths (see Section 2.4.1) allows, in a data parallel manner, for specifying operations to be performed on many nodes or paths simultaneously.

2.4.3 Fortran 95π Notation

In the following, the primitives described are expressed as a notation in the style of Fortran 95, termed Fortran 95π. It relies on Fortran 95 semantics and presumes the extensions of a node-type, a path-type with a concatenation operator, a cash-flow function interface, a discount operator, node references via index pairs, the specification of sets of paths through starting node and length, inquiry functions for the time increment and number of time steps, and arrays of nodes and paths with the following properties. They are implicitly dynamic, i.e., allocatable, their extents need not to be specified explicitly, and they can be accessed within the `FORALL` construct.

Table 2-1 lists data types and operations of the abstract lattice, along with the corresponding Fortran 95π notation (operations for creating and naming lattices are not given).

Lattice	getLength()	`LENGTH()` intrinsic function
	getΔt()	`DELTA_T()` intrinsic function
Node `n`	node type	`NODE`
	getNode(i,j)	`(i,j)`
	getNodes(i)	`(i,:)`
	getRate(n)	`n%RATE`
Path `p`	path type	`PATH`, attribute `LENGTH`
	getProbability(π)	`p%PROBABILITY`
	getNode(π,k)	`p%NODE(k)`
	getLength(π)	`p%LENGTH`
	getPaths(n,ℓ)	`(n|l)`
Array `a`	node or path index domain	`INDEX` attribute
	getElement(a,n)	`a(n)`
	getElement(a,π)	`a(p)`
	sum(a)	`SUM(a)`
Operations	cashflow(π)	`CASHFLOW(p)` function interface
	discount(c,π)	`c.DISCOUNT.p`
	concat(π_1,π_2)	`p1 & p2`
Array assignment	for all nodes n with time index i	`FORALL (n = (i,:))`
	for all paths π with starting node n and length ℓ	`FORALL (p = (n,l))`

Table 2-1: Abstract lattice and Fortran 95π notation

2.5 OpusJava

OpusJava [91, 93] is a Java [66] component framework that provides high level means for interaction among distributed active objects in a heterogeneous and multi-lingual environment. The development of OpusJava is based upon the coordination language *Opus* [94, 96], a task parallel extension to HPF [69] 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.

The OpusJava distributed object model introduces a class called *ShareD Abstraction (SDA)* which allows to instantiate, manipulate, migrate, and terminate remote objects. In contrast to Java's remote object model RMI [147], 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 of an SDA are invoked.

The OpusJava framework provides for mutual exclusive execution of SDA methods, thus ensuring a consistent state of the SDA data. Arguments to Java 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. 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 to the user, OpusJava provides high level synchronization by means of condition clauses and events. It also ensures a consistent state of a remote object by imposing mutual exclusive method executions. By providing all of 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. Since any OpusJava SDA represents a separate thread of control, creating multiple SDAs on an SMP allows also for the exploitation of shared memory thread parallelism.

The OpusJava framework has been implemented as a pure Java library without any modifications to the language and the Java Virtual Machine [92]. OpusJava introduces four new classes visible to the user. Class `SDA` is a stub for a remote SDA object, class `Event` is used to reference asynchronous method invocations, class `Resource` describes an SDA's resources. Class `Request` is primarily used as wrapper for requests posed to HPF SDAs.

2.6 JavaSymphony

Most Java-based systems that support portable parallel and distributed computing either require the programmer to deal with low-level details of Java, or prevent the programmer from controlling locality of data. In contrast, JavaSymphony—a class library written entirely in Java—allows to control parallelism, load balancing, and locality at a high level [50, 53].

JavaSymphony introduces the concept of *dynamic virtual distributed architectures*, which allows the programmer to define a structure of a heterogeneous network of computing resources and to control mapping, load balancing, and migration of objects and code placement. Virtual architectures consist of a set of components, each of which is associated with a level. A level-1 virtual architecture corresponds to a single compute node such as a PC, workstation or a SMP node. A level-i virtual architecture, $i \geq 2$, denotes a cluster of level-$(i-1)$ virtual architectures which among others allows to define arbitrary complex clusters, clusters of clusters, etc. Every level-$(i-1)$ virtual architecture can only be part of one unique level-i virtual architecture. Static (e.g., machine name, operating system, etc.) and dynamic parameters (e.g., system load, idle times, etc.) can be indicated when requesting a virtual architecture. Constraints defined on parameters can be checked any time during execution of a JavaSymphony application, which enables a programmer to control, for instance, load balancing. Computing resources can be dynamically added or removed to/from a virtual architecture.

In order to distribute objects onto virtual architectures, these objects are encapsulated into JavaSymphony objects. JavaSymphony objects can be created by generating instances of a class *JSObject*, which is part of the JavaSymphony class library. JavaSymphony objects can be explicitly mapped to specific level-1 virtual architectures. Constraints can be provided to control the mapping of JavaSymphony objects to virtual architectures. A JavaSymphony object can be created single- or multi-threaded. A single-threaded object has one thread associated with it that executes all of its methods, whereas a multi-threaded object can be assigned multiple threads by the Java runtime system that execute its methods simultaneously.

Remote method invocation is the main mechanism to exchange data among distributed objects and to process work by remote objects. Besides synchronous and one-sided method invocations, JavaSymphony also offers asynchronous method invocations, which enable the user to continue the execution and retrieve the results at a later time.

2.7 Distributed Active Tree

Problems which are defined on trees arise in various application areas. Solution methods operate on the tree and solve subproblems assigned to the tree nodes. For this purpose, the nodes need to interact with each other. By repeatedly solving and updating their subproblems, all nodes contribute to the overall solution. Tree structured algorithms of this kind can be implemented in parallel by assigning subsets of tree nodes to physical processors. In order to support the high level parallel implementation of a whole class of tree structured algorithms, a parallel programming model based on the object-oriented paradigm has been designed [112]. In this model, the algorithm is expressed at a high level with transparent remote accesses, communication and synchronization. The model has been—in the form of an implementation in Java—successfully employed in the parallelization of the nested Benders decomposition algorithm (see Section 5.4). In the following, the distributed active tree model as well as its implementation in Java are described.

2.7.1 Programming Model

The *distributed active tree (DAT)* is a distributed data structure which executes a parallel program. The tree comprises active node objects, i.e., every tree node possesses a separate thread of control. Whereas in principle the tree nodes could execute different programs, in this model they execute the same code, eventually parameterized with particular node properties. As the nodes operate on different data, the programming model is of the type Single-Program-Multiple-Data (SPMD). The set of tree nodes is distributed onto a set of compute nodes. The nodes altogether execute the parallel algorithm, thus the whole tree can be viewed as a distributed processing unit, or an abstract machine, respectively. Tree nodes are mapped onto compute nodes individually or at the level of subtrees. With emphasis on iterative tree structured algorithms, the node activity is modeled as a while-loop (which can contain additional, nested, loops). Conceptually, all nodes are simultaneously active. They can exchange data with other nodes. Due to synchronization requirements, a node can have a waiting state.

The distributed active tree model distinguishes two layers. The algorithm to be executed is implemented in the *algorithm layer*, on top of the *coordination layer*, which provides services

for the communication among tree nodes. The core abstraction of the coordination layer is the *accessible*, representing a set of tree nodes which participate in a communication operation, for example, the successors and the predecessor of a node. Incoming data is buffered at the target nodes. For example, in Figure 2-4, node n_1 sends data to its successors, which are combined in the accessible $\{n_2, n_3\}$.

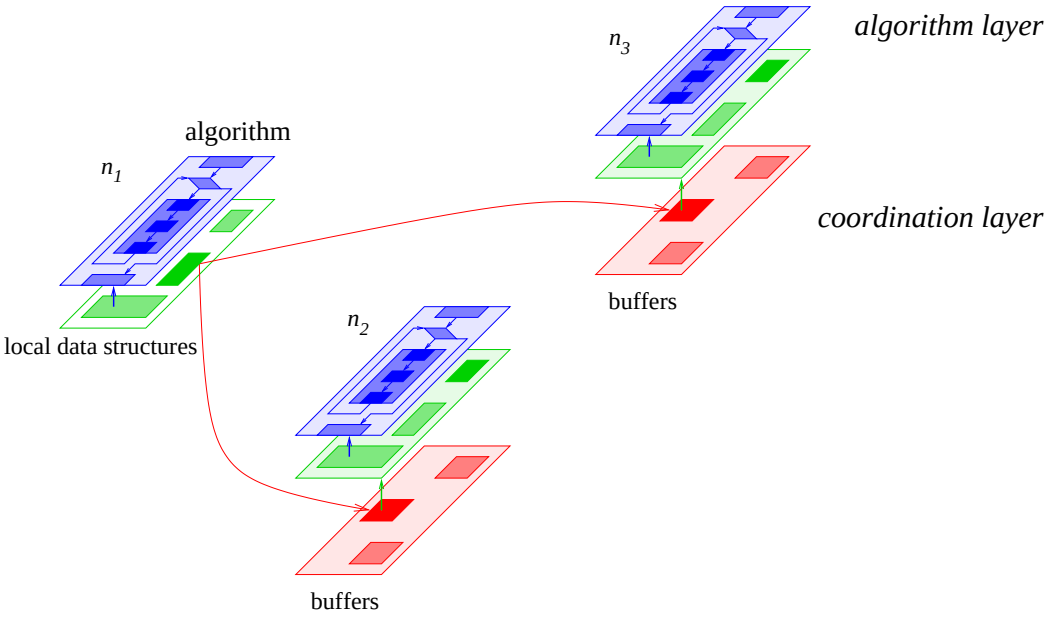


Figure 2-4: Layers in the distributed active tree model

Data transfers from or to a set of nodes are performed by the following methods of the class Accessible. The targetAccessible.put(data) method, called in the algorithm layer, copies a data object into buffers located at the elements of the target accessible. The sourceAccessible.get(condition) method, called at the target nodes, retrieves the data (put by the source nodes) from the buffer and copies it into local data structures read later on by the algorithm. A condition object specifies synchronization constraints. The standard synchronization patterns *any* and *all* specify an asynchronous mode, in which get() waits until data has arrived from at least one source node, and a full synchronous mode, in which it waits for incoming data from all source nodes (independent of the order).

These methods represent collective communication operations, which are transparent with respect to the tree distribution. Nodes residing on the same compute node, called *local nodes*,

are addressed in the same way as nodes residing at a different compute node, referred to as *remote nodes*, thus a shared memory view of the whole tree is maintained.

The simultaneous execution of many node activities on a compute node can be scheduled following different strategies. A *scheduling strategy* maps all the iterations to be executed by the tree nodes residing at a certain compute node onto threads of the run time system. For example, following the *single thread strategy*, one thread cyclically executes the first iteration on all tree nodes, then the second iteration, and so on. Alternatively, the *node thread strategy* associates every tree node with a thread, thus delegating the scheduling to the underlying runtime system. Combinations of these strategies are possible, e.g., each of a set of m threads is responsible for a set of n tree nodes.

2.7.2 Java Implementation

The distributed active tree has been implemented in Java, because of Java's portability and other features such as the object-oriented programming paradigm, robustness, automatic memory management and support for multithreading and network programming. In particular, financial applications are assumed to be used in a heterogeneous computing environment. In Java, the distributed active tree model is described as a set of interfaces, organized in the packages

dat.tree for the specification of the tree structure,

dat.alg for the specification of the node algorithm,

dat.dist for the specification of the tree distribution,

dat.sched for the specification of the scheduling strategy, and

dat.coord specifying the interface between the coordination layer and the algorithm layer.

An instance of a distributed active tree is defined by the classes implementing these interfaces. A tree node is an instance of a class which implements the `dat.alg.AlgorithmNode` interface and overrides the methods `iteration()` and `terminate()` of the `dat.alg.LoopActivity` interface, thus specifying the body and termination condition of the main loop. Associated

to every tree node there is a coordination object, an instance of a class implementing the `dat.coord.Coordination` interface, which maintains the buffer for incoming data and provides accessible objects to the `AlgorithmNode` objects. Parameters of the tree structure and its mapping onto compute nodes are specified using Java code, as an implementation of the `dat.dist.DistributedTreeSpecification` interface. The loop scheduling strategy is specified in an implementation of the `dat.sched.LoopScheduler` interface which operationally defines the mapping from the set of iterations onto a set of threads. An instance of the scheduling class creates threads, assigns loop iterations and starts them. A distributed active tree at runtime comprises, at each compute node, the objects listed in Table 2-2, which are combined by the main program at a compute node within a `ComputeNode` object (see Figure 2-5).

`AlgorithmNode`	active objects executing the node algorithm
`Coordination`	their corresponding coordination objects
`DistributedTreeSpecification`	a specification object for the tree and its distribution
`LoopScheduler`	an object specifying the scheduling strategy

Table 2-2: Objects of the distributed active tree at a compute node

```
new ComputeNode(
    "<AlgorithmNode>",
    "<Coordination>",
    new <DistributedTreeSpecification>(...)
    ).run(new <LoopScheduler>(...));
```

Figure 2-5: Running the compute node program

2.7.2.1 Coordination using RMI

The implementation of the coordination layer deals with all the intra- and inter-processor communication and -synchronization. It will employ underlying mechanisms such as Sockets, Java RMI, CORBA, and MPI. In the following, an implementation employing Java remote method invocation (RMI) is described.

The communication scheme makes use of buffers for incoming data, thus the data structures of the algorithm layer are not affected by arrivals of new data, which can occur at any point

in time. The buffers are part of the underlying coordination layer. The coordination class, implementing the `dat.coord.Coordination` interface, has a `synchronized` method `add()` which appends new data to the input buffer. It is called by the `put()` operation. Local nodes are directly accessed through shared memory, via local invocation of `add()` at the target coordination objects. Remote nodes are accessed through calling, via RMI, a remote method of the target `ComputeNode` object, which then locally calls the `add()` method at the target coordination objects. In the following, the implementation of the *any* and *all* synchronization patterns for the node thread scheduling strategy as a collection of state dependent actions [99], using Java's built-in synchronization features, is described.

With regard to coordination, the logical state space of a node (more precisely, of the coordination object) consists of the following states. In the Active state, the node is actually performing computations of the node algorithm. When it waits for data from other nodes, it enters the Wait state. When new data has arrived, it enters the Transfer state. In this state, the buffer contents are copied into local data structures. States are left either when the activity associated with a state is completed, or when `add()` has been called, i.e., some node has added data to the buffer. Accesses to the buffers are `synchronized` in order to prevent the modification of a buffer while its contents are transfered to the local data structures.

The UML [70] statechart diagram in Figure 2-6 shows the additional substates New, which are entered when new data has arrived and which allow for some optimizations. In the case of Active, Wait can be skipped, and in the case of Transfer, the new data will just be considered during the `get()` operation currently executed. In addition, two boolean guard conditions control the state transitions. First, incoming data is regarded relevant only if the node, which has put it into the buffer, is an element of the Acessible executing `get()`. Second, in case the *all* synchronization pattern is applied, data is regarded available only if arrived from all element nodes of the accessible. If the *any* pattern is applied, the all condition always evaluates to true.

2.7.2.2 Synchronization

The statechart diagram Figure 2-6, specifying the synchronization behavior, is implemented by a `synchronized` method `nextState()` of the coordination object, which updates an instance variable `state`. This method is called in `get()` and, by the source node thread, on completing

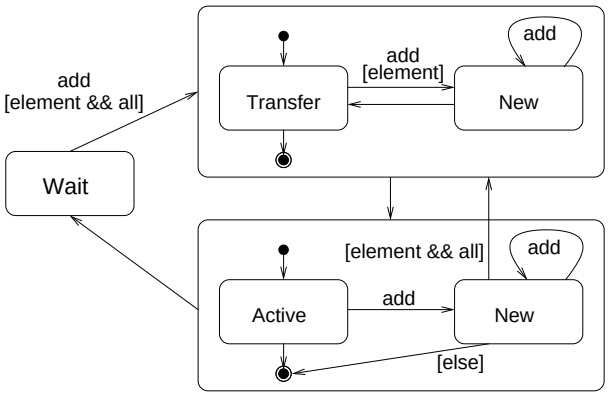


Figure 2-6: States of the coordination object

`add()`. The two cases are distinguished by the parameter, the invoking accessible in the first case as opposed to the source coordination object in the latter one. The code excerpt in Figure 2-7 shows the implementation of the `get()` method of an inner Accessible class (an implementation of the interface `dat.coord.Accessible`). If the state has changed to Wait, the node thread executes the `Object.wait()` method. The `Object.notify()` method is called in `nextState()`, just in the case that the Wait state has been left. There is no more than one thread waiting, which allows for safely using `notify()` instead of `notifyAll()`.

The `synchronized` method `transfer()` of the coordination object copies the buffered data objects into the local data structures of the `AlgorithmNode` object. As these data structures are part of the algorithm layer, they are not visible to the coordination object. Therefore the buffer contents are converted into `AlgorithmData` objects by a factory object created by the `AlgorithmNode` object. Subsequently, the methods of these objects are called, which perform the copy operation.

2.7.2.3 Optimized Handling of Data Transfers

In the following, the communication scheme is described in more detail. In a basic version, the computation would proceed, after data in the buffers has been transfered to the local data structures. Data received during the accomplishment of the transfer operation would not be transfered and thus not considered in the computation that follows. Data ignored in this way potentially enforces additional iterations to be performed by the algorithm.

```
public class Coordination implements dat.coord.Coordination {
    ...

    public synchronized void add(..., Coordination source) {
      // add data to buffer
      ...

      // state transitions due to new data arrived
      nextState(source);
      }

    public class Accessible implements dat.coord.Accessible {
        ...
        public void get(dat.coord.Pattern condition) {
            setPattern(condition);
            // assertion(state == Active || state == NewActive )

            // state transitions due to completed node activity
            nextState(this);

            synchronized(Coordination.this) {
                if (state == Wait)
                    try { Coordination.this.wait();
                        } catch (InterruptedException e) {...}
                    }

            do {
                transfer(this);    // copy buffer to local data structures
                // assertion(state == Transfer || state == NewTransfer)

                // state transitions due to completed data transfer
                nextState(this);
                } while (state == Transfer);

            }
        ...
        } // Accessible
    }
```

Figure 2-7: Synchronization in the coordination object

The semantics of the "transference" (upper) superstate in Figure 2-6 is "a node is in the transference state iff data is transfered". It comprises the substates Transfer and New, which represent the two possibilities of staying in the transference state. Either data has arrived during the execution of the current transfer operation (New), or not (Transfer). Hence New represents the information of an additional transfer operation, implemented by `transfer()`, needed after completion of the current one. When executing a transfer operation, a node can either stay in Transfer and on completion leave the transference state, or change to New, in which case it is forced to return to Transfer. Entering Transfer is equivalent to starting the transfer operation. Staying in the transference state is equivalent to the sequence of states described by the regular expression Transfer(New Transfer)*. In order to prevent very long sequences, an upper limit for the number of transitions from Transfer to New is defined.

The behavior specified in this way can be directly mapped to Java code (see Figure 2-7). Note that the call to `transfer()` with the subsequent call to `nextState()` must not be within a `synchronized` block, as this would inhibit state changes due to the arrival of new data during the execution of `transfer()`.

There are situations in which the resulting performance still is not optimal. It can happen, that, after data d_1 has arrived and caused a transition from Wait to Transfer, data d_2 arrives immediately before the call to `transfer()`, and thus New is entered, and no more data arrives during this call. Both d_1 and d_2 will be properly transfered, indeed, a second call to `transfer()` will follow, although the buffers are empty. This problem can be easily tackled by checking the buffers to be non-empty on entering `transfer()`. A different situation arises if data d arrives during the execution of `nextState()` (after the call to `transfer()`), which switches to Active. This means that no data has arrived during the execution of `transfer()`. Conceptually, the node is already active, and the data will *not* be transfered and used in subsequent computations. However, the time spent in `nextState()` is very short. The problem can be tackled by including `NewActive` in the termination condition of the do-while-loop.

Chapter 3

Computational Problems in Finance

This work deals with the application of high performance computing for solving problems in pricing and optimization. Though in principle these problem classes are independent and as such constitute important areas in computational finance, they arise combined in a real application such as a financial management system. The AURORA Financial Management System is a decision support tool for portfolio and asset liability management under development at the University of Vienna [129]. Parts of the system are being developed in cooperation with a pension fund company [131, 132]. The following overview of the system is given by the authors in [95].

The classical 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 is maximized, subject to uncertainty of future markets' development 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, as shown in Figure 3-1), and a precise statement of a coherent objective function (in the sense of [126]). 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.

The AURORA ALM model is based on a pricing model for financial instruments on the asset side, and a multivariate Markovian birth-and-death model for liabilities. The core of the AURORA Financial Management System is a large scale linear or convex optimization problem,

solution of which is the ultimate aim of the modeling effort. Figure 3-1 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) while the liability scenarios must adhere to the accounting rules specific to the pension funds. 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, an optimization problem is formed (3) and solved (4).

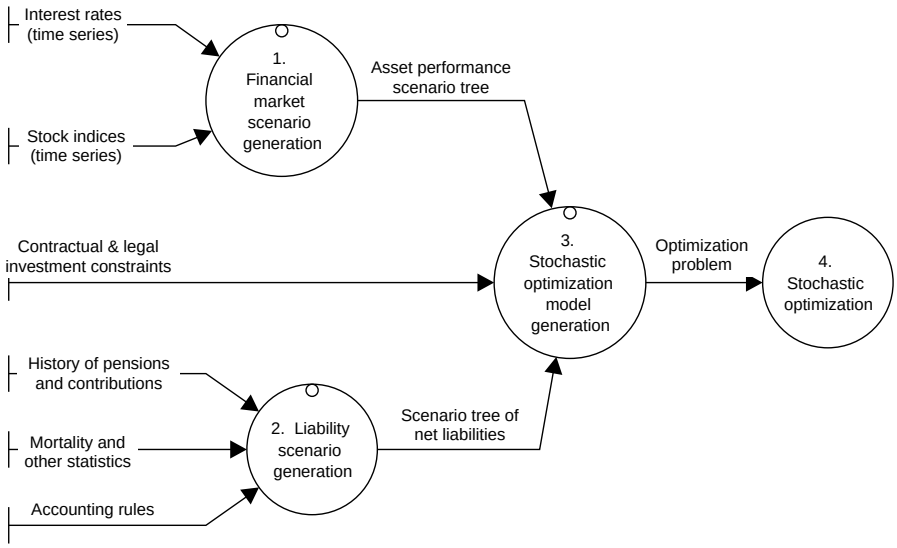


Figure 3-1: The AURORA Financial Management System.

3.1 Pricing of Interest Rate Derivatives

Since an investor in a financial management problem bases his decision on the prices for different instruments, a financial management system includes a pricing model. It forms an important part of financial market scenario generation and serves to determine the price of a financial

instrument at a specific time and in a specific state of the economic environment. For interest rate dependent instruments in particular, the pricing problem can be formulated as follows: what is the price today of an instrument with future payments depending on the development of interest rates ?

According to the arbitrage free pricing principle, which states that two financial contracts that deliver identical future cash-flows must have the same price today, the price is the present value of future cash-flows. The present value equals the expected, discounted cash-flow of the instrument [75]. Analytical pricing formulas are available for simple cases, but in general numerical techniques have to be applied. Estimates of future interest rates needed both for the calculation of cash-flows and for discounting have to be supplied by an interest rate model. In the following, the interest rate instruments dealt with, as well as the problem of pricing these, and the interest rate model employed are described.

3.1.1 Interest Rate Dependent Instruments

A financial instrument is characterized by a stream c of cash-flows c_{t_i} which are paid at payment dates t_i, $1 \leq i \leq n$,

$$c = (c_{t_1}, \ldots, c_{t_n}), \tag{3.1}$$

where $n > 0$ denotes the number of payment dates. The present value PV_t at time $t < t_i$ is defined as the expected value of discounted future cash-flows,

$$PV_t = \mathbb{E}_t \left[\sum_{i=1,n} c_{t_i} \exp(-\int_0^{t_i} r_\tau d\tau) \right], \tag{3.2}$$

where $\mathbb{E}_t$ is the expectation at time t and r_τ is the instantaneous short rate at time τ, used for continuous discounting. The price of an instrument at valuation date t_0 is given by PV_{t_0}.

In case of fixed income securities the cash-flows are deterministic. Alternatively, for floating rate securities ("floaters") such as variable coupon bonds, they are specified in terms of a reference interest rate, e.g. the LIBOR (London Inter-Bank Offered Rate).

Path dependent instruments are instruments paying cash-flows which depend on the value of a financial variable in the past. In the case of interest rate dependent instruments, the

cash-flows c_{t_i} depend on past levels of a reference interest rate R

$$c_{t_i}(\{R_{\tau_{i,1}}, \ldots, R_{\tau_{i,m_i}}\}), \tag{3.3}$$

where $m_i > 0$ denotes a number of interest rate realizations $R_{\tau_{i,j}}$, $1 \leq j \leq m_i$, and $\tau_{i,j} < t_i$. Figure 3-2 shows a *lookback cap* instrument with its coupon payment at time t specified as

$$c_t = t_d \max\left(\max_{\tau=t-k\Delta t \,\wedge\, k\in\mathbb{N} \,\wedge\, \tau\geq t-t_d} L_\tau - \rho,\ 0\right), \tag{3.4}$$

where L_τ is the level of LIBOR at times τ, and ρ is a constant cap rate. Here, the cash-flow depends on the path of reference interest rates from time $t - t_d$ to time t.

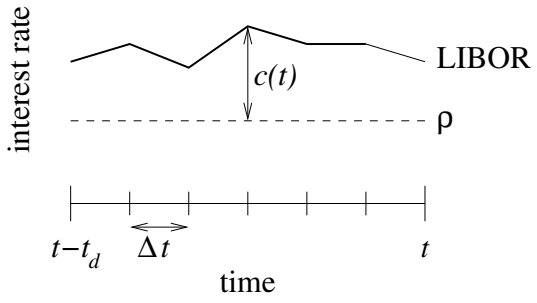


Figure 3-2: Lookback cap instrument

3.1.2 Constant Maturity Floaters

For so called constant maturity instruments, the maturity of the reference interest rate exceeds the period of adjustment. Constant maturity instruments, based in particular on the so called secondary market yield (SMY), are very popular among Austrian banks, e.g., loans, credits, and bonds that use the SMY as a reference interest rate. The SMY is an index of Austrian government bond yields traded in the secondary market and can be interpreted as an average yield to maturity in the range of five to seven years. Consequently, instruments based on the SMY are constant maturity floaters. Some instruments provide an embedded option that allows for redeeming, and thus payment of the nominal value, before time of maturity. Such an option

is typically exercised by the issuer in case the discounted stream of remaining coupon payments is expected to be greater than the redemption price.

In the following, two types of instruments with embedded options are considered. These are either bonds with a coupon rate equal to the SMY which include a call option by the issuer, or SMY floaters with variable caps and floors and the following characteristics. An initial interest rate r_0 applies as long as the SMY at time t does not hit a lower or an upper bound specified symmetrically around a basis rate r_b, i.e.,

$$\mid \mathrm{SMY}_t - r_b \mid \leq \rho. \tag{3.5}$$

If on the contrary $\mid \mathrm{SMY}_t - r_b \mid > \rho$, the interest rate is adjusted to a new level of

$$r_{t+\Delta t} = r_t + f(\mathrm{SMY}_t - r_b), \tag{3.6}$$

where $0 < f < 1$ is the factor of adjustment. In addition, the basis rate is updated to $r_b = \mathrm{SMY}_t$. As the adjustment depends on past interest rate realizations, these products are path dependent. They are priced under the assumption that the SMY can be specified as a linear combination of spot rates, based on the replication of the SMY introduced in [133],

$$\Delta\mathrm{SMY}_t = \gamma_0 + \gamma_1 \Delta R(t + T_1) + \gamma_2 \Delta R(t + T_2), \tag{3.7}$$

where T_i are the maturities of spot rates and γ_i are constants.

3.1.3 Specification of Path Dependent Instruments

A general implementation of a pricing model requires to be parameterized with respect to the products to be priced. For the specification of interest rate dependent products, a specification approach has been chosen which is based on an internal state. Using this state, information needed for the calculation of the path dependent cash-flow at time t is accumulated in a recursive manner in time steps Δt over the period $t - t_d$ to t.

The definition of the cash-flow c_t at time t is a function of the current state s_t and time independent parameters and does not refer to past interest rate values. A state transition at

time $\tau + \Delta t$, $t - t_d \leq \tau \leq t - \Delta t$, is specified as a function of state s_τ and interest rate level r_τ. The following examples represent specifications of exotic interest rate derivatives.

Lookback cap $s \in \mathbb{R}$ (see Figure 3-2),

Initialization $s_{t-t_d} = -\infty$,

State transition $s_{\tau+\Delta t} = \max(s_\tau, r_\tau)$,

Cash-flow $c_t = t_d \max(s_t - \rho, 0)$.

Down and out barrier cap $s \in \{\text{false}, \text{true}\} \times \mathbb{R}$,

Initialization $s_{t-t_d} = (\text{true}, 0.0)$,

State transition $s_{\tau+\Delta t} = (s_\tau^{[1]} \wedge (r_\tau \geq H), r_\tau)$,

Cash-flow $c_t = t_d \max(s_t^{[2]} - \rho, 0)\mathbf{1}_{s_t^{[1]}}$.

Average rate cap $s \in \mathbb{N} \times \mathbb{R}$,

Initialization $s_{t-t_d} = (0, 0.0)$,

State transition $s_{\tau+\Delta t} = (s_\tau^{[1]} + 1, s_\tau^{[2]} + r_\tau)$,

Cash-flow $c_t = t_d \max(s_t^{[2]}/s_t^{[1]} - \rho, 0)$.

In case the state is a tuple, $s^{[i]}$ denotes its i-th component. $\mathbf{1}_{cond}$ is an indicator function yielding one if the condition *cond* holds, and zero otherwise. Both the state transition and the cash-flow functions are specified by means of expressions, thus avoiding the need for control constructs such as conditional statements and loops.

3.1.4 The Hull and White Interest Rate Model

In order to calculate the price of an interest dependent instrument, a model of interest rates is needed that allows for estimating future interest rates. Hull and White developed a single factor model, in which the factor of uncertainty is the short term interest rate [76]. The model is expressed as the stochastic differential equation

$$dr(t) = (\theta(t) - a\, r(t))\, dt + \sigma dz, \tag{3.8}$$

where $r(t)$ denotes the instantaneous short rate at time t, σ is the volatility, and dz is an increment of a standard Wiener process. The stochastic process specified is a mean reverting process with an average short rate $\theta(t)$ as a function of time, and adjustment rate a. The model can be calibrated such that it is consistent with an initial term structure.

In order to efficiently implement numerical pricing procedures, a discrete representation of the model has been chosen. The continuous time model (3.8) is approximated by a trinomial lattice, the so called Hull and White tree. It describes the future development of interest rates in discrete time, with increment Δt, and discrete interest rate value. Each node represents a state, defined by a time value and an interest rate value, which is the spot rate for maturity Δt. The lattice is evenly spaced in the interest rate dimension, with interval Δr.

Nodes are denoted by the pair (m, j), where m is a time index and j is an interest rate index. Node (m, j) represents a time value of $m\Delta t$ and an interest rate value of $\alpha_m + j\Delta r$, $0 \leq m \leq M$, where M is the number of time steps, α_m an adjustment term, and j is bounded by $j_{\min} \leq j \leq j_{\max}$. Node (m, j) has three successor nodes $(m + 1, succ_{dir}(j))$ with transition probabilities $p_{dir}(j)$, where $dir \in \{\text{up}, \text{mid}, \text{down}\}$. Both the successors and the probabilities depend on the interest rate index j only. Figure 3-3 shows an example of a Hull and White tree with $\Delta t = 1$ year and $\Delta r = 0.0173$.

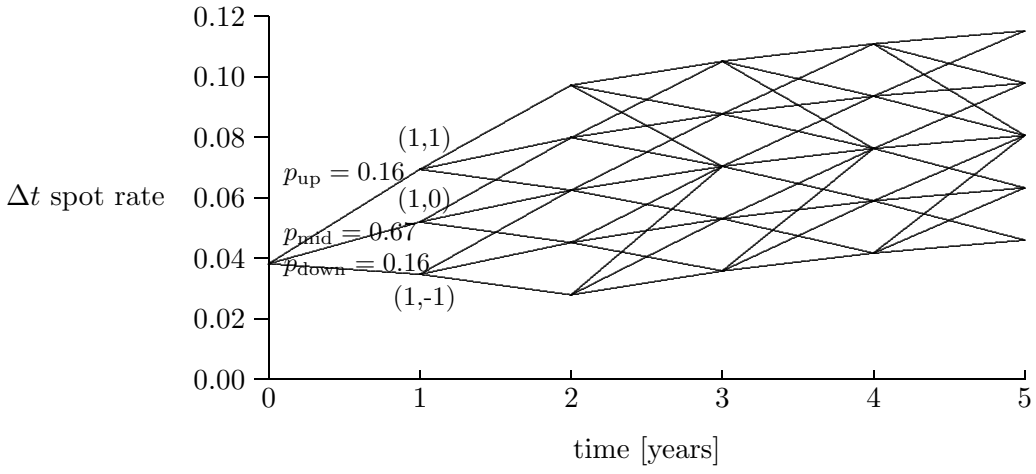


Figure 3-3: Hull and White interest rate tree

According to Hull and White, the spacing between interest rates Δr is set to $\Delta r = \sigma\sqrt{3\Delta t}$. The probabilities are specified such that they are consistent with the probability properties of the continuous model. More precisely, both the expected change in r and the variance of the change in r over the next interval Δt must be matched. In addition, $p_{\text{up}} + p_{\text{mid}} + p_{\text{down}} = 1$. These conditions allow for an initial calculation of the probabilities. In order to take into account the mean reverting property, the probabilities need to be adjusted.

Using a forward induction method, the lattice is calibrated to the initial term structure [83]. Let $Q_{m,j}$ denote the present value of a zero coupon bond maturing at time $m\Delta t$ that pays 1 if state (m, j) is reached and 0 otherwise. The $Q_{m,j}$ are also called state prices. Their values can be computed inductively, starting with $Q_{0,0} = 1$, according to

$$Q_{m+1,j} = \sum_k Q_{m,k} q(k, j) \exp\left(-r_{m,k}\Delta t\right), \tag{3.9}$$

where $q(k, j)$ denotes the probability of a transition from node (m, k) to node $(m + 1, j)$, and the summation is over all k for which this is positive, i.e., for all predecessor nodes [161]. The price of a discount bond maturing at time $(m + 1)\Delta t$ can be expressed as

$$P_{m+1} = \sum_j Q_{m,j} \exp\left(-r_{m,j}\Delta t\right). \tag{3.10}$$

The tree construction procedure takes as an input the number time steps M, the length of a time step (in years) Δt, the model parameters σ and a, and the initial term structure r_{init}. In a first phase, the interest rate interval Δr, the tree height, defined by $j_{\min}$ and $j_{\max}$, and the successor probabilities for each j are calculated. A preliminary version of the tree is defined by assigning the rate $r'_{m,j} = j\Delta r$ to each node. In a second phase, the preliminary tree is adjusted to the initial term structure by computing the Δt spot rates $r_{m,j} = \alpha_m + r'_{m,j}$. The values α_m are computed iteratively, starting with $\alpha_0 = r_{\text{init}}(\Delta t)$, the current Δt spot rate. The calculation of α_m uses the vector Q_m, and the calculation of Q_m uses the value α_{m-1} of the previous step. In the final phase, the instantaneous short rate is calculated from the Δt spot rate at each node, and from the latter an entire term structure is derived. Based on the interest rate lattice, numerical pricing algorithms can be applied [47, 113].

3.2 Multistage Stochastic Portfolio Optimization

Problems in which decisions are made in stages occur in different areas, for example in resource acquisition, reservoir optimization, and in asset and liability management. In fact, they represent an important class of the problems dealt with in finance. Finding optimal investment strategies is important for long term investors, especially those who wish to achieve goals and meet future obligations.

Portfolio management is a classical problem in financial optimization. An investor chooses a portfolio of various assets, in such a way that some objective, including a risk measure, is maximized, subject to uncertainty of future markets' development and additional constraints such as budget restrictions. Objectives of a portfolio optimization problem include the maximization of an investors return, cash-flow, wealth, or some utility function. The result is a decision on how to spend a budget on different investments, or investment categories, respectively. If future asset returns are not known, a decision problem under uncertainty must be solved, which can be formulated as a stochastic programming model. If, in addition, the portfolio is rebalanced at several dates within a planning horizon, i.e. decisions are taken in stages, there is a dynamic multistage stochastic program [117].

The uncertainty of future asset returns is modeled by means of a tree structured representation of future scenarios for asset prices, e.g., stock prices, along with their realization probabilities. The investor has to plan all present and future activities conditional on the realized scenario. Figure 3-4 shows a two-stage problem for three assets, based on three future scenarios, for increasing, decreasing, and constant remaining asset prices. Circle sizes correspond to portfolio values. The optimal decisions at the second stage depend on the asset prices in that period and on the decision taken at the first stage. The objective is, e.g., the maximization of the expected value of the portfolio after the second-stage decision. It can be expressed as the sum of the portfolio values, weighted with the scenario realization probabilities at the terminal nodes.

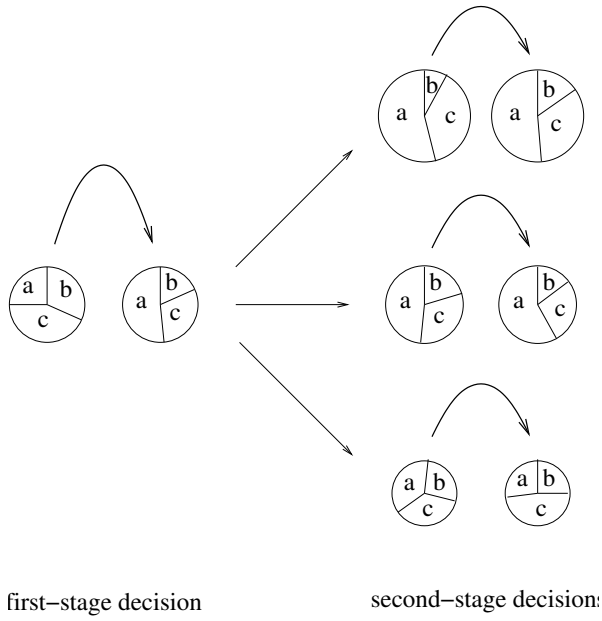


Figure 3-4: Two-stage portfolio optimization problem

3.2.1 Specification of Node Problems

Problems in which decisions are made in stages can be described by means of tree structured models. A scenario tree defines the possible developments of the environment in which the decisions are taken. Every node of the tree corresponds to a specific future scenario. At every node, based on the scenario assigned to it, a node specific objective function is formulated. The multistage stochastic optimization problem (3.11) is defined in terms of the sum of all node specific objectives.

$$\left\| \begin{array}{l} \text{Minimize} \sum_{n \in \mathcal{N}} f_n(x_n) \\ \forall (n \in \mathcal{N}) \quad \left\{ \begin{array}{l} T_n x_{pre(n)} + A_n x_n = b_n \\ x_n \in S_n, \end{array} \right. \end{array} \right. \tag{3.11}$$

where $\mathcal{N}$ denotes the set of nodes in the scenario tree. Every node $n \in \mathcal{N}$ is associated with a local objective function f_n wrt. decision variables x_n. The constraints expressed by A_n, b_n, and T_n represent the dependency of the current decision x_n on the decision $x_{pre(n)}$ taken at the previous stage, i.e., at predecessor node $pre(n)$. They are called linking constraints, the variables $x_{pre(n)}$ are called linking variables. In addition, there are constraints local to the

node, denoted by S_n. If n is the root, then $T_n = 0$. In the following, the optimization problems *maximization of terminal wealth* and *index tracking* are defined in terms of linear node problems.

A *node problem* is defined by the local objective f_n and both local and linking constraints. Its decision variables $\bar{x}_n$ combine the local decision variables x_n and the linking variables $x_{pre(n)}$. If the objective function f_n and the constraints in (3.11) are linear, a node problem can be formulated as the linear program

$$\left\| \begin{array}{l} \text{Minimize } \bar{c}_n^T \bar{x}_n \\ \text{s.t.} \\ \bar{A}_n \bar{x}_n \leq \bar{b}_n \\ \bar{x}_n \geq 0. \end{array} \right. \tag{3.12}$$

Here, the objective function is defined by the cost vector $\bar{c}_n$. Local as well as linking constraints are described by means of the constraint matrix $\bar{A}_n$ and the right hand side $\bar{b}_n$. The specification of all node problems defines the whole optimization problem. In the decomposition procedure, the node problems form the cores of the linear programs to be solved at each node (see Section 4.2.2). The objective functions and constraints specified in the following are based on the approaches of [129, 130, 154].

Let N_c denote the number of financial contracts in a portfolio, c denote the amount of cash in the portfolio at a node, h_i denote the number of pieces of contract i held, b_i denote the buy price for of one piece of contract i, s_i denote the sell price, and v_i denote its value, respectively. The value is set equal to the sell price, thus the total portfolio value V at the node is

$$V = c + \sum_{i=1}^{N_c} h_i v_i, \tag{3.13}$$

with $v_i = s_i$. The number of decision stages is denoted by T, the absolute probability of a node by p_n, and the set of tree nodes at time stage t, $0 \leq t \leq T$, is denoted by $\mathcal{N}_t$. Note that $\sum_{n \in \mathcal{N}_t} p_n = 1$. In the following, the node index n can be omitted, if the context is clear.

3.2.2 Maximization of Terminal Wealth

The objective function is

$$\text{Maximize } \mathbb{E}(V_T) = \sum_{n \in \mathcal{N}_T} p_n V_n, \tag{3.14}$$

the maximization of the terminal wealth, defined as the expected value of the portfolio value v_T at the final decision stage, where the total portfolio value at node n is denoted by v_n.

The local linear program has $3N_c+1$ decision variables. For every contract i, there is a hold variable $\bar{x}_i^{(h)}$, representing the number of pieces of the contract held in the portfolio—after the decision taken, a buy variable $\bar{x}_i^{(b)}$, representing the number of pieces being bought as a part of the decision, and a sell variable $\bar{x}_i^{(s)}$, representing the number of pieces being sold. In addition, there is a cash variable $\bar{x}^{(c)}$, representing the amount of cash in the portfolio—after the decision taken. Short selling is not permitted. Note that (3.12) allows for non-integer numbers of pieces of contracts. Both cash and hold variables are linking variables. They are part of the right hand sides of the successors' problems.

variables	cost vector	right hand side
$\bar{x}^{(c)}$	$-p$	$pre(n).\bar{x}^{(c)}$
$\bar{x}_1^{(h)}$	$-p\,v_1$	$pre(n).\bar{x}_1^{(h)}$
$\bar{x}_2^{(h)}$	$-p\,v_2$	$pre(n).\bar{x}_2^{(h)}$
$\bar{x}_3^{(h)}$	$-p\,v_3$	$pre(n).\bar{x}_3^{(h)}$
$\bar{x}_1^{(b)}$	0	0
$\bar{x}_2^{(b)}$	0	0
$\bar{x}_3^{(b)}$	0	0
$\bar{x}_1^{(s)}$	0	
$\bar{x}_2^{(s)}$	0	
$\bar{x}_3^{(s)}$	0	

Table 3-1: Terminal wealth maximization problem

The cost vector $\bar{c}$ has $3N_c+1$ elements. For all nodes other than the terminal nodes, $\bar{c}=0$. For terminal nodes, the first N_c+1 entries are the cash- and contract values, weighted with probability p. The local objective function is the minimization of the portfolio value at the node, weighted with probability p. The sum of the local objective functions is the global objective, in fact, it is the maximization of the expected portfolio value at the final stage. The formulation of the objective function as a minimization requires the entries of the cost vector to be negative.

The constraint matrix $\bar{A}$ has $3N_c+1$ columns and $2N_c+1$ rows. The first N_c+1 rows represent balance equations for cash and contract holdings, including dependencies from the previous stage problem, through the linking variables. Note that, with the terminal wealth

maximization objective, they need not to be equality constraints. The last N_c rows describe additional (local) constraints, the maximum contribution β_i of the value of contract i to the total portfolio value. They are needed to prevent portfolios consisting of a single asset only.

cash	hold			buy			sell		
1	0	0	0	b_1	b_2	b_3	$-s_1$	$-s_2$	$-s_3$
0	1	0	0	-1	0	0	1	0	0
0	0	1	0	0	-1	0	0	1	0
0	0	0	1	0	0	-1	0	0	1
$-\beta_1$	$(-\beta_1+1)v_1$	$-\beta_1 v_2$	$-\beta_1 v_3$	0	0	0	0	0	0
$-\beta_2$	$-\beta_2 v_1$	$(-\beta_2+1)v_2$	$-\beta_2 v_3$	0	0	0	0	0	0
$-\beta_3$	$-\beta_3 v_1$	$-\beta_3 v_2$	$(-\beta_3+1)v_3$	0	0	0	0	0	0

Table 3-2: Constraint matrix of terminal wealth maximization problem

The right hand side vector $\bar{b}$ has $2N_c + 1$ elements. For $t > 0$, the first $N_c + 1$ elements are the values of the linking variables (the cash- and hold values of the previous stage—after the decision taken). During the solution procedure they are retrieved from the predecessor node. At the root node, these are the initial cash- and hold values.

Table 3-1 and Table 3-2 illustrate the local linear program for $N_c = 3$. Figure 3-5 shows as an example of a terminal wealth maximization problem, defined on a binary scenario tree. For the root (node 1) and its successors (nodes 2 and 3), the constraint matrix $\bar{A}$, as well as the right hand side $\bar{b}$, and the cost vector $\bar{c}$ are given.

3.2.3 Index Tracking

The objective function is

$$\text{Minimize } \mathbb{E}(\mid V_t - I_t \mid) = \sum_{n \in \mathcal{N}_t} p_n |V_n - I_t|, \tag{3.15}$$

which is equivalent to the minimization of the mean absolute deviation of the tracking error $d_t = V_t - I_t$ between the portfolio value V_t and an index value I_t, for all time stages $t \geq 1$. The sequence $I_1, \ldots, I_T$ of index values is an external parameter to the problem. It can be derived from historical data, which allows for backtesting an index tracking model [154]. Alternatively, the index values can be calculated as a function of the prices of instruments in a set which defines the index, allowing for deriving a future investment strategy. The local linear programs

```
node 1
A = ( 1.00    0.00    0.00    0.00  122.64  183.32  206.18 -112.64 -173.32 -196.18
      0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00    0.00
      0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00
      0.00    0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00
     -0.50   56.32  -86.66  -98.09    0.00    0.00    0.00    0.00    0.00    0.00
     -0.54  -60.83   79.73 -105.94    0.00    0.00    0.00    0.00    0.00    0.00
     -0.47  -52.94  -81.46  103.97    0.00    0.00    0.00    0.00    0.00    0.00 )
b=(3000.00   10.01    9.68    9.81    0.00    0.00    0.00 )
c = ( 0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00 )


node 2
A = ( 1.00    0.00    0.00    0.00  139.66  178.38  244.03 -129.66 -168.38 -234.03
      0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00    0.00
      0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00
      0.00    0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00
     -0.50   64.83  -84.19 -117.02    0.00    0.00    0.00    0.00    0.00    0.00
     -0.54  -70.02   77.45 -126.38    0.00    0.00    0.00    0.00    0.00    0.00
     -0.47  -60.94  -79.14  124.04    0.00    0.00    0.00    0.00    0.00    0.00 )
b = ( 0.00    0.00    0.00    0.00    0.00    0.00    0.00 )
c = ( 0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00 )


node 3
A = ( 1.00    0.00    0.00    0.00  147.50  260.44  428.80 -137.50 -250.44 -418.80
      0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00    0.00
      0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00    0.00
      0.00    0.00    0.00    1.00    0.00    0.00   -1.00    0.00    0.00    1.00
     -0.50   68.75 -125.22 -209.40    0.00    0.00    0.00    0.00    0.00    0.00
     -0.54  -74.25  115.20 -226.15    0.00    0.00    0.00    0.00    0.00    0.00
     -0.47  -64.62 -117.71  221.96    0.00    0.00    0.00    0.00    0.00    0.00 )
b = ( 0.00    0.00    0.00    0.00    0.00    0.00    0.00 )
c = ( 0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00 )
```

Figure 3-5: Node problems for terminal wealth maximization

of the index tracking problem are presented in terms of differences to the terminal wealth maximization problem (see Section 3.2.2). For the root node, the local problem is identical to terminal wealth maximization.

variables	cost vector	right hand side
$\bar{x}^{(c)}$	0	$pre(n).\bar{x}^{(c)}$
$\bar{x}_1^{(h)}$	0	$pre(n).\bar{x}_1^{(h)}$
$\bar{x}_2^{(h)}$	0	$pre(n).\bar{x}_2^{(h)}$
$\bar{x}_3^{(h)}$	0	$pre(n).\bar{x}_3^{(h)}$
$\bar{x}_1^{(b)}$	0	0
$\bar{x}_2^{(b)}$	0	0
$\bar{x}_3^{(b)}$	0	0
$\bar{x}_1^{(s)}$	0	I_t
$\bar{x}_2^{(s)}$	0	
$\bar{x}_3^{(s)}$	0	
d^+	p	
d^-	p	

Table 3-3: Index Tracking problem ($N_c = 3$)

The difference $d_n = V_n - I_t$, $n \in \mathcal{N}_t$, between the portfolio value at a node and the index value is represented as $d_n = d_n^+ - d_n^-$, $d_n^+ \geq 0$ and $d_n^- \geq 0$. The vector of variables is extended by the variables $d^+$and d^-. The cost vector $\bar{c}$ refers to the newly introduced variables only. For the root node, $\bar{c} = 0$. For all other nodes, the local objective function is the minimization of $d^+ + d^-$, which is is equivalent to the minimization of $|d| = |d^+ - d^-|$. If $d^+ + d^-$ is minimal, either $d^+ = 0$ or $d^- = 0$. The sum of the local objective functions is the global objective, i.e. the minimization of

$$\sum_{1 \leq t \leq T} \sum_{n \in \mathcal{N}_t} p_n (d_n^+ + d_n^-), \tag{3.16}$$

which is equivalent to the minimization of $\sum_{n \in \mathcal{N}_t} p_n |d_n|$ for all time stages $t \geq 1$. If the total sum is minimal, then the partial sums related to the individual time stages are minimal. The constraint matrix $\bar{A}$ shows an additional row for the difference between the portfolio value and the index value

$$d = c + \sum_{i=1}^{N_c} \bar{x}_i^{(h)} v_i - I_t = d^+ - d^-, \tag{3.17}$$

which is represented by an equality constraint. Also the first $N_c + 1$ (linking) constraints are equality constraints. The right hand side vector $\bar{b}$ has an additional element for the new constraint.

cash	hold			buy			sell			d^+	d^-
1	0	0	0	b^1	b^2	b^3	$-s^1$	$-s^2$	$-s^3$	0	0
0	1	0	0	-1	0	0	1	0	0	0	0
0	0	1	0	0	-1	0	0	1	0	0	0
0	0	0	1	0	0	-1	0	0	1	0	0
$-\beta_1$	$(-\beta_1+1)v_1$	$-\beta_1 v_2$	$-\beta_1 v_3$	0	0	0	0	0	0	0	0
$-\beta_2$	$-\beta_2 v_1$	$(-\beta_2+1)v_2$	$-\beta_2 v_3$	0	0	0	0	0	0	0	0
$-\beta_3$	$-\beta_3 v_1$	$-\beta_3 v_2$	$(-\beta_3+1)v_3$	0	0	0	0	0	0	0	0
1	v_1	v_2	v_3	0	0	0	0	0	0	-1	1

Table 3-4: Constraint matrix of index tracking problem ($N_c = 3$)

3.2.4 Problem Generation

```
<!ELEMENT treeproblem (nnodes, ncontr, node+)>
<!ELEMENT nnodes      (#PCDATA)>
<!ELEMENT ncontr      (#PCDATA)>
<!ELEMENT node        (obj?, constr+)>
<!ELEMENT obj         (#PCDATA)>
<!ELEMENT constr      (lhs, rhs?)>
<!ATTLIST constr type (less|equal|greater) "less">
<!ELEMENT lhs         (#PCDATA)>
<!ELEMENT rhs         (#PCDATA)>
```

Figure 3-6: Document type definition of a node problem

A test problem generator has been developed [114], which takes as an input the number of tree nodes and the number of contracts and generates an XML file for every tree node, conforming to the document type definition shown in Figure 3-6. The constraints are described in terms of the left hand side vector, the right hand side, and the type of the constraint. Optional elements have a predefined value of zero. Figure 3-7 shows the generated root problem.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE treeproblem SYSTEM "treeproblem.dtd">

<treeproblem>
<nnodes> 7 </nnodes> <ncontr> 3 </ncontr>

<node>   <n> 1 </n>
  <obj>    0.0   0.0   0.0   0.0    0.0     0.0     0.0     0.0     0.0     0.0  </obj>
  <constr type="equal">
  <lhs>    1.0   0.0   0.0   0.0  122.64  183.32  206.18 -112.64 -173.32 -196.18 </lhs>
  <rhs> 3000.0 </rhs>
  </constr>
  <constr type="equal">
  <lhs>    0.0   1.0   0.0   0.0   -1.0     0.0     0.0     1.0     0.0     0.0  </lhs>
  <rhs>   10.01 </rhs>
  </constr>
  <constr type="equal">
  <lhs>    0.0   0.0   1.0   0.0    0.0    -1.0     0.0     0.0     1.0     0.0  </lhs>
  <rhs>    9.68 </rhs>
  </constr>
  <constr type="equal">
  <lhs>    0.0   0.0   0.0   1.0    0.0     0.0    -1.0     0.0     0.0     1.0  </lhs>
  <rhs>    9.81 </rhs>
  </constr>
  ...
</node>

...
</treeproblem>
```

Figure 3-7: XML specification of the root node problem

Chapter 4

Solution Procedures

4.1 Numerical Pricing

Analytical pricing formulas for the calculation of the present value (3.2) of interest rate dependent instruments are available for simple cases, but in general numerical techniques have to be applied, depending on the type of the instrument. For path dependent products, Monte Carlo simulation techniques are employed [22]. Instruments without path dependence can be priced by means of the more efficient backward induction method [75]. Both types of methods use an interest rate lattice, a discrete representation of the stochastic process describing the future development of interest rates (see Section 3.1.4). A numerical pricing procedure takes as an input the interest rate lattice and, as a specification of the financial instrument, the principal payment value and a function calculating the coupon payment at a node. As an output, it yields the price of the instrument as the present value at the root node.

4.1.1 Monte Carlo Simulation

If the expectation in (3.2) cannot given by a formula, one possibility of determining a numerical approximation is by employing Monte Carlo simulation. The continuous time stochastic process (3.8) is represented by a number of paths describing realizations of the process. Every path is built from samples of the random variable, based on standard normal samples drawn by a random number generation procedure. Hence there is a discretization in the time dimension, but not in the dimension of the random variable. Following an alternative approach, paths are

sampled on the basis of the interest rate lattice. The resulting discretization in the interest rate dimension allows for a more efficient sampling procedure [47].

The Monte Carlo simulation algorithm selects a number N of paths π_i, $1 \leq i \leq N$, in the Hull and White tree from the root node to some final node. Every path corresponds to a scenario of future interest rates. The present value at time 0 associated with path π_i equals

$$PV_0^{(\pi_i)} = \sum_{m=1,M} c_m^{(\pi_i)} \prod_{k=0,m-1} exp(-r_k^{(\pi_i)} \Delta t), \tag{4.1}$$

where M denotes the length of the tree, $c_m^{(\pi_i)}$ denotes the cash flow at time step m in π_i, i.e., at time $m\Delta t$, and r_k is the interest rate at time step k in π_i, respectively. The average of all path specific present values

$$\hat{PV}_0 = \frac{1}{N} \sum_{i=1,N} PV_0^{(\pi_i)}. \tag{4.2}$$

is an estimate of the price of the instrument. In order to sample from the distribution of paths as defined by the probabilities in the model, a path is constructed in the following way. Starting at the root node, for each node (m, j) a successor node $(m+1, j')$ is selected, according to the transition probabilities, such that

$$j' = \begin{cases} succ_{\text{down}}(j) & \text{if } r < p_{\text{down}}(j) \\ succ_{\text{mid}}(j) & \text{if } p_{\text{down}}(j) \leq r < 1 - p_{\text{up}}(j) \\ succ_{\text{up}}(j) & \text{otherwise,} \end{cases} \tag{4.3}$$

where r denotes a standard uniform number drawn at every time step. The algorithm recursively discounts along each path, backwards from the final node to the root node, the cash-flow generated by the instrument. For node (m, j), $0 \leq m < M$, it calculates the present value

$$PV_{m,j} = (PV_{m+1,j'} + c_{m+1,j'}) \exp(-r_{m,j} \Delta t), \tag{4.4}$$

where j' is the interest rate index of the successor node in the path, and $c_{m+1,j'}$ is the respective cash-flow. The recursion starts with $PV_{M,j''} = 0$, where j'' is the interest rate index of the final node of the path. For path dependent instruments, the cash-flow depends on interest rate values at predecessor nodes. Making use of Monte Carlo simulation techniques together

with the Hull and White interest rate tree allows for pricing a large spectrum of interest rate products. However, they are computationally very intensive. For example, modeling one year on a monthly basis results in approx. $5.3\ 10^5$ paths, and modelling four years on a weekly basis results in approx. $1.7\ 10^{99}$ paths.

In case the instrument has an embedded option, the cash-flows need to be adjusted. If premature redeeming takes place, i.e., if at a node the present value is greater than the exercise value, the principal payment value is substituted for the cashflow. Starting at the final node, the conditional substitution is applied recursively for each node of the selected path. Since also the iterative discounting process is performed backwards, the redefinition of the present value expresses the cancellation of all future payments.

The modeling of the redeeming decision at a node is further improved by considering the future cash-flow not only along the path selected, but along a representative subset of all paths emanating from the node. A nested simulation calculates an approximation of the present value by applying the same Monte Carlo Simulation algorithm at that node [47]. Thus, at the main simulation level, paths starting at the root node are selected, and at the nested simulation level, for every node in such a path, paths emanating from this node are selected in order to derive the redeeming decision. Figure 4-1 shows a scenario sampled at the main simulation level, defined by the path emphasized. In order to calculate the redeeming decision at node n, paths within the shadowed area are sampled.

4.1.2 Backward Induction

If paths are not sampled randomly, but instead, in a deterministic manner, every path of the total number of M^3 paths from the root node to some final node in the lattice is selected once, the exact (with respect to the lattice) present value PV_0 can be calculated as

$$PV_0 = \sum_{\pi_i \in \Pi_{0,0}^{(M)}} p(\pi_i)\ PV_0^{(\pi_i)}, \tag{4.5}$$

where $p(\pi_i)$ is the path's probability. The notation $\Pi_{m,n}^{(\ell)}$ is used to describe the set of all paths of length ℓ starting at node (m, n). The enumeration of all paths in (4.5) shows a computational complexity of $O(3^M)$, which is exponential in M.

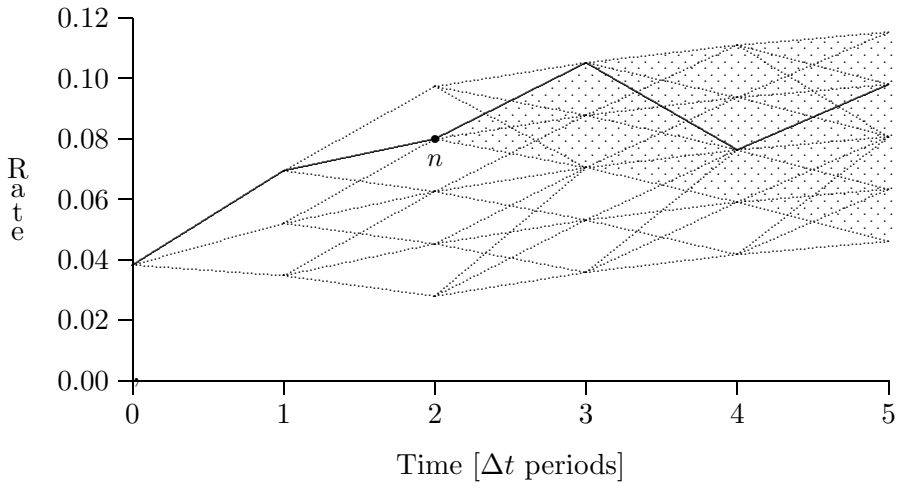


Figure 4-1: Sampled path and nested simulation

For path independent cash-flows, the backward induction algorithm computes PV_0 by recursively applying the following scheme at each node. The present value $PV_{m,j}$ at node (m, j), $0 \leq m < M$, is calculated through discounting the cash-flows and present values at every (direct) successor node,

$$PV_{m,j} = \sum_{j'} p_{j'} \, (PV_{m+1,j'} + c_{m+1,j'}) \exp(-r_{m,j}\Delta t), \tag{4.6}$$

where $j' = succ_{dir}(j)$, $p_{j'} = p_{dir}(j)$, $dir \in \{\text{up}, \text{mid}, \text{down}\}$. The backward induction starts with $PV_{M,j} = 0$, $j_{\min} \leq j \leq j_{\max}$. The algorithm computes in a backward sweep, from the final nodes to the root node, the price of the instrument as the present value $PV_0 \equiv PV_{0,0}$ at the root node. Since the present value embodies information of the future of a node which is propagated backwards in time, this method can be applied only in case the cash-flows are determined by the state represented by the node, i.e., they are not path dependent. The backward induction on a mean reverting lattice exhibits a complexity of $O(M)$, which is only linear in M.

4.1.3 Generalized Backward Induction

Searching in sorted data sets is faster than in unsorted ones. It can be performed using an improved algorithm which exploits a known property of the input data. Similarly, syntax

analysis is less costly for context free grammars which are regular. In both cases, known properties of the problem allow for a faster algorithm. The pricing of some path dependent derivatives can be performed significantly more efficient if specific characteristics of the path dependence are considered [111].

4.1.3.1 Limited Path Dependence

For a subclass of path dependent instruments, the price calculation effort can be significantly reduced, based on the following observation. A path dependent instrument exhibits a restricted form of path dependence, if the cash-flows depend on "early" values of a financial variable in the past only.

Definition 1 *The depth t_d of the path dependence of an instrument specified by cash-flows c_{t_i} (3.3) is defined as the maximum of the differences between t_i and the payment date $\tau_{i,j}$ of the earliest cash-flow on which c_{t_i} depends, i.e.,*

$$t_d = \max_{1 \le i \le n} \left(t_i - \min_{1 \le j \le m_i} \tau_{i,j} \right), \tag{4.7}$$

where $n > 0$ is the number of payments dates, and $m_i > 0$ is the number of reference interest rate realizations $R_{\tau_{i,j}}$, and $\tau_{i,j} < t_i$.

If t_d is small with respect to the life time T of an instrument, in particular, if it is not increasing with T, the path dependence is termed *limited*. This is not a definite property of the instrument, but rather serves to indicate a case in which the optimized algorithm will be profitable. In a discrete model such as the interest rate lattice, the depth can be quantified by the number d of time steps corresponding to t_d. Figure 4-2 shows an example with depth $d = 4$, i.e., the cash-flows at time step m depend on interest rates at timesteps $m - i$, $0 < i \le 4$.

4.1.3.2 Incomplete present values

A first approach to exploit limited path dependence and to allow for pricing these instruments more efficiently than, e.g., by Monte Carlo simulation, introduces path dependent present values. Through calculating the present values at the nodes of the lattice dependent on history paths of

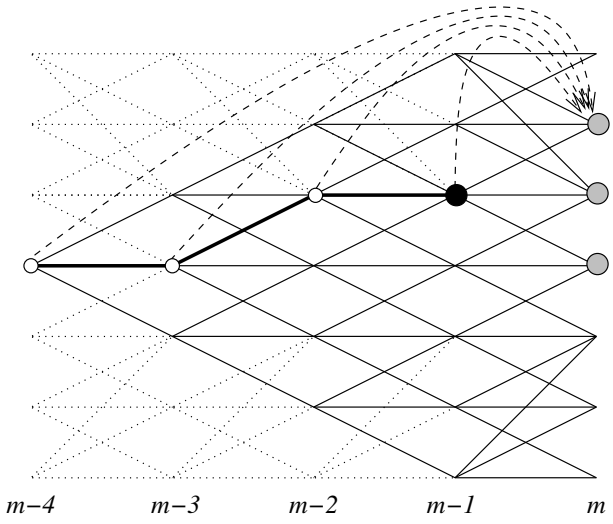


Figure 4-2: Limited path dependence ($d = 4$)

length $d-1$ (see Figure 4-2), the backward induction scheme could be maintained. However, it is necessary to calculate at every node 3^{d-1} different such values. Since they are used repeatedly, they need to be stored and retrieved, addressed in a costly way via tuples of nodes.

The core step in backward induction is the computation of the present value at a node through the discounted cash-flows and present values at the direct successor nodes. The propagation of information backwards in time reflects the present value's property of incorporating future information (i.e., future cash-flows). Through the iterative application of the induction step, the information at all successors of a node, representing its complete future, has an impact on the present value at that node. This principle works well if the future information is fully determined by the state represented by the node. In case of path dependent instruments, the future cash-flows, and thus the present value, are specific to the history of the node. An induction step must therefore regard future information only to a part which does not depend on the history of the node. The design of an induction step in accordance to this condition leads to a modified kind of present value, which in fact does not take into account cash flows in the "near" future.

Instruments exhibiting limited path dependence can be priced using a method which generalizes the backward induction in such a way that it computes the *incomplete present value* at

a node with respect to depth d, denoted by $PV^{(d)}_{m,j}$. With this modified type of present value, cash-flows paid at times earlier then $m+d$ timesteps in the future (as illustrated in Figure 4-3 by a dotted line) are ignored, i.e., the cash-flows taken into account are those paid at time steps $m+d, m+d+1, \ldots, M$. The incomplete present value contains information of the future to the part which is path independent only and thus can be propagated backwards in time. Hence, the incomplete present value itself is path independent. Note that $PV^{(1)}_{m,j} = PV_{m,j}$. An incomplete present value defined in this manner has an important property. It can be completed by taking into account the missing cash-flows. As they are path dependent, a history path of length $d-1$ is required for the completion operation.

4.1.3.3 The Generalized Backward Induction Algorithm

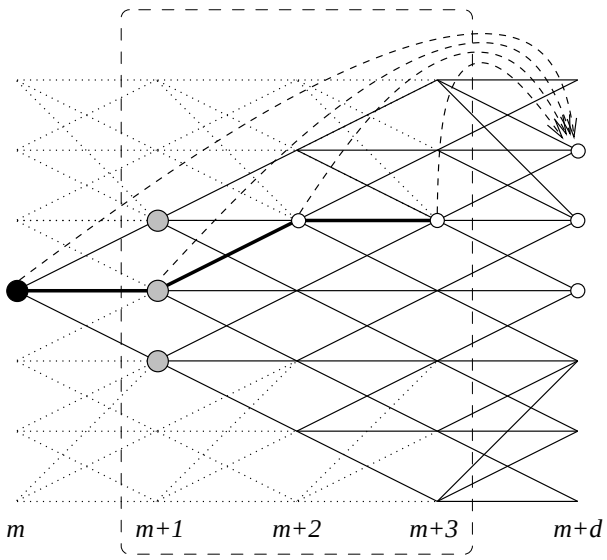


Figure 4-3: Induction step in generalized backward induction ($d = 4$)

The algorithm involves two phases [111]. The first phase is similar to the standard backward induction, however, it calculates incomplete present values at every node. Unlike (4.6), which includes the calculation of the present value at time step m of the cash-flows paid at time step $m+1$, the recursion step (4.8) in the generalized backward induction algorithm calculates the present value at time step m of the cash-flows paid at time step $m+d$, and adds them to the present value at time step m of the incomplete present values at the successor nodes $PV^{(d)}_{m+1,j'}$.

The cash-flows at time step $m + d$ are path dependent and are calculated for every path of length $d - 1$ emanating from (m, j).

$$PV_{m,j}^{(d)} = \sum_{j'} p_{j'} PV_{m+1,j'}^{(d)} \exp(-r_{m,j}\Delta t) + \sum_{\pi_i \in \Pi_{m,j}^{(d-1)}} p(\pi_i) c_{m+d}^{(\pi_i)} \prod_{k=0,d-1} exp(-r_{m+k}^{(\pi_i)}\Delta t), \quad (4.8)$$

where $j' = succ_{dir}(j)$, $p_{j'} = p_{dir}(j)$, $dir \in \{\text{up}, \text{mid}, \text{down}\}$. In the second phase, the incomplete present value at the root node is completed by adding the present value at time step 0 of the path dependent cash-flows between time step 1 and $d - 1$. The latter are calculated based on the history path $((-d+1, j'), (-d+2, j''), \ldots, (0, 0))$ concatenated with all paths in $\Pi_{0,0}^{(d-2)}$. A reordering of discounting and summation operations in (4.8) results in a deeper nesting level.

$$PV_{m,j}^{(d)} = \sum_{j'} p_{j'} \exp(-r_{m,j}\Delta t) \left[PV_{m+1,j'}^{(d)} + \sum_{\pi_i \in \Pi_{m+1,j'}^{(d-2)}} p(\pi_i) c_{m+d}^{(\pi_i)} \prod_{k=0,d-2} exp(-r_{m+k}^{(\pi_i)}\Delta t) \right]. \quad (4.9)$$

With the backward induction algorithm generalized in this manner, the exponential problem of pricing path dependent derivatives[1] is reduced to a fixed-parameter tractable problem for a particular type of path dependence. On a mean reverting lattice of length M, the price of an instrument with limited path dependence of depth d can be calculated with a complexity of $O(M\, 3^{d-1})$, which is exponential in small values of d, but still linear in the length of the lattice. The generalized algorithm includes both the path independent case $d = 1$ and the full path dependent case $d = M$. For the latter, it enumerates all paths emanating from the root node of length $M - 1$, which is equivalent to (4.5). For $d = 1$, it is equivalent to the the standard backward induction (4.6). As a modification of the algorithm, only a subset of $\Pi_{m,j}^{(d-1)}$ may be sampled in a Monte Carlo simulation manner.

[1] The method is not restricted to interest rate derivatives and applies for lattices with an arbitrary number of successors per node.

4.2 Model Decomposition

The dynamic stochastic optimization problem (3.11) can be formulated as a deterministic equivalent problem, if all objective functions and constraints are linear. The resulting large linear program can be solved by means of, e.g., the Simplex method, however, its extremely sparse constraint matrix makes an efficient solution difficult. Figure 4-4 illustrates, through displaying non-zero entries by dots, the sparsity of the 217×310 constraint matrix of the deterministic equivalent of a small terminal wealth problem of three financial contracts with four stages, where every node has two successors (see Section 3.2.2). Here, the portion of non-zero entries falls exponentially with approx. $46 \cdot 2^{-t}\,\%$, where t is the number of time stages.

Model decomposition algorithms exploit the structure of a problem. They allow for solving a set of smaller subproblems to optimize components of the problem, e.g., scenarios. The subproblems are coordinated by a *master* which repeatedly produces estimates of the overall solution. Based on the current estimate of the master, a new subproblem is defined and solved. This procedure continues in an iterative manner [5, 29, 86, 90]. For tree structured problems, the whole problem is decomposed into (much smaller) subproblems that correspond to the objective functions and constraints associated with the nodes of the tree.

4.2.1 Two-Stage Decomposition

In the simple case of a two-stage problem, the second stage decomposes into the sum of local objective functions of all successor nodes. Denoting the root node by m, the two-stage problem can be formulated as

$$\left\| \begin{array}{l} \text{Minimize } f_m(x_m) + \displaystyle\sum_{n \in succ(m)} f_n(x_n) \\ \text{s.t.} \\ T_n x_m + A_n x_n = b_n \\ x_m \in S_m \\ x_n \geq 0 \qquad \text{for all } n. \end{array} \right. \tag{4.10}$$

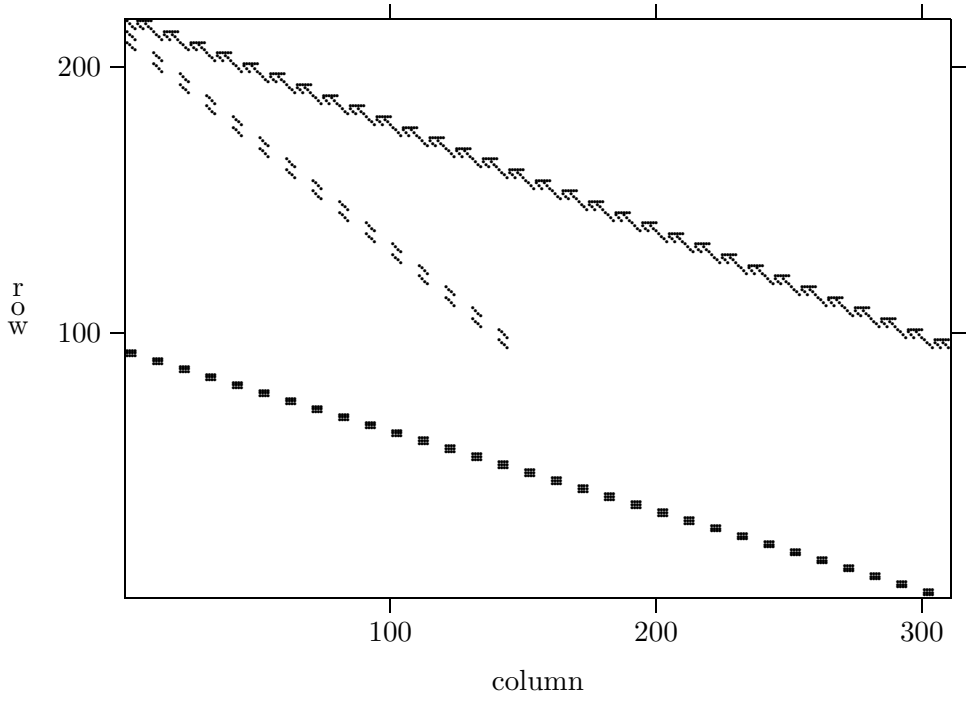


Figure 4-4: Constraint matrix of the deterministic equivalent of a 31-nodes problem

During the iterative solution procedure the master node m makes, for each successor n, use of the *value functions*

$$Q_{m,n}(x_m) = \min\{f_n(x_n)\colon T_n x_m + A_n x_n = b_n,\, x_n \geq 0\}, \tag{4.11}$$

describing the optimal solution of the problem associated with successor n as a function of the decision variable x_m at the master node. With these functions, the master node m optimizes the whole problem (4.10), whereas the *slave* nodes n solve their local problems while taking into account the master's solution. In each iteration, the master minimizes

$$f_m(x_m) + \sum_{n \in succ(m)} Q_{m,n}(x_m), \tag{4.12}$$

sends the solution to the slaves, and receives constraints from the slaves which will improve its solution in the next iteration.

4.2.2 Nested Benders Decomposition

The decomposition principle can be applied to multistage, tree structured problems [16]. In the nested version of Benders decomposition, every intermediate node of the tree acts both as a master and as a slave, whereas the root node acts only as a master, and leaf nodes only as slaves. In the following, let m denote the master node $pre(n)$ of node n. The value functions are defined recursively as

$$Q_{m,n}(x_m) = \min \left\{ f_n(x_n) + \textstyle\sum_{k \in succ(n)} Q_{n,k}(x_n) \colon T_n x_m + A_n x_n = b_n,\ x_n \geq 0 \right\}, \tag{4.13}$$

and express the dependence of the non-local part of the global objective of the master (more specifically, those parts associated with the slaves) on the value of the local variable x_m. Actually, the value functions are approximated by maxima of linear functions, which form a set of linear constraints called *optimality cuts*. In the following, one iteration step of the algorithm performed by tree node n is described for both the master and the slave role. In iteration i, node n uses the approximation $Q_{n,k}^{(i)}$ of the value function related to successor k. $R_n^{(i)}$ denotes a set of feasibility constraints called *feasibility cuts*. By sending a so called *null cut* signal a slave notifies its master that the approximation of the value function is correct.

Algorithm Nested Benders Decomposition

Step 0: Initialize $i = 0$, $R_n^{(0)}$, $Q_{n,k}^{(0)}$.

[Slave] Step 1: Node n receives the solution $x_m^{(\hat{\imath})}$, calculated by the master in iteration $\hat{\imath}$, and the current approximation value $\hat{q} = Q_{m,n}^{(\hat{\imath})}(x_m^{(\hat{\imath})})$ from the master node.

[Master] Step 1: Node n receives cuts from the slave nodes.

Step 2: Node n solves the local problem

$$\left\| \begin{array}{l} \text{Minimize } f_n(x_n) + \displaystyle\sum_{k \in succ(n)} Q_{n,k}^{(i)}(x_n) \\ \text{s.t. } T_n x_m^{(\hat{\imath})} + A_n x_n = b_n,\ x_n \in \{S_n \cap R_n^{(i)}\},\ x_n \geq 0. \end{array} \right. \tag{4.14}$$

If the problem is feasible, it yields solution x_n with objective value q_n.

[Master] Step 3: If the problem (4.14) is feasible, node n sends the solution x_n and the approximation values $Q_{n,k}^{(i)}(x_n)$ to every slave k.

[Slave] Step 3: If the problem (4.14) is feasible, node n compares the objective value q_n with the master's approximation $\hat{q}$.

[Slave] Step 3.1: If the approximation is correct, i.e., $q_n \leq \hat{q}$, the master is notified via sending a null cut signal.

[Slave] Step 3.2: Otherwise, the hyperplane $c(\xi) = \lambda_n \xi + \tau$ is calculated and sent to the master as an an optimality cut, forming an additional constraint to the master's problem. λ_n is the dual solution of (4.14) and $\tau = q_n - \lambda_n x_m^{(\hat{i})}$. This constraint will improve the master's approximation of the value function related to node n.

[Slave] Step 3.3: If the problem (4.14) is infeasible, a new constraint to the master's problem is calculated and sent as a feasibility cut to the master. By this it is ensured that in a following iteration the local problem, based on the master's solution taking this feasibility cut into account, has a feasible solution.

Step 4: Let $i \leftarrow i + 1$ and return to Step 1.

If a node has received null cuts from all slaves, it enters a sleeping state from where it resumes by receiving a new solution from its master or new (non null) cuts from its slaves. Once the root node enters the sleeping state when the rest of the tree is in the sleeping state, the optimal solution of the whole problem has been found and the algorithm terminates.

In the original version of the method, each node receives the solution from its master, builds and solves the local problem, and sends the solution to its slaves. Then it waits until it received cuts from every slave. The solution process takes the form of forward- and backward sweeps over the whole tree. It is possible to relax the tight coupling between master and slaves by allowing a node to use any new data—from its master or from its slaves, as soon as it is available—for the building of a new linear program, resulting in a more general, asynchronous version of the algorithm (see Section 5.4.2). More detailed discussions, including the calculation of feasibility cuts, can be found in [117, 128].

Chapter 5

Parallel Implementation

In the following, the parallelization of the solution procedures described in Chapter 4 is discussed in detail. The parallel implementation of numerical pricing procedures is presented in Sections 5.1–5.3. Parallel versions of nested Benders decomposition are presented in Section 5.4.

5.1 Monte Carlo Simulation

A parallel pricing kernel has been developed [47, 113] based on Fortran 90, by annotating the Fortran 90 program with HPF and HPF+ directives (see Section 2.1) and compiling it with the VFC compiler (see Section 2.2). Fortran 90 supports a modular programming style while providing a vector notation for arrays. The pricing kernel builds the lattice representation of the interest rate model and applies pricing algorithms, based on an interest rate lattice and the cash-flow specification of an instrument. It comprises several modules, encapsulating the Hull and White tree, the specification of instruments (see Section 3.1.3), basic financial operations, e.g., discounting, and both the Monte Carlo simulation and backward induction algorithms.

```
REAL(dbl)       :: rate(0:maxM,minJ:maxJ,0:maxM) ! interest rates
TYPE(direction) :: successor(minJ:maxJ)          ! successors
TYPE(direction) :: probability(minJ:maxJ)        ! probabilities
REAL(dbl)       :: factor(0:maxM,minJ:maxJ)      ! discount factors
```

Figure 5-1: Data representation of the interest rate lattice

The major data structure is the representation of the Hull and White tree, as shown in Figure 5-1. Data structures describing graphs are often based on a node data type with pointers to successor nodes. In order to save dereferencing costs during state transitions, a more efficient array representation has been chosen. Every array element represents an entire term structure. The height of the lattice is defined by `minJ:maxJ`, and `direction` is a type with components for the up-, mid-, and down direction. The element `rate(m,j,k)` holds the interest rate with maturity `k`Δt in the state (`m`,`j`). Successors and probabilities are independent of the time step. The discount factors associated with the lattice nodes are computed in the lattice building procedure and stored in the array `factor`.

```
TYPE(instrument) :: inst            ! the instrument to be priced
INTEGER          :: path(0:maxM)    ! path in the Hull and White lattice
INTEGER          :: nPaths          ! number of paths selected
REAL(dbl)        :: price(nPaths)   ! path specific prices

DO i = 1, nPaths
    path = randomPath(0, 0, maxM)   ! select a path starting at the root node

    ! discount the cashflow to time 0 using the discount factors at the path
    price(i) = discount(0, cashFlow(inst,1,maxM), factorAt(path))

END DO

priceFinal = SUM(price)/nPaths
```

Figure 5-2: Monte Carlo simulation on the interest rate lattice

The Monte Carlo simulation module computes the price of a financial instrument as the present value at the root node of the Hull and White tree (see Section 4.1.1). Figure 5-2 shows the main simulation loop. The `randomPath()` function samples a path by choosing successor nodes according to the transition probabilities. The `discount()` function discounts the sequence of cash-flows returned by `cashFlow()`, based on a vector of discount factors. The latter is returned by `factorAt()`, corresponding to the interest rates at the selected path. For embedded options, the discounting function performs a test for redeeming at every node, based on the present value of future cash-flows along the selected path. For nested simulations, at both levels the same recursive simulation procedure is called. During the simulation at the

first level, it calls an extended discount function, which calls the same simulation procedure to perform the second level simulation. At that level, redeeming can again be handled by a nested simulation at an additional, third level, or by the standard discount function, i.e., without further recursion. In fact, the nesting level can arbitrarily be chosen as a parameter to the procedure.

5.1.1 Parallelization Strategy

Interest Rate Lattice During the simulation, data at all nodes of the lattice is potentially needed by every path, thereby motivating the replication of the lattice. The storage requirement is not critical in terms of local memory size, however, according to the owner-compute rule, the replication of the lattice implies also the lattice building procedure to be replicated. On the other hand, the calculation (3.9) of the values $Q_{m,j}$ exhibits dependencies which require communication and thus limit the parallelism achievable in a parallelized lattice construction. Moreover, the distributed lattice generated needs to be totally replicated, i.e., at every compute node the local portion of the generated lattice must be sent to the other compute nodes. The resulting all-to-all communication is expensive and burdens the parallel performance. Because the frequency of price calculations is much higher than that of the construction of a new interest rate lattice, both the data structures and the construction procedure are replicated.

Monte Carlo Simulation Sampling as well discounting along paths can be performed in parallel. No communication is necessary, because the processing of a path has access to the whole (replicated) lattice, and it is independent of every other path. The final price is computed via a summation of the prices associated to the individual paths over all compute nodes. This is the only operation which requires communication. A sum reduction operation calculates partial sums on all compute nodes simultaneously and sends the partial results to a selected compute node which builds the final sum.

5.1.2 HPF+ Version

The parallelization strategy described has been expressed in HPF+. The array `price` stores the prices calculated along individual paths. It is distributed in equally sized blocks onto the

```
!HPF$ PROCESSORS P(number_of_processors())

      REAL(dbl)                  :: price(nPaths)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: price
```

Figure 5-3: Data distribution of parallel Monte Carlo simulation

available compute nodes, inquired by the HPF intrinsic function `number_of_processors()`, as shown in Figure 5-3.

In addition to distributing the data, potential parallelism needs to be specified. Figure 5-4 shows the parallel version of the main loop. HPF+ offers a number of directives which assist the compiler in parallelizing the code. In particular, the `INDEPENDENT` directive marks the main loop of the Monte Carlo simulation, indicating that the loop does not contain dependencies which might inhibit parallelization. It is annotated with further clauses that enforce the generation of efficient code. The `NEW` clause specifies that every iteration of the loop owns a private copy of variable `path`, which consequently cannot introduce any dependence. `ON HOME` applies the owner-compute rule, `RESIDENT` asserts that all data accesses are local, and the `REUSE` clause allows for the reuse of communication schedules. Finally, the summation of the prices related to individual paths in variable `priceSum` is marked as a `REDUCTION` to allow for an optimized implementation. The `discount()`, `cashFlow()`, and `factorAt()` functions, called from within the `INDEPENDENT` loop have been designed in such a way that they conform to `PUREST`.

Each compute node executes the iterations which, according to the distribution, compute the elements of the array `price` the compute node owns. The summation has been transformed from vector form to a sequential form, because a reduction operation based on the HPF `SUM` intrinsic is not supported by the version of the VFC employed. For nested simulations, the second level is not parallelized, because the maximum parallelism is already exploited at the first level. This poses a problem related to the recursive use of the simulation procedure. So far it specifies a distributed computation, which is required for the first level only. Therefore, a clone of the procedure, without distribution, performs the nested simulations. A dynamic data distribution [30], including the total replication in its distribution range, would allow for a version of the procedure which is parameterized with regard to the distribution.

```
      priceSum = 0.0d0

!HPF$ INDEPENDENT, NEW(path), ON HOME(price(i)), RESIDENT, REUSE, REDUCTION(priceSum)
      DO i = 1, nPaths
          path = randomPath(0,0,maxM)

          price(i) = discount(0, cashFlow(inst,1,maxM), factorAt(path)) ! purest call
          priceSum = priceSum + price(i)

      END DO

      priceFinal = priceSum/nPaths
```

Figure 5-4: Parallel Monte Carlo simulation

5.1.3 Experimental Results

5.1.3.1 Numerical Results

In the following, numerical results for coupon bonds with fixed or variable interest rates and embedded options are presented. The variable interest rate is specified as the SMY approximated by a linear combination of spot rates as described in Section 3.1.2. In addition, prices of SMY floaters with variable caps and floors are given. These products are path dependent and hence do require Monte Carlo simulation. Although this does not apply to the products of the first group, their prices are given for the purpose of demonstrating differences between simulated and precise values.

Tables 5-1–5-3 show prices of bonds with embedded options. The Hull and White tree is calibrated to an initial term structure $r_{\text{init}}(t) = 0.08 - 0.05e^{-0.18t}$, and the model parameters are specified as $a = 0.1$, $\sigma = 0.25$, $\Delta t = 1$ year. The SMY at time $m\Delta t$ is given by $\text{SMY}(m\Delta t) = 0.95r^{(5\Delta t)}(m-1,j) + 0.05r^{(7\Delta t)}(m-1,j)$. The theoretical price in case of a fixed coupon bond is the result of discounting along the initial term structure. In addition to the prices calculated by traversing all paths in the lattice, results of Monte Carlo simulations are reported. The sampling rates for these simulations are given as percentages of the total number of paths, which equals to 3^M. For the nested version in case of embedded options, the same percentage has been choosen at the first and the second level. For all simulations, the relative differences ϵ' (to the price calculated by traversing all paths at the first level only), and ϵ'' (to the price

	method	coupon			
		fix option		variable option	
		no	yes	no	yes
	theoretical	9332.61			
	backward	9332.61	9131.09	10537.83	10186.02
	all paths	9332.61	9081.20	10537.83	10177.30
	10%	9365.94 ϵ' 0.357	9131.38 ϵ' 0.552	10548.37 ϵ' 0.100	10183.93 ϵ' 0.065
	1%	9548.40 ϵ' 2.312	9285.55 ϵ' 2.250	10597.04 ϵ' 0.561	10193.15 ϵ' 0.155
nested	all paths		9107.67		10185.27
	10%		9097.06 ϵ''-0.116		10183.45 ϵ''-0.017
	1%		9171.73 ϵ'' 0.703		10191.01 ϵ'' 0.056
	10% once		9047.72 ϵ''-0.658		10180.59 ϵ''-0.045
	1% once		9302.12 ϵ'' 2.135		10192.16 ϵ'' 0.067

Table 5-1: Prices of 6 year bonds

	method	coupon			
		fix option		variable option	
		no	yes	no	yes
	theoretical	8883.49			
	backward	8883.49	8703.52	10540.12	10174.84
	all paths	8883.49	8592.34	10540.13	10148.45
	10%	8895.03 ϵ' 0.129	8597.73 ϵ' 0.062	10542.87 ϵ' 0.026	10148.00 ϵ'-0.004
	1%	8882.95 ϵ'-0.006	8622.23 ϵ' 0.347	10542.28 ϵ' 0.020	10152.76 ϵ' 0.042
nested	all paths		8626.26		10171.85
	10%		8625.38 ϵ''-0.010		10172.13 ϵ'' 0.002
	1%		8602.90 ϵ''-0.270		10170.63 ϵ''-0.012
	10% once		8622.27 ϵ''-0.046		10172.46 ϵ'' 0.006
	1% once		8557.21 ϵ''-0.800		10172.16 ϵ'' 0.003

Table 5-2: Prices of 8 year bonds

	method	coupon			
		fix option		variable option	
		no	yes	no	yes
	theoretical	8476.90			
	backward	8476.90	8249.79	10525.22	10164.51
	all paths	8476.90	8150.92	10525.22	10116.92
	10%	8475.38 ϵ'-0.017	8147.16 ϵ'-0.046	10524.61 ϵ'-0.006	10116.56 ϵ'-0.004
	1%	8512.37 ϵ' 0.418	8168.58 ϵ' 0.216	10534.62 ϵ' 0.089	10117.71 ϵ' 0.008
nested	all paths		8179.09		10160.34
	10%		8178.26 ϵ''-0.010		10160.27 ϵ''-0.001
	1%		8189.47 ϵ'' 0.126		10160.96 ϵ'' 0.006
	10% once		8153.83 ϵ''-0.308		10160.39 ϵ'' 0.001
	1% once		8233.72 ϵ'' 0.667		10160.74 ϵ'' 0.004

Table 5-3: Prices of 10 year bonds

calculated by traversing all paths at both levels, i.e., via nested simulation) are presented. As an optimization, it is possible to reuse the result of a nested simulation at a node, when this node is encountered later on, during subsequent simulations. In this version, at every node a nested simulation is performed only once, however, independence of the samples is not maintained.

	12 months	18 months
10%	15030.79	
1%	15025.08	
0.1%	15021.14	20531.19
0.01%		20518.46
0.001%		20514.67

Table 5-4: Prices of bonds with variable cap/floor

For the floaters with varying caps and floors, a precise result (with respect to the lattice) would require to solve for 3^{20} scenarios in case of 20 time steps, which is virtually impossible. Hence Monte Carlo simulation is the only feasible approach. Table 5-4 presents numerical results based on $r_0 = 5\%$, $r_B = 3.5\%$, $k = 3\%$, $f = 8\%$, $\Delta t = 1$ month, and maturities of 12 and 18 months. The corresponding execution times on a Sun Ultra 2 workstation are given in Table 5-5. The comparison of execution times for the different methods in Table 5-6 demonstrates, that the added effort for the nested simulation is very high. Significant improvements can be gained through the reuse of simulation results.

sampling rate	12 months	18 months
10%	15.041	
1%	1.525	
0.1%	0.195	155.977
0.01%		15.502
0.001%		1.600

Table 5-5: Execution times of Monte Carlo simulation (seconds)

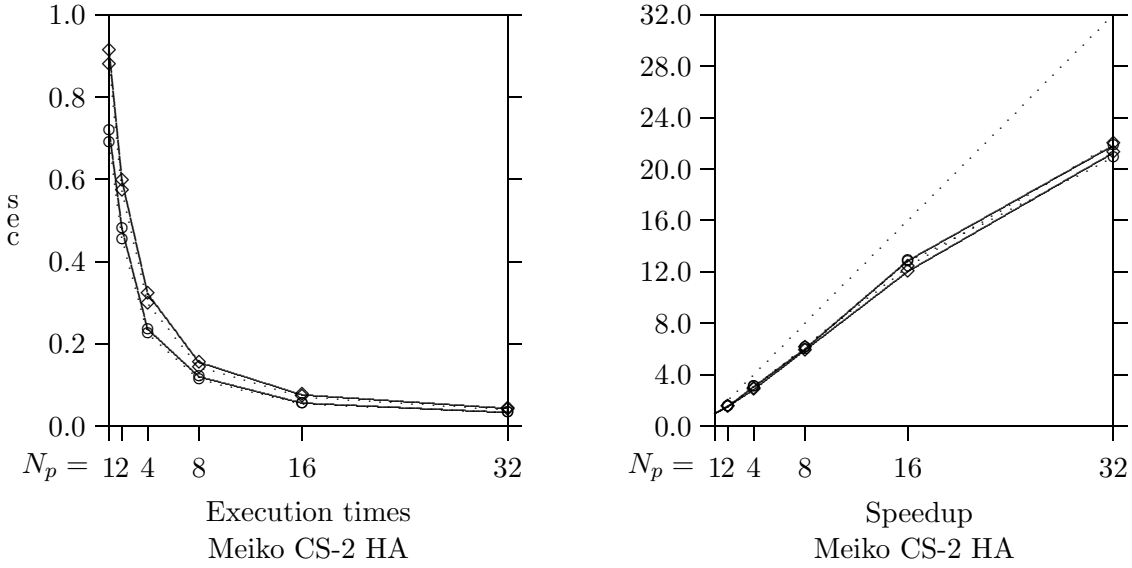
	method	$10\Delta t$-period	$8\Delta t$-period	$6\Delta t$-period
	backward	0.108		
	all paths	9.694		
	10%	0.914		
	1%	0.106		
nested	all paths	27614.018	2579.550	255.928
	10%	4712.586	6.723	1.435
	1%	110.993	2.543	0.428
	10% once	14.655		
	1% once	2.090		

Table 5-6: Execution times for a variable coupon bond with embedded option (seconds)

5.1.3.2 Performance Results

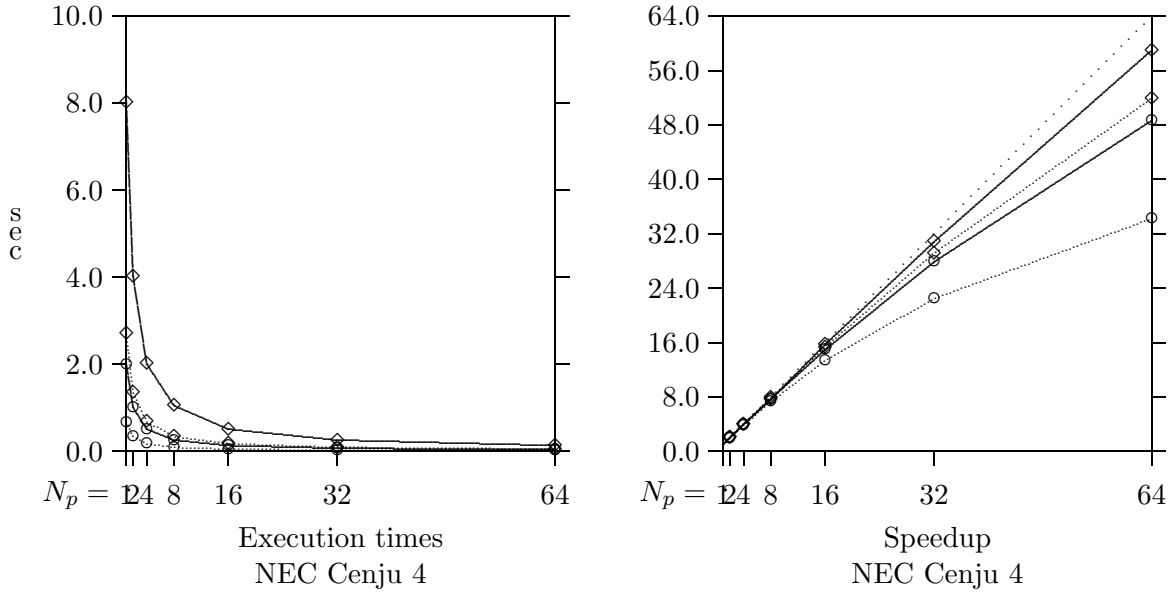
The performance of the parallelized program has been measured on both the Meiko CS-2 HA and NEC Cenju-4 distributed memory systems. The program has been compiled with the Vienna Fortran compiler (VFC) which makes use of the PARTI runtime routines [42]. Figure 5-5 shows execution times of parallel Monte Carlo simulations with 10% of the total 59049 paths sampled on up to 32 compute nodes of the Meiko CS-2. The parallelization is very efficient, since the communication overhead, induced only by the computation of the total sum, is small. The computationally intensive part is performed in parallel without communication.

Figure 5-6 shows the execution times and the speedup for a number of test cases, which vary in the number of timesteps, i.e., the length of the paths and the number of paths sampled, running on up to 64 compute nodes of the NEC Cenju-4. The impact of the communication overhead—due to the sum reduction operation—on the efficiency increases with the number of compute nodes, but decreases with the problem size. For larger problems, the speedup achieved is close to linear. Figure 5-7 shows the effect of different parallelization strategies on the performance of the main simulation loop. In the first version (a), the distribution of array



○	fix coupon
◇	variable coupon
—	with embedded option
·····	without option
N_p	number of compute nodes
· · ·	linear speedup

Figure 5-5: Performance of bond pricing via Monte Carlo simulation—HPF+ version



○	10000 paths
◇	40000 paths
—	40 time steps
·····	10 time steps
N_p	number of compute nodes
· · ·	linear speedup

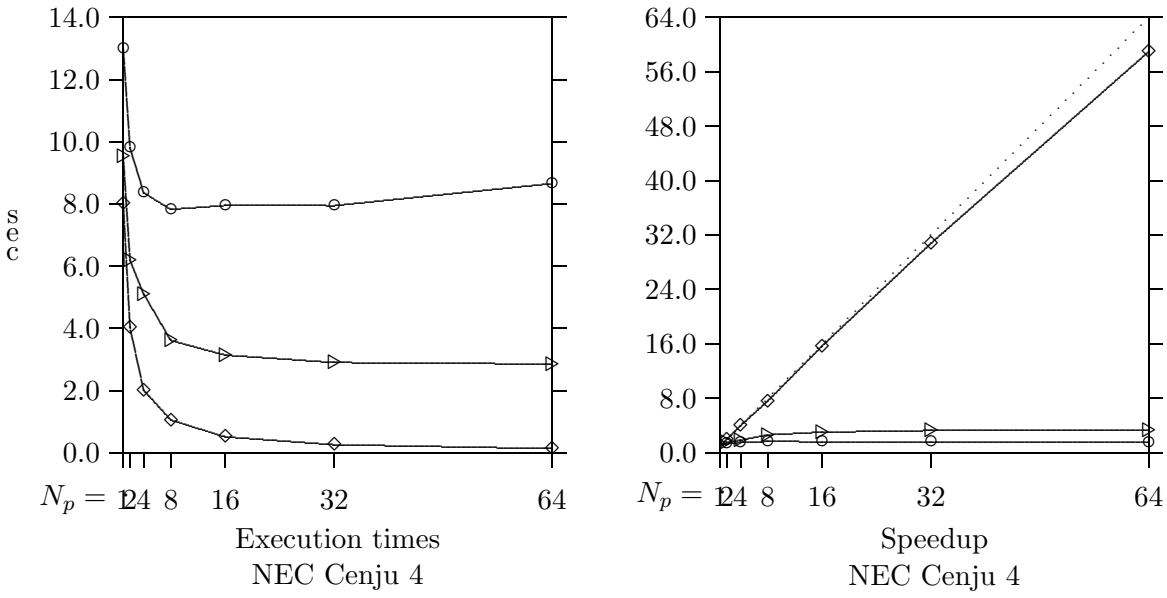
Figure 5-6: Performance of bond pricing via Monte Carlo simulation—HPF+ version

`price`, via the `DISTRIBUTE` directive, onto all compute nodes is the only parallelization directive applied. Every compute node executes all loop iterations, however, the assignment statetment is masked, i.e., it is executed only in case a compute node owns `price(i)`. The summation of the path results is totally replicated, i.e., every compute node computes the final `priceSum`, and thus has to exchange data with every other compute node, which causes a large communication overhead. For the second version (b), the `REDUCTION` directive enforces an efficient machine reduction operation to be employed for the summation. Just as in version (a), the loop is replicated but the assignment statetment is masked. In the third version (c), the `INDEPENDENT` directive specifies that each iteration of main simulation loop can be executed in parallel. Every compute node executes only the iterations which assign owned array elements, according to the owner-compute rule. The experiments demonstrate that version (b) outperforms version (a), and that version (c) is optimal. Therefore, the rest of the performance studies presented in this section is based on the parallelized version (c).

5.2 Backward Induction

The implementation of the backward induction algorithm (see Section 4.1.2) is based on the Hull and White tree [113]. The height of the lattice is specified via `maxJ` $\equiv j_{max} = -j_{min}$. The two-dimensional array `rate` holds the interest rate values `rate(m,j)`, where `m` is the time index and `j` equals the (adjusted) interest rate index $j - j_{min} + 1$ (see Figure 5-8). Successors and probabilities do not depend on the time index and are represented by two-dimensional arrays. Element `successor(k,j)` holds the `j`-index of the `k`-th successor of a node, and element `probability(k,j)` the corresponding transition probability. The present values computed during the backward induction are stored in the two-dimensional array `presentValue`.

The algorithm comprises an outer time step loop (see Figure 5-9), which is executed sequentially. The subroutine `computePVColumn()` calculates the present values at the nodes at time step `m` (see Figure 5-10). The backward induction method is also employed in an extended version. For instruments with limited path dependence (see Section 4.1.3.1), the calculation of the coupon payment at a node uses history paths of length $d - 1$, thereby reducing the complexity of pricing path dependent instruments with limited dependence to an exponential of $d - 1$.



40 time steps, 40000 paths

	loop	summation
○ (a)	replicated	replicated
▷ (b)	replicated	reduction operation
◇ (c)	parallelized	reduction operation
N_p	number of compute nodes	
···	linear speedup	

Figure 5-7: Different parallelization strategies for Monte Carlo simulation

```
      INTEGER, DIMENSION(:) :: HH, HP                    ! halo information

      DOUBLE PRECISION, DIMENSION(2*maxJ+1, maxM)       :: rate
      INTEGER, DIMENSION(3,2*maxJ+1)                    :: successor
      DOUBLE PRECISION, DIMENSION(3,2*maxJ+1)           :: probability

      DOUBLE PRECISION, DIMENSION                       :: principal, price
      DOUBLE PRECISION, DIMENSION(2*maxJ+1, maxM+1)     :: presentValue
      DOUBLE PRECISION, DIMENSION(2*maxJ+1)             :: pVCol, rateCol
      ...

!HPF$ PROCESSORS p(number_of_processors())
!HPF+ NODES n(number_of_nodes())                          ! abstract nodes arrangement
!HPF+ DISTRIBUTE p(BLOCK) ONTO n                          ! processor mapping

!HPF$ DISTRIBUTE (BLOCK,*) ONTO p                        :: presentValue, rate
!HPF$ DISTRIBUTE (*,BLOCK) ONTO p                        :: probability, successor
!HPF$ DISTRIBUTE (BLOCK) ONTO p                          :: pVCol, rateCol
!HPF+ DISTRIBUTE (BLOCK) ONTO p,                         &! HPF+ halo specification
          HALO(HH(HP(I):HP(I+1)-1)) on p(I)              :: pVColOld
```

Figure 5-8: Data structures and distribution specification of HPF+ backward induction

5.2.1 Parallelization Strategy

Both the arrays `rate` and `presentValue` are distributed in the first dimension, whereas the arrays `probability` and `successor` are distributed in the second dimension. In order to optimize for clusters of SMPs, the hierarchical structure of the cluster is specified by means of a processor mapping directive, which specifies a distribution of a processor arrangement to a nodes arrangement. A processor mapping as described in Section 2.3.1 is used, which distributes the processor array `p` block-wise onto the array of compute nodes `n`.

The temporary one-dimensional arrays `rateCol`, `pVCol`, and `pVColOld` support the efficient parallelization with the VFC compiler through storing exactly the column slices of the arrays `rate` and `presentValue` which are required during the call of `computePVColumn()` in time step `m`. Array `pVColOld` holds the present values computed during the previous call (in time step `m+1`). These temporary arrays are distributed as well.

5.2.2 HPF+ Version

The specification of the data distribution in the HPF+ implementation of the backward induction algorithm is shown in Figure 5-8. On each processor, the set of non-local elements of `pvColOld` required by its neighbors has been specified by means of the HPF+ `HALO` attribute [7]. The halo information is computed for each processor via a simple function at runtime and is represented by the arrays `HH` and `HP`. Communication associated with the halo of array `pvColOld` is triggered explicitly by means of the HPF+ `UPDATE_HALO` directive, as shown in Figure 5-9. Subsequently, no communication is required during the execution of `computePVColumn()`. The `j`-loop in `computePVColumn()` can be fully parallelized since it does not exhibit any loop-carried dependencies (see Figure 5-10). In particular, the `computeCoupon()` function does not access the present values computed. This is asserted through the `INDEPENDENT` directive.

```
      presentValue(:,maxM+1) = principal

      DO m = maxM, 1, -1                 ! time step loop

          rateCol(:)  = rate(:,m)
          pVColOld(:) = presentValue(:,m+1)

!HPF+     UPDATE_HALO :: pVColOld             ! update halo area
          CALL computePVColumn(m)             ! compute pVCol(m)
          present_value(:,m) = pVCol(:)

      END DO

      price = presentValue(maxJ+1,1)         ! at root node
```

Figure 5-9: Main loop in HPF+ backward induction

Due to the indirect data accesses via the index array `successor`, the VFC compiler applies runtime parallelization techniques. For the hybrid parallel execution (MPI process parallelism across compute nodes, OpenMP thread parallelism within compute nodes) the directive `!$OMP PARALLEL DO` is inserted. Because the access patterns of the distributed arrays are invariant for all calls of `computePVColumn()`, the HPF+ clause `REUSE` is specified. This allows the VFC compiler to minimize the overheads of runtime parallelization, i.e., the translation of the loop from global to local indices is performed only once, when `computePVColumn()` is

executed for the first time. Moreover, due to the use of halos, no communication is required within `computePVColumn()`. Since the communication pattern required for accesses to array `pvColOld` has been explicitly specified by means of the HPF+ `HALO` attribute, the communication schedules required for non-local data accesses to `pvColOld` need not to be computed by an inspector, but can be determined at the time the distribution of `pvColOld` is evaluated.

```
      SUBROUTINE computePVColumn(m)
!     computes pVCol(m), using rateCol, pVColOld
      INTEGER, INTENT(IN) :: m

!HPF+ INDEPENDENT, NEW(c), ON HOME(pVCol(j)), REUSE
      DO j = 1, 2*maxJ+1
          c = computeCoupon(m,rateCol(j))   ! compute coupon at successor nodes
          pVCol(j) = exp(-deltaT*rateCol(j)) * (                        &
              probability(UP,j)   * (pVColOld(successor(UP,j))  + c) + &
              probability(MID,j)  * (pVColOld(successor(MID,j)) + c) + &
              probability(DOWN,j) * (pVColOld(successor(DOWN,j))+ c) )
      END DO

 END SUBROUTINE computePVColumn
```

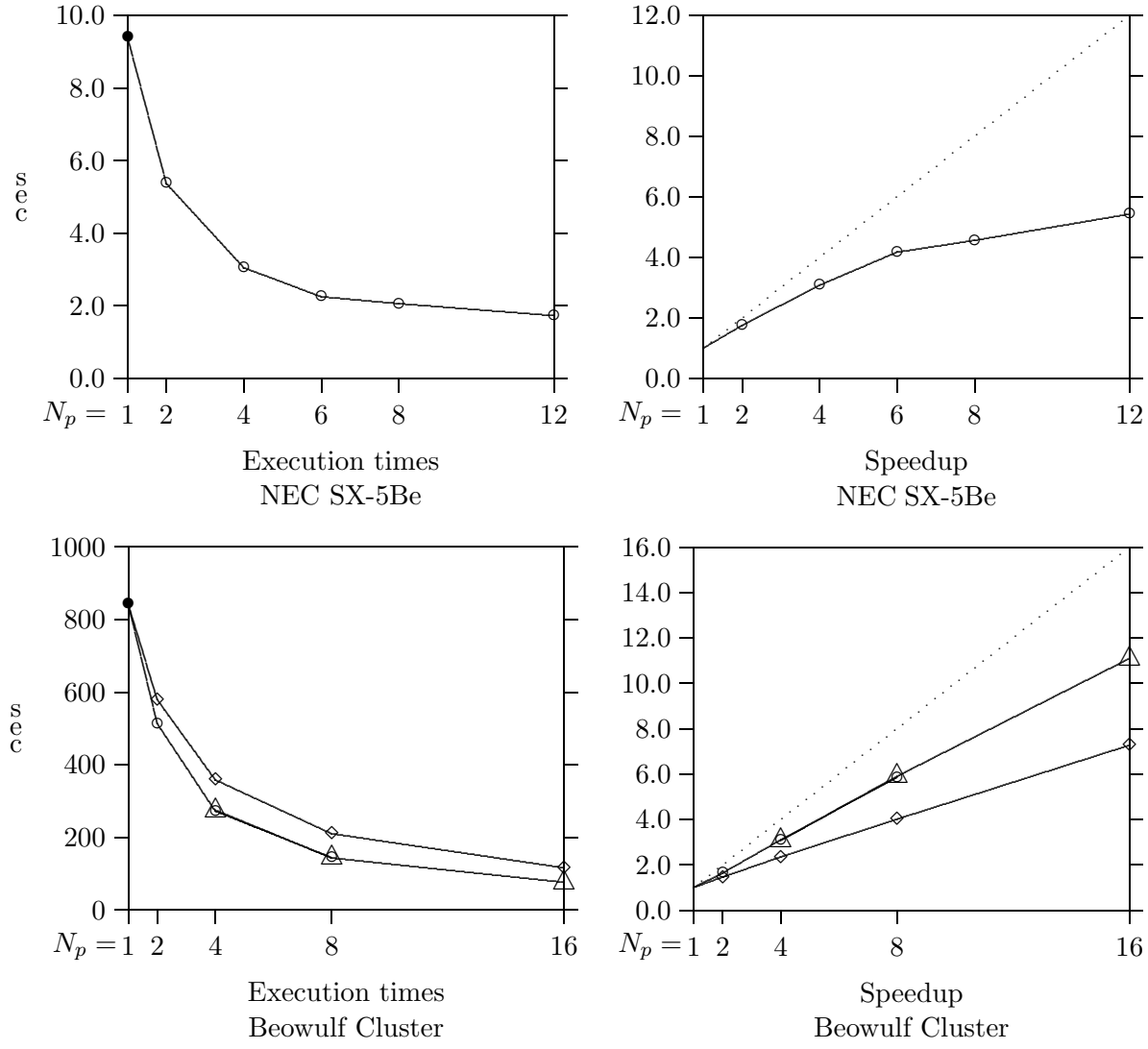
Figure 5-10: Computation at time step `m` in HPF+ backward induction

The data dependence carried by the time step loop induces single element communication among processors, which, depending on the computational complexity of the function `computeCoupon()`, can impair the parallel performance. On architectures with multi-processor compute nodes communicating via shared memory such as clusters of SMPs, processors within the same compute node perform the exchange of elements of `pVColOld` through shared memory accesses, thus avoiding communication over the network.

For instruments with limited path dependence, `computeCoupon()` processes history paths of a node. The increased workload results, depending on the value of d, in a higher computation/communication ratio and an improved efficiency.

5.2.3 Experimental Results

In the performance experiments, the price of a variable coupon bond with a maturity of 10 years is calculated on a one-day basis. The workload of the `computeCoupon()` function used is



•	sequential	# processors per compute node
○	(a) HPF+/MPI	1
◇	(b) HPF+/MPI	2
△	(c) HPF+/MPI-OpenMP	2
N_p	number of processors	

Figure 5-11: Performance of backward induction—HPF+ and hybrid-parallel version

slightly smaller than the workload of the extended version processing paths of length one. The code has been parallelized with the VFC compiler and executed on a NEC SX-5Be compute node as well as on a Beowulf cluster consisting of 8 compute nodes, each equipped with two Pentium II processors, connected via Fast Ethernet. On the SX-5, the Fortran90/MPI code generated by the VFC compiler has been compiled with the NEC `sxf90`, while on the PC cluster the Portland `pgf90` compiler has been employed as a backend compiler of VFC. Figure 5-11 shows the performance results.

For the MPI code generated by VFC to be executed on the SX-5, special care has been taken in order to support the `sxf90` compiler in fully vectorizing the `j`-loop in subroutine `computeCol()`. The function `computeCoupon()` has been inline-expanded. The speedups achieved are quite satisfactory.

On the PC cluster, two different parallel code versions were generated by VFC. The first version is a pure MPI message passing code, while the second version employs a hybrid execution model utilizing MPI process parallelism across the compute nodes of the cluster and OpenMP thread parallelism within compute nodes. The pure MPI version of the generated code has been executed on two to eight compute nodes, where on each compute node only one processor is utilized and no intra-node communication takes place (a). In addition, it has been executed on one to eight compute nodes, where on each compute node both processors are utilized (b), i.e., on two to sixteen processors. Both inter- and intra-node communication is performed via calls to MPI routines. The hybrid MPI-OpenMP version (c), also utilizing both processors at each compute node, has been executed on two to eight compute nodes, i.e., on four to sixteen processors. It performs MPI communication across compute nodes and OpenMP shared memory accesses within compute nodes. This version was generated by VFC by taking into account the HPF extensions, i.e., the `NODES` directive and processor mapping.

Figure 5-11 demonstrates that the MPI-OpenMP code generated by VFC (c) outperforms the pure MPI version on two processors per compute node (b). One reason is seen in the overhead induced by intra-node MPI communication. The MPI implementation employed does not provide an optimized intra-node communication via `shmem` within the compute nodes of the cluster. In addition, variant (b) requires twice as much memory for data structures which are shared in variant (c). The utilization of both processors of a compute node can induce overheads

due to the potential of bottlenecks during the shared use of memory and other resources. These overheads are assumed to be responsible for the poorer performance of variant (b) as compared to variant (a). The intra-node MPI communication can inhibit and slow down the inter-node communication performed simultaneously. The times measured for variants (a) and (c) are almost the same, however, variant (a) requires twice as many compute nodes. Also on the Beowulf cluster the parallelized backward induction kernel exhibits good speedups.

5.3 Generalized Backward Induction

5.3.1 Parallelization Strategy

The generalized backward induction algorithm (see Section 4.1.3.3) sequentially proceeds backwards from the last to the first time step, whereas all operations at a time step can be performed in parallel. It is parallelizable analogous to the classical Backward Induction (see Section 5.2). The lattice is partitioned into blocks of rows, each processor computes the incomplete present values within one block. Due to the loop carried dependence in the time step loop, values computed by the neighboring processor in the previous time step are required. On parallel architectures with distributed memory, the resulting single element communication can impair the parallelization. In the generalized Backward Induction, the additional processing of 3^{d-1} paths at a node results in higher computation/communication ratios and thus improved efficiency.

5.3.2 Fortran 95π Version

The computation of the incomplete present value $PV^{(d)}$ at every node of the lattice via generalized backward induction [111] is presented in two versions using the notation introduced in Section 2.4.3. Both versions possess parallelism, expressed by the `FORALL` construct.

Figure 5-12 lists the variable declarations. The first version, shown in Figure 5-13, implements (4.8) and directly calculates the present value at node $n = (m, j)$ of the cash-flows at time step $m+d$. The second version, shown in Figure 5-14, reorders discounting and summation operations, resulting in shorter paths and a smaller number of discounting operations. At the root node, the incomplete present value is completed based on a history path (see Figure 5-15). The `IF`-statement as well as the variable `v` within the body of the `FORALL` construct have been

inserted to clearly arrange the initialization of the `timesteps`–loop. After applying a loop peeling transformation related to the first iteration, they can be eliminated.

```
integer           ::    d           ! depth of limited path dependence
path, length(d-2) ::    p0          ! history path to root node

real, index(node) ::    PV_d(:)     ! incomplete present value at node

real, index(path) ::    pvcash(:)   ! present value of cash-flows
real, index(path) ::    pvsucc(:)   ! present value of PV_d of successors
real, index(path) ::    pvroot(:)   ! present value of cash-flows near root node

real              ::    PV          ! final present value at root node

path, length(:)   ::    p           ! path
path, length(1)   ::    s           ! path to successor
node              ::    n           ! node

integer           ::    m           ! time step
real              ::    v           ! temporary value
```

Figure 5-12: Variable declarations of generalized backward induction code

5.3.3 Experimental Results

Table 5-7 shows the price of a path dependent variable coupon bond with a maturity of 18 periods, paying at timestep m the average of the interest rates at time steps $m-4, \ldots, m-1$. The interest rate lattice has 89 nodes. In addition, the number of paths considered and the execution times on a Sun Ultra 4 workstation, using Monte Carlo Simulation, with varying sample sizes, and generalized backward induction, are given.

method	number of paths	price	execution time (sec)
MC simulation	$3.87\ 10^3$	8573.703	0.65
MC simulation	$3.87\ 10^4$	8590.857	6.55
MC simulation	$3.87\ 10^5$	8589.274	64.17
Generalized Backward	(all) $3.87\ 10^8$	8586.582	1.81

Table 5-7: Performance of pricing an instrument with limited path dependence

```
 timestep: DO m = LENGTH()-d, 0, -1

     node: FORALL (n = (m,:))
        ALLOCATE(pvcash,pvsucc)

        path: FORALL (p = (n|d-1))
            ! present value at n of cash-flows paid at timestep m+d
            pvcash(p) = (CASHFLOW(p) .DISCOUNT. p) * p%PROBABILITY
        END FORALL paths

        init: IF (m == LENGTH()-d)
            ! first iteration of timestep-loop
            PV_d(n) = SUM(pvcash)
        ELSE

            successors: FORALL (s = (n|1))
                ! present value at n of incomplete p.v. at successor node
                pvsucc(s) = (PV_d(s%NODE(1)) .DISCOUNT. n) * s%PROBABILITY
            END FORALL successors

            PV_d(n) = SUM(pvcash) + SUM(pvsucc)
        END IF init

        DEALLOCATE(pvcash,pvsucc)
    END FORALL node

END DO timestep
```

Figure 5-13: Generalized backward induction (4.8)

```
timesteps: DO m = LENGTH()-d, 0,-1

    node: FORALL (n = (m,:))
        ALLOCATE(pvsucc)

        successor: FORALL (s = (n|1))
            ALLOCATE(pvcash)

            path: FORALL (p IN (s%NODE(1)|d-2))
                ! present value at successor node of cash-flows paid at timestep m+d
                pvcash(p) = (cashFlow(n & p) .DISCOUNT. p)* p%PROBABILITY
            END FORALL path

            init: IF (m == LENGTH()-d)
                ! first iteration of timestep-loop
                v = SUM(pvcash)
            ELSE
                ! add PVd at successor node
                v = SUM(pvcash) + PV_d(s%NODE(1))
            END IF init

            ! present value at n of v
            pvsucc(s) = (v.DISCOUNT.n) * s%PROBABILITY

            DEALLOCATE(pvcash)
        END FORALL successors

        PV_d(n) = SUM(pvsucc)

    DEALLOCATE(pvsucc)
    END FORALL node

END DO timesteps
```

Figure 5-14: Generalized backward induction (4.9)

```
ALLOCATE(pvroot)
FORALL (m = 0:d-2)
    FORALL (p == ((0,0)|i))
        ! PV at root of cash-flow paid at m+1
        pvroot(p) = (CASHFLOW(p0 & p) .DISCOUNT. p) * p%PROBABILITY
    END FORALL
END FORALL
PV = PVd(0,0) + SUM(pvroot)
DEALLOCATE(pvroot)
```

Figure 5-15: Completion operation at the root node

5.4 Nested Benders Decomposition

Due to the computational complexity of nested Benders decomposition, parallel decomposition methods have been developed [46, 115, 128]. In the following, the synchronous as well as the asynchronous parallelization approach is described.

5.4.1 Synchronous Parallelization

In the nested Benders decomposition algorithm, every node performs an iterative procedure, acting as a master, a slave, or both of them (see Section 4.2.2). With a synchronous computation scheme, each node except the root receives the pair $(x_m^{(\hat{i})}, \hat{q})$ from its master in order to build and solve a local linear program. Then it sends the pair $(x_n, Q_{n,k}^{(i)}(x_n))$ to every slave. Subsequently it waits until it receives a complete new set of cuts. It cannot continue before it received cuts from every slave. This coupling results in a sequence of forward and backward sweeps over the whole tree, as illustrated in Figure 5-16. Every sweep in turn comprises a sequence of simultaneous activities of the tree nodes at a certain level, whereas the rest of the nodes remain idle. Hence, in a parallel version, the synchronous communication scheme exhibits poor processor utilization, because it allows for parallelism (indicated by dotted boxes in Figure 5-16) only within nodes at the same level.

5.4.2 Asynchronous Parallelization

In an asynchronous version, every node waits until it has received data from the master or from at least one of its slaves, i.e., the local problem is solved using only the best information

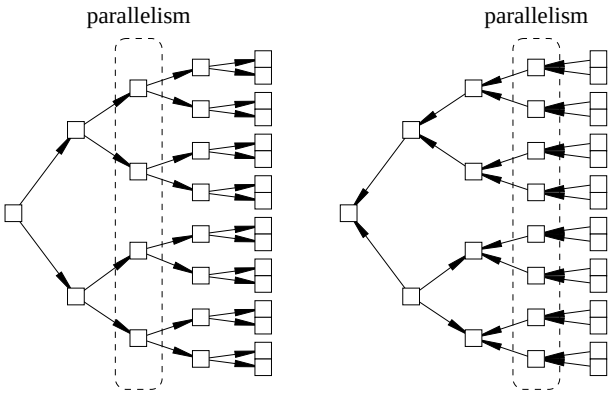


Figure 5-16: Flow of information during synchronous forward and backward sweeps

obtainable so far [117]. A node does not have to wait for its master to send a new solution (during a forward sweep) and for all slaves to send their cuts (during a backward sweep) prior to start a new iteration. Rather, it is sufficient to receive any new data, thus more nodes can be simultaneously active. However, since in the asynchronous version there is no tight coupling between sending and receiving nodes, the data received has to be stored in communication buffers. It is possible that the master's solution did not change from the last (local) iteration, since the master did not yet finish its optimization, and that not all slaves sent new cuts. At each new iteration cycle, a node reads the buffers and uses the data to build a new local linear program.

The activity diagram in Figure 5-17 shows the asynchronous node algorithm. Q denotes the set of optimality-, and R denotes the set of feasibility cuts, respectively. Every node performs an activity in parallel with all other nodes. The activity consists of a loop, in which a linear program is repeatedly created and solved. As each node executes the same program, but operates on different data, the situation adheres to the SPMD programming model. All tree nodes are distributed onto the compute nodes of a parallel system. On each compute node, the tree node computations are executed concurrently. Figure 5-18 illustrates the coordination of a node with its neighbors. Subactivity states represent a preprocessing (construction of new linear program) and postprocessing (cut calculation) phase.

Figure 5-19 depicts an asynchronous execution scenario. After the first iteration, the node starts a second one, as soon as it got the new cut 4 from slave 1. In particular, it does not

wait for cut 5 and solution 6, which are incorporated into the local problem not until the third iteration. Since the algorithm is an iterative procedure with increasing accuracy, it is not relevant whether a particular information is provided sooner or later. However, because a new local problem is constituted by a smaller amount of new information than in the synchronous version, more iterations are needed in the asynchronous version. The argument for asynchronous parallelization is that the additional computational work is more than made up by improved parallelism due to the fact that tree nodes at more than one levels may be simultaneously active in the asynchronous version.

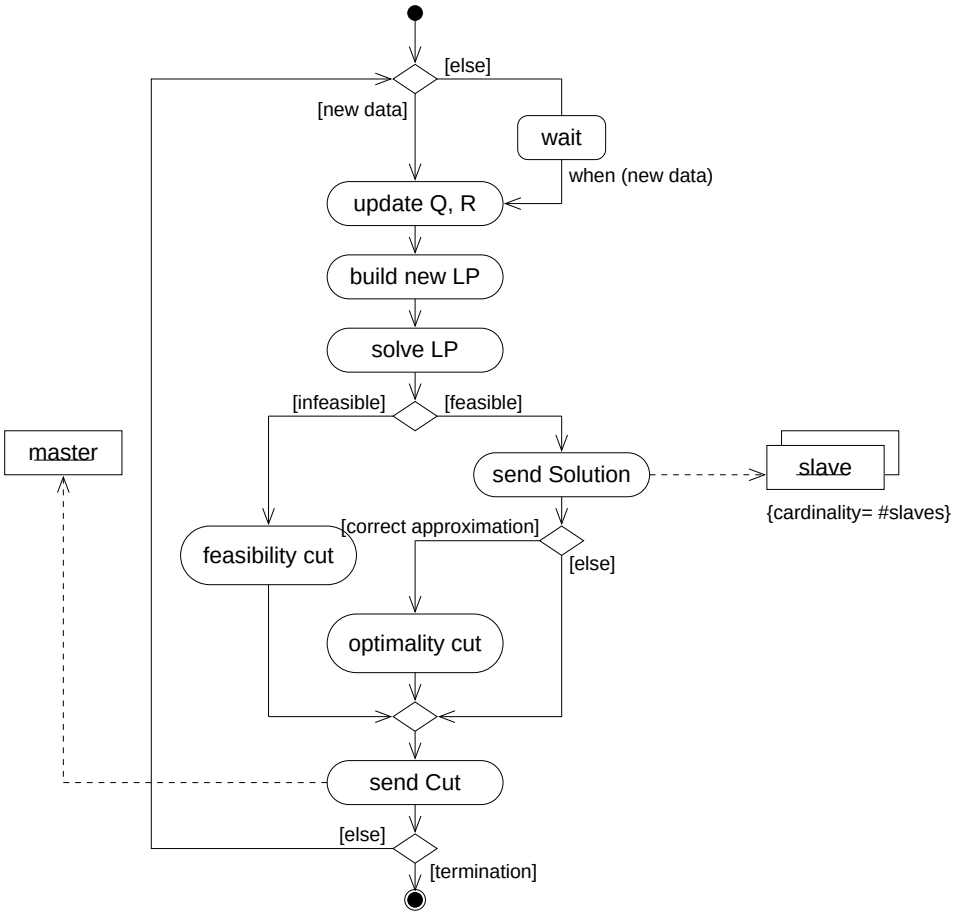


Figure 5-17: Node algorithm of asynchronous nested Benders decomposition

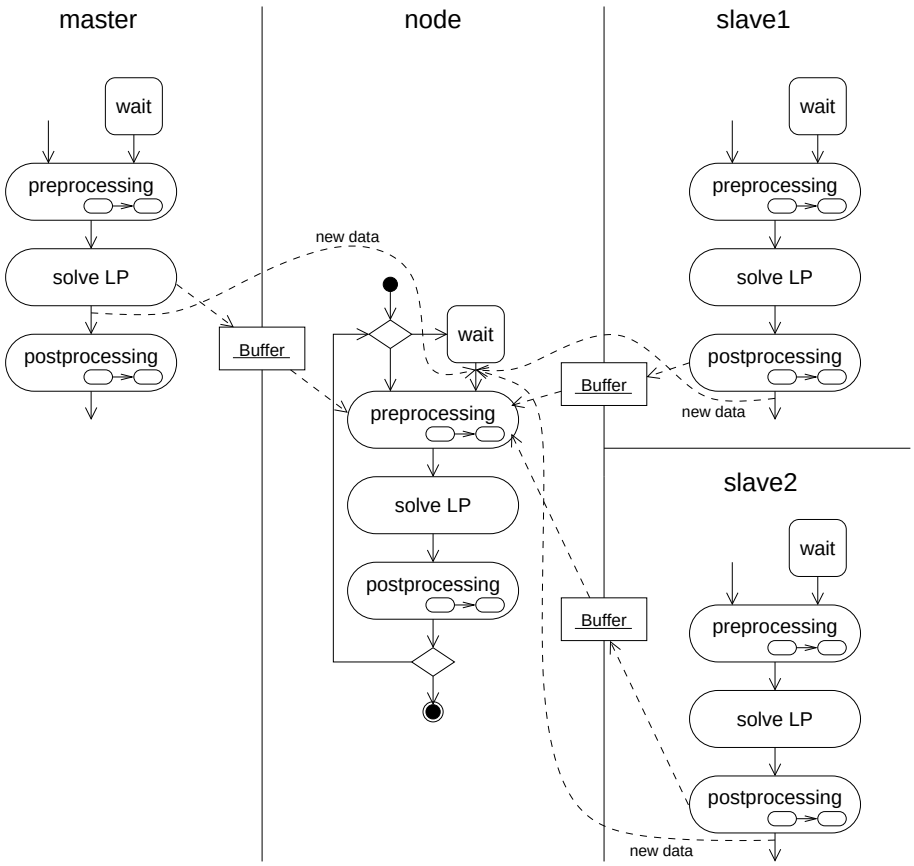


Figure 5-18: Interaction among node programs in nested Benders decomposition

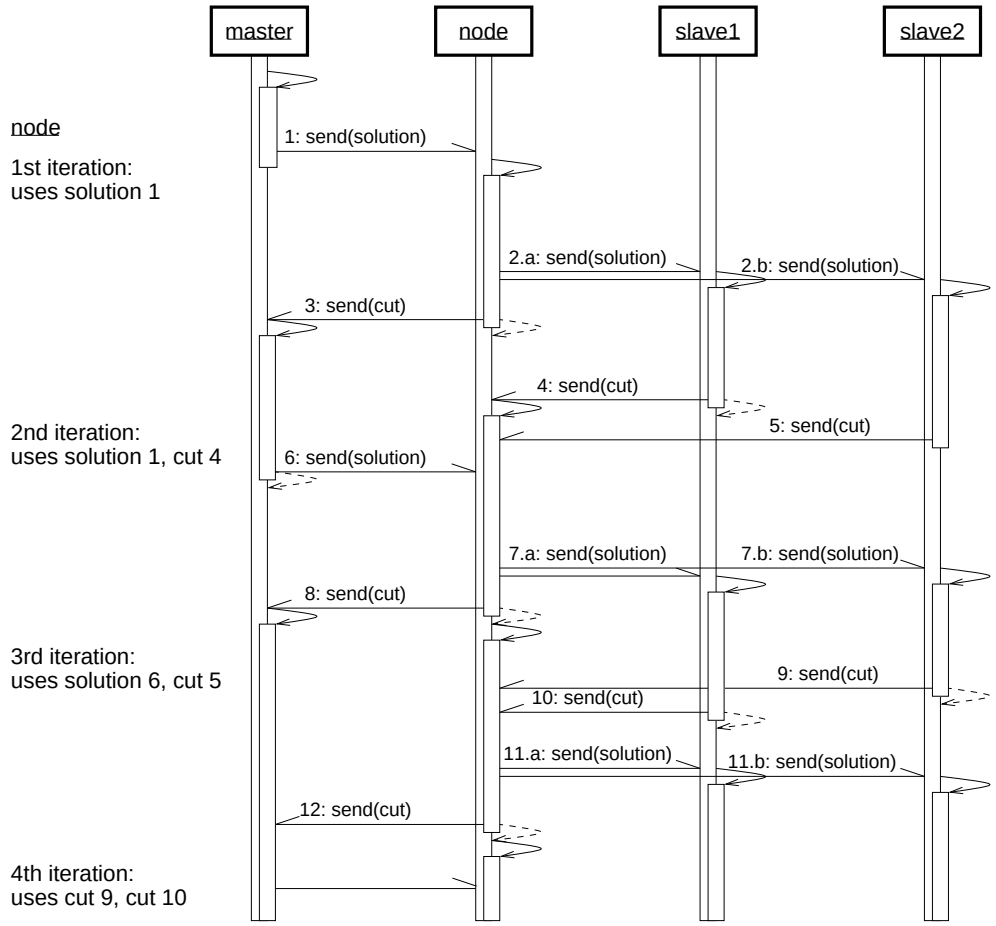


Figure 5-19: Asynchronous execution scenario of nested Benders decomposition

5.4.3 Java Distributed Active Tree Version

The parallel version of the nested Benders decomposition algorithm can be directly expressed as a distributed active tree [118]. The node program shown in Figure 5-17 is implemented as an `AlgorithmNode` class by overriding the `iteration()` and `terminate()` methods, specifying the body and termination condition of the main loop. Figure 5-20 shows the variable fields of the `NestedBendersNode` class, which contains the inner classes `Solution` and `Cut` with subclasses. Figure 5-21 presents the loop body of the node algorithm. Using predefined accessibles and the standard synchronization constraints, provided by the coordination layer through `getAllNeighbours()`, `getSuccessors()`, `getPredecessor()`, and `Pattern.all`, `Pattern.any`, respectively, and the variable `source`, all the communication and synchronization needed for both the asynchronous and the synchronous version, and, for the latter, for both the forward sweep and the backward sweep can be expressed at a high level of abstraction within two statements. The implementation of the algorithm does not need explicit buffers and is combined with a coordination layer implemented in Java, on top of Java remote method invocation. Details related to the solving of local linear programs are discussed in Section 5.4.6.

5.4.4 OpusJava Version

The asynchronous version of nested Benders decomposition has also been implemented employing OpusJava. The description in this subsection is a short summary of the work presented in [97, 98]. Tree nodes are implemented as SDA objects, because they perform their activity in parallel with other nodes. In order to prevent data which arrives during the solution of the linear program from overwriting the data which is currently used in the solution procedure, all incoming data at a node is stored in a buffer associated with the node. The buffer is implemented as an SDA as well, allowing for exploiting the high level synchronization mechanism provided by OpusJava. As SDAs have monitor semantics, it is not necessary to synchronize accesses to the buffer. The synchronization condition is expressed as a condition clause guarding the method which retrieves the data from the buffer, thus no synchronization is needed within the nodes. Figure 5-22 shows the `Buffer` class. It contains instance variables for solution and cuts, and their respective access methods. The condition for executing `getData()` is specified in the method `getData_cond()`.

```
class NestedBendersNode implements dat.alg.AlgorithmNode {

    abstract class BendersData implements AlgorithmData, SerializableCoordinationData
        {...}
    class Solution extends BendersData
        {...}
    abstract class Cut extends BendersData
        {...}
    class FeasibilityCut extends Cut
        {...}
    class OptimalityCut extends Cut
        {...}

    dat.rmi.Coordination coordination;  // the coordination object of this node

    boolean root;                       // this is the root node
    boolean master, slave;              // node acts as master and / or slave
    boolean synchronous;                // synchronous / asynchronous version

    LP lp;                              // linear program object
    Cut nullCut;                        // null cut signal

                                        // local data:
    Solution solutionMaster;            // solution received from master
    Vector Q[];                         // optimality cuts received from slaves
    Vector R[];                         // feasibility cuts received from slaves
    boolean forward;                    // sweep direction is forward / backward
    boolean optimize;                   // perform optimization
    ...
    }
```

Figure 5-20: Data fields of the distributed active tree node

```
public void iteration() throws CoordinationException, LPException {

    dat.coord.Accessible source;         // source nodes to get data from
    Solution solution;                   // solution of the local LP
    Cut cut;                             // cut to be sent to the master

    source = (synchronous ? (            // determine source nodes
                forward ?
                    coordination.getPredecessor() :
                    coordination.getSuccessors()
                    ) :
                coordination.getAllNeighbours()
                );
    source.get(synchronous ?             // get data (possibly wait for it)
                 dat.coord.Pattern.all :
                 dat.coord.Pattern.any
                 );
                                         // local data structures are updated
    if (optimize) {
        lp = new LP(solutionMaster,Q,R); // create and solve the linear program
        solution = lp.getSolution();

        switch (lp.getStatus()) {
            case LP.OPTIMAL:
                if (master)
                    coordination.getSuccessors().put(solution);
                if (slave) {
                    if (solution.getQ() <= solutionMaster.getQ(this))
                        cut = nullCut;
                    else
                        cut = new OptimalityCut(solution);
                    }
                break;
            case LP.INFEASIBLE:
                if (root)
                    throw new LPException(this,LP.INFEASIBLE);
                if (slave)
                    cut = new FeasibilityCut(lp);
                break;
            case LP.UNBOUNDED:
                throw new LPException(this,LP.UNBOUNDED);
                break;
            }
        if (slave)
            coordination.getPredecessor().put(cut);
        }
    }
```

Figure 5-21: Iteration in the distributed active tree node

```
public class Buffer {

    private Solution masterData;                // Buffer for predecessor
    private Vector slaveData[];                 // Buffer for successors

    public void putSlaveData(Cut cut, ...)      // data from successor
        {...}
    public void putMasterData(Solution s, ...) // data from predecessor
        {...}
    public Object[] getData()                   // reads all data out from the buffer
        {...}
    public boolean getData_cond()               // condition clause for getData()
        {...}
    }
```

Figure 5-22: The OpusJava buffer

Figure 5-23 shows the implementation of a tree node as an SDA `Node`. It contains references to other SDAs, actually to its own buffer, to the successor nodes, and to the buffers of both the successor and predecessor nodes. The constructor is parameterized and requires the caller to pass a reference to an SDA object, in this case to the buffer. When a node instantiates its successors, it passes its local buffer as a parameter. The tree is constructed by the method `init()`. After the local buffer of a node is instantiated, 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 compute nodes. The `init()` method of the successors is invoked in an asynchronous manner, causing a parallel growth of the tree. The return value of `init()` is a reference to the local buffer which is stored in the variable `successorBuffer`. The `init()` methods of all successors run in parallel, associated with `Event` objects acting as container for their results. Via calling the `getResults()` method of such an event, the caller synchronizes with the callee and eventually retrieves the results of the asynchronous call. The method `iterate()`, shown in Figure 5-24, implements one iteration of the main loop of the node algorithm. Once it is invoked within a node, the request is in parallel propagated to the successor nodes. After having initiated the computation in the successor nodes, a node enters the main loop.

```
public class Node {

    SDA successors[];               // successor Nodes;
    SDA predecessorBuffer;          // predecessor buffer
    SDA myBuffer;                   // local buffers
    SDA successorBuffer[];          // successor buffers
    // problem specific data
    ...
    public Node (SDA buffer) {      // constructor
       predecessorBuffer = buffer;
       ...
       }

    public SDA init(...) {

        myBuffer = new SDA("Buffer", ...);
        successors = new SDA[nrOfSuccessors]; // create local buffer
        Event e[] = new Event[nrOfSuccessors];

        if (!leaf()) {

            // create successors passing the local buffer as input argument
            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",...);
               }

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

Figure 5-23: Initialisation in the OpusJava node

```
public void iterate() {
   Event e[] = new Event[nrOfSuccessors];

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

   do {                                    // main loop
       // 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", ...);
           }

       if (!root()) {
           // send cuts to predecessor
           predecessorBuffer.call("putSlaveData",...);
           }

       } while (!solutionFound);
   }
```

Figure 5-24: Main loop in the OpusJava Node

5.4.5 JavaSymphony Version

The asynchronous version of the nested Benders decomposition algorithm has also been implemented on top of JavaSymphony [54]. This version has been built starting from the existing Java distributed active tree implementation (see Section 5.4.3). Both the coordination layer and and the initialization part have been adapted to the JavaSymphony API, whereas the implementation of the node algorithm remained unchanged. The threads associated with the tree nodes are encapsulated within *JSObject*s. Communication among tree node objects located at the same compute node is performed through direct access, and among objects residing on different computing nodes through JavaSymphony remote method invocation.

5.4.6 Experimental Results

A number of experiments have been conducted on various hardware platforms, including a cluster of Sun Ultra 10 workstations connected via fast Ethernet, and a Beowulf cluster consisting of quad-board PC compute nodes, each equipped with 4 Pentium III Xeon 700MHz processors.

In the following, the performance results of the distributed active tree implementation (see Section 5.4.3) are presented. The nested Benders decomposition is written in pure Java and represents the algorithm layer of a distributed active tree. The latter, essentially the coordination layer, is also implemented in Java, on top of Java remote method invocation. Initially, also a Java Simplex routine has been employed, which in fact proved to be numerically very unstable and could be applied for simple problems only. As an alternative, a Fortran solver has been integrated, via wrapper routines in C and the Java Native Interface. Following the node thread scheduling strategy, every tree node possesses a separate thread. Since the solver is not thread safe, it is called from within a `synchronized` block.

Both the synchronous and the asynchronous version of the algorithm have been measured for problems with up to 511 tree nodes, i.e., scenarios. The linear programs generated during the execution of the decomposition algorithm demand for high quality solvers in terms of numerical stability. It is possible, that newly generated constraints turn out to be almost identical to previously generated ones. The solver can expose a "cycling" behavior in which it repeatedly switches among a set of solutions, without improvement. Problems of this kind, related to the solving of linear programs as a subtask in nested Benders decomposition, require the local linear

programs in the experiments to be fairly small. Consequently, short times are spent for computational work, as compared to communication times, and a poor computation/communication ratio limits the speedups achieved. It can be observed, that the application of the algorithm on larger scenario trees in particular suffers from the numerical difficulties described.

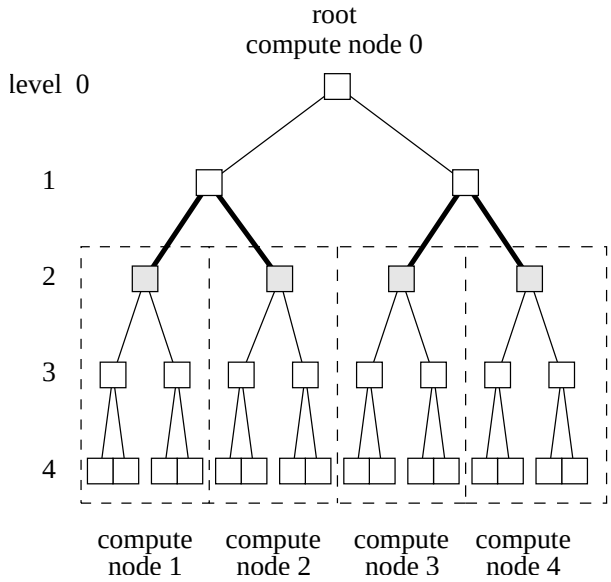


Figure 5-25: Mapping of tree nodes onto compute nodes at distribution level 2

The scenario tree is distributed according to the pattern shown in Figure 5-25. The root node and its descendents up to a certain level are mapped onto the "root" compute node. The rest of the tree is distributed in such a way that subtrees emanating from the "distribution" level (level 2 in Figure 5-25) as a whole are mapped onto the remaining compute nodes. Consequently, they can be processed with remote communication (displayed in bold lines) required to the root compute node only.

Table 5-8 shows the performance of both versions on one, three, and five Sun workstations. The first two columns comprise the problem size, defined by the number of tree nodes N_n, and the number of financial contracts N_c in the portfolio, column 3 shows the number of compute nodes N_p employed, and column 4 the distribution level δ. The initial local constraint matrices are of size $2N_c+1 \times 3N_c+1$. Execution times are given in seconds, t_a for the asynchronous, and t_s for the synchronous version. The speedup sp_a for the asynchronous version equals $t_a^{(N_p)}/t_a^{(1)}$, and for the synchronous version $sp_s = t_s^{(N_p)}/t_s^{(1)}$, where the superscript denotes the number of

N_n	N_c	N_p	δ	t_a	t_s	sp_a	sp_s	t_s/t_a
63	2	1	–	0.846	0.582	-	-	0.68
63	2	3	4	0.905	0.890	0.93	0.65	0.98
63	2	5	3	0.487	0.510	1.73	1.14	1.04
63	3	1	–	0.946	0.483	-	-	0.51
63	3	3	4	1.082	0.865	0.87	0.55	0.80
63	3	5	3	0.630	0.421	1.50	1.14	0.66
127	2	1	–	1.930	1.211	-	-	0.62
127	2	3	5	1.893		1.01		
127	2	5	4	1.035	0.970	1.86	1.24	0.93
127	2	5	5	2.079		0.92		
127	3	1	–	2.268	1.090	-	-	0.48
127	3	3	5	2.083		1.08		
127	3	5	4	1.302	1.223	1.74	0.89	0.93
127	3	5	5	2.150		1.05		
255	2	1	–	4.875	2.575	-	-	0.52
255	2	3	6	3.955		1.23		
255	2	5	5	2.227	2.307	2.18	1.11	1.03
511	2	1	–	9.245		-	-	
511	2	3	7	8.344		1.10		
511	2	5	6	4.493		2.05		

Table 5-8: Performance of nested Benders decomposition on Sun workstation cluster

compute nodes. In the last column, the ratio t_s/t_a of the synchronous and the asynchronous execution time is given. Due to the numerical problems with linear problem solving, some entries of the table remain empty.

The results demonstrate the impact of the tree distribution. The distribution level parameter allows for priorities such as optimal load balancing or minimal communication. For example in the two cases $(N_n, N_p, \delta) = (127, 5, 4)$, 15 tree nodes are assigned to the root compute node and 28 tree nodes to every other compute node, yielding 16 edge cuts. As opposed, with $\delta = 5$, 31 nodes are assigned to the root compute node and 24 nodes to every other compute node, resulting in 32 edge cuts. The first cases perform better, because less communication is required and more processor cycles are available to the thread associated with the root node. Because it provides information to the rest of the tree, fast execution is crucial. As far as the comparison of the two versions is concerned, the synchronous version clearly dominates the asynchronous one on a single compute node, due to its faster convergence in terms of number of iterations. However, the ratio t_s/t_a increases for larger numbers of tree and compute nodes, respectively.

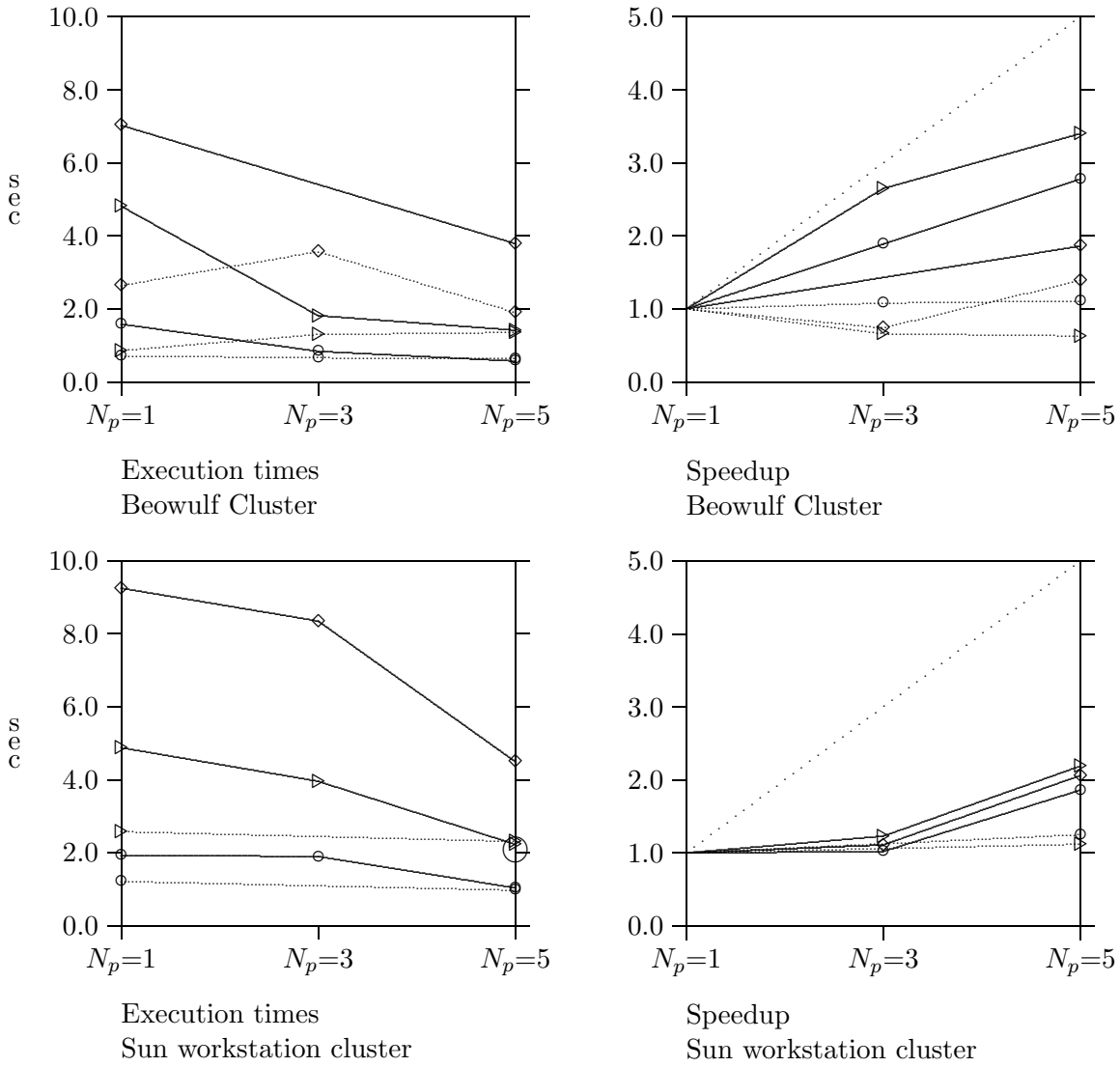


Figure 5-26: Performance of nested Benders decomposition—distributed active tree version

Figure 5-26 shows the performance on both clusters for the cases $N_c = 2 \wedge (N_n, N_p, \delta) \in \{(127, 3, 5), (127, 5, 4), (255, 3, 6), (255, 5, 5), (511, 3, 7), (511, 5, 6), \}$. As an important result of improved processor utilization, the asynchronous algorithm exhibits superior speedups. In the synchronous version, the node threads perform on average less iterations than in the asynchronous one, which results in shorter execution times on a single compute node, whereas in the asynchronous version, the threads spend less time in waiting for new data. Now, when running on more than one compute node in parallel, a compute node as a whole is idle, if all threads are waiting, and shorter idle times result in the larger speedups observed for the asynchronous version.

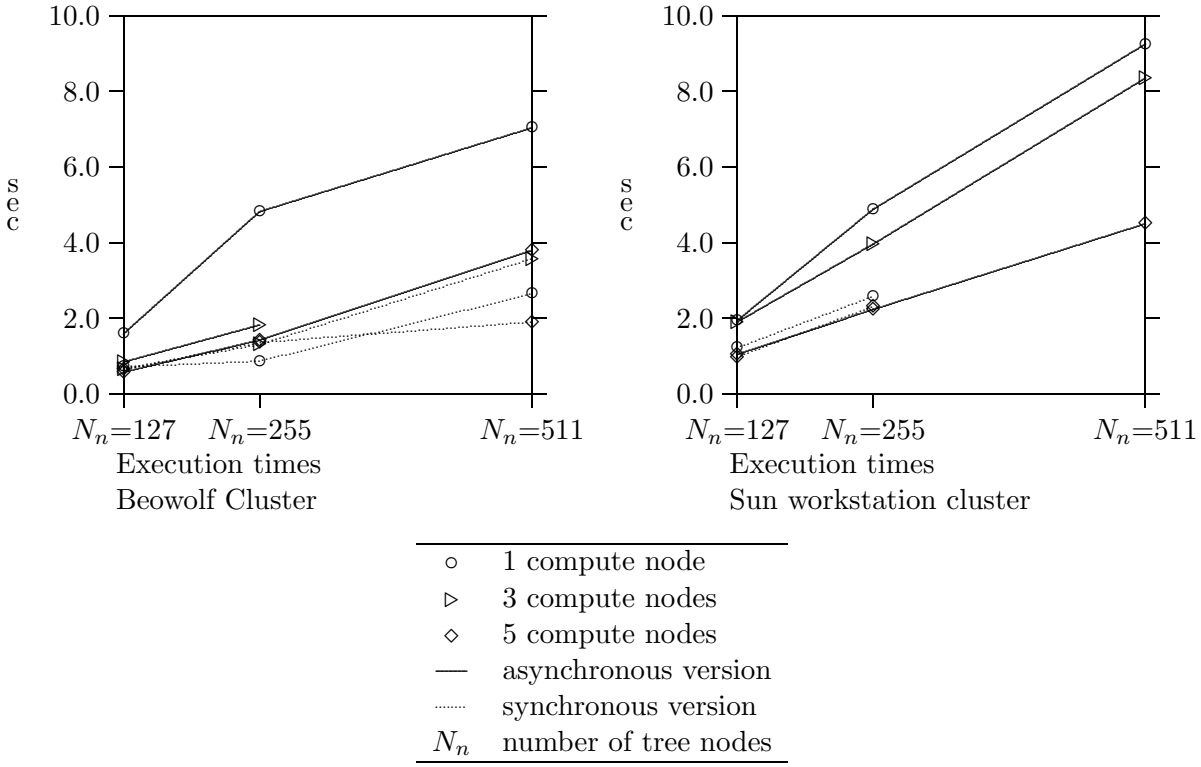


Figure 5-27: Scaling of nested Benders decomposition—distributed active tree version

The experiments also revealed that the number of iterations per tree node increases with the tree size. The resulting increase in the number of communication operations can be to the disadvantage of the asynchronous version.

As a supplement demonstrating the scaling behavior of both versions, Figure 5-27 shows the same execution times as displayed in Figure 5-26 as a function of tree sizes. All in all,

the performance of the asynchronous algorithm is expected to improve with more realistic, i.e., larger problems, due to more advantageous communication/computation ratios.

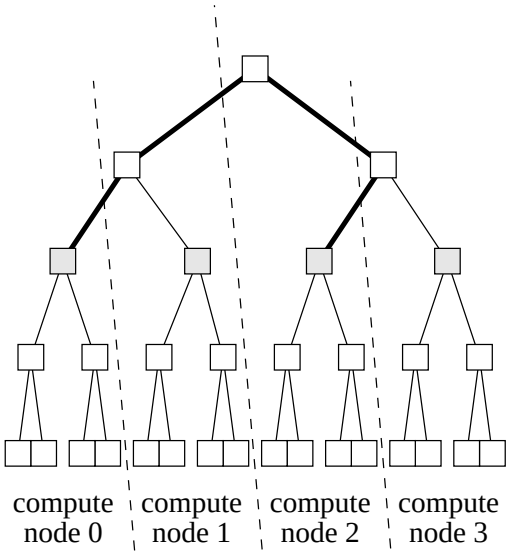


Figure 5-28: Mapping of tree nodes onto compute nodes—optimal load balancing strategy

In the experiments with the OpusJava implementation, a tree distribution is chosen which follows an optimal load balancing strategy as shown in Figure 5-28. As opposed to Figure 5-25, the compute nodes have to remotely exchange data with more than on neighbor. Figure 5-29 shows the execution times and corresponding speedups, on one, two and four Sun workstations and on one, two, four, and eight compute nodes of the Beowulf cluster, respectively. Every tree node there are two threads associated with, the first one for the buffer and the second one for the algorithm (see Section 5.4.4). Due to the small size of the local problems, the communication overhead overwhelms parallelism gains also in the OpusJava version.

On the Beowulf cluster, a significant thread scheduling overhead caused by the runtime system, i.e., the Java virtual machine and the operating system, can be observed when a lot of tree nodes are mapped to a compute node, in particular, when the maximum of 511 nodes is mapped to a single compute node. However, for experiments that do not suffer from the scheduling overhead, satisfactory speedups are obtained. They are even superlinear in case the scheduling overheads disappear, due to the higher number of compute nodes and thus smaller number of threads to be scheduled on a compute node.

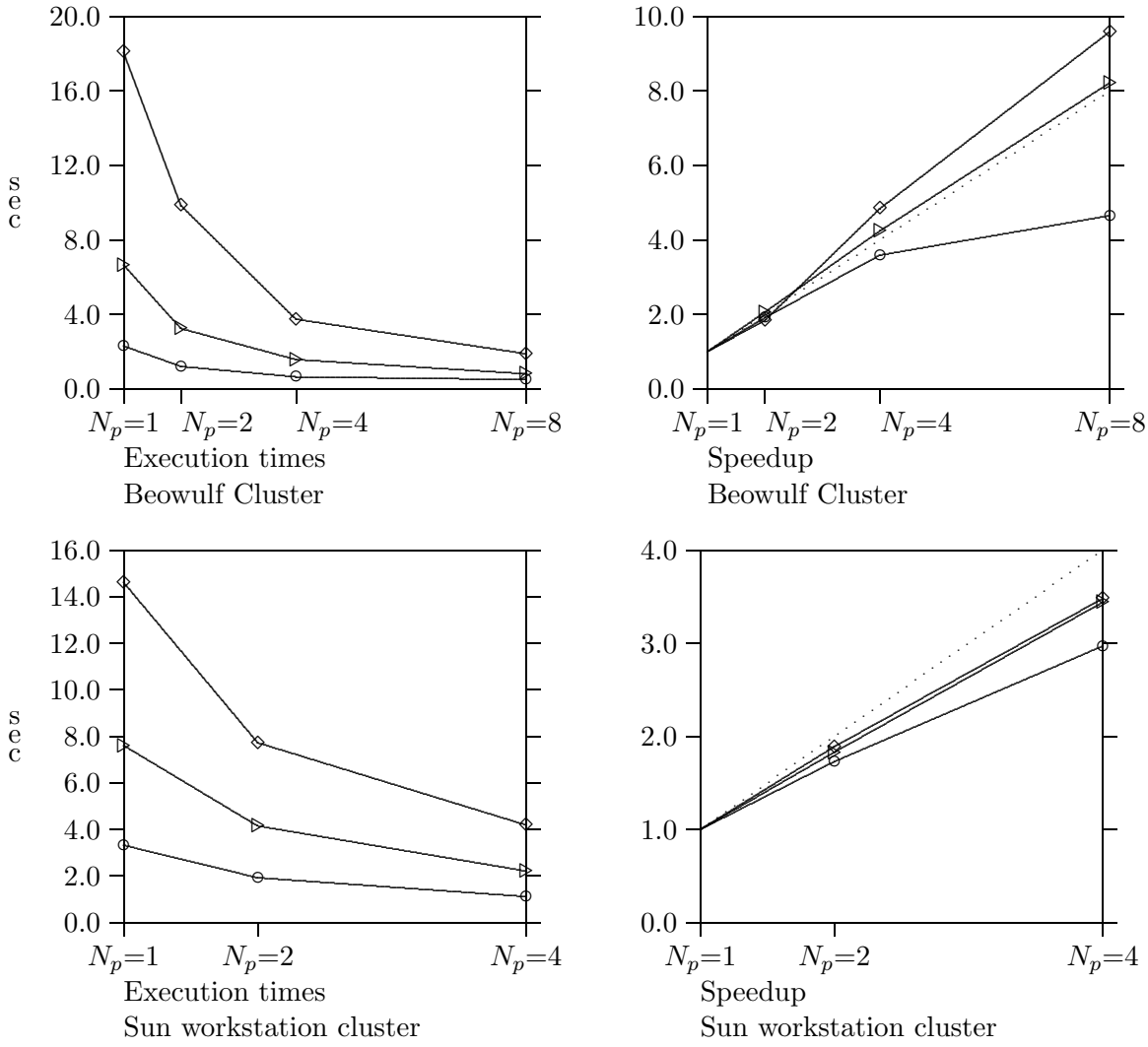
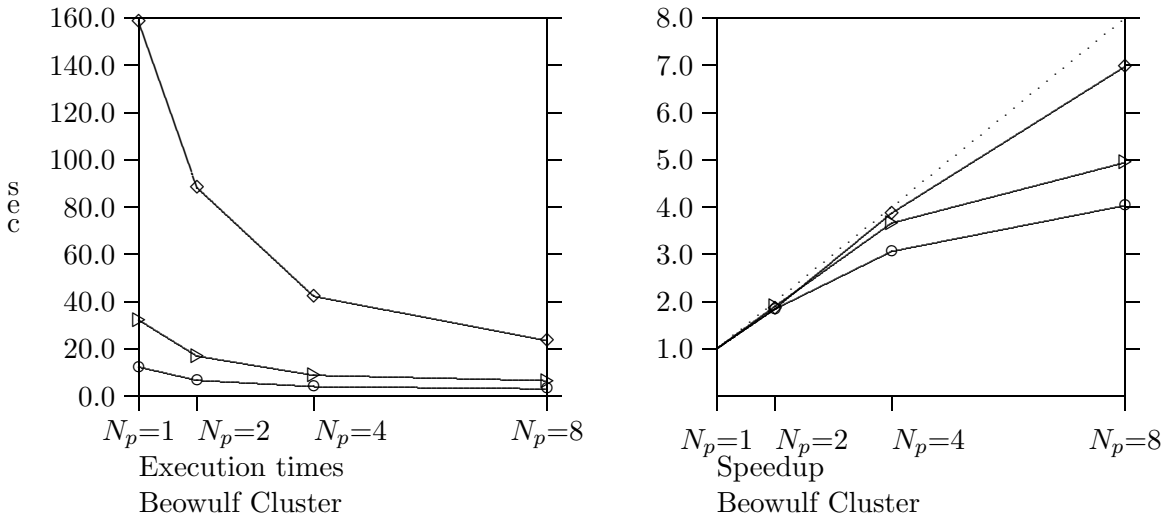


Figure 5-29: Performance of nested Benders decomposition—OpusJava version

For the performance evaluation of the JavaSymphony implementation, the active tree node objects are artificially augmented with additional work in order to reflect more realistically sized optimization problems with larger portfolios. Figure 5-30 shows the execution times and speedups on the Beowulf cluster. With increasing number of compute nodes, the overall communication effort increases and lowers the speedups. However, in case of larger scenario trees, the effect is smaller, and in particular for the 511-nodes problem a very good speedup is achieved.



○	63 tree nodes
▷	255 tree nodes
◇	511 tree nodes
—	asynchronous version
N_p	number of compute nodes
···	linear speedup

Figure 5-30: Performance of nested Benders decomposition—JavaSymphony version

Chapter 6

Conclusion

6.1 Summary

In this work, the question of how to efficiently implement high performance financial applications—in particular on parallel platforms—has been tackled at the levels of algorithms, the parallelization of these, and the programming tools employed. So it has been interpreted in the double sense of how to obtain an efficient parallel program as a result of an efficient implementation process. The performance of the code implemented has been evaluated on various platforms, including a massively parallel system, a vector supercomputer, a Linux SMP cluster, and a network of workstations. The resulting speedups demonstrate the potential of high performance computing for financial applications.

The programming model of a distributed active tree for the parallel and distributed implementation of tree structured algorithms including varying synchronization requirements, tree distributions, and scheduling strategies has been designed and implemented as a Java package on top of Java's threads and Java RMI. In addition, a path notation for the parallel formulation of algorithms operating on lattice structures has been specified as a Fortran 95 extension.

Procedures for pricing interest rate derivatives based on the Hull and White tree have been implemented in parallel in HPF+. A hybrid implementation of backward induction on clusters of SMPs combines distributed and shared memory parallelism via MPI and OpenMP. The parallel pricing kernel developed applies backward induction and nested Monte Carlo simulation and provides for a specification of path dependent instruments based on a history state. A class of instruments with limited path dependence has been identified, and the backward induction method has been generalized to allow for calculating their prices in linear time.

The nested Benders decomposition algorithm has been implemented in parallel in Java, both in a synchronous and in an asynchronous version, on top of a Java implementation of the distributed active tree model. In addition, the algorithm has been parallelized using OpusJava and a distributed active tree implemented on top of JavaSymphony. A test problem generator has been developed which builds wealth maximization and index tracking optimization models in a node-local formulation.

Parallelization is not the only source of performance improvement, the seek for performance starts with the selection of modeling techniques yielding structures which can be exploited—in particular with respect to parallel execution—by specialized algorithms. It continues with the choice or design, respectively, of such an algorithm, followed by an implementation which utilizes other sources of sequential performance optimization.

The combination of sequential and parallel performance optimization provides for synergy. In the case of generalized backward induction, the optimized sequential method is superior to the original version also with respect to its potential for parallelism.

Performance improvement is not an end in itself, but a means to support the user in achieving higher productivity. The requirement for short computation times must not sacrifice the quality of the data delivered, especially with respect to its purpose of supporting decision makers. In fact, early availability of a decision is a factor of productivity, and hence an argument for high performance computing.

6.2 Open Issues

A lot of issues are rather scratched on the surface than dealt with in depth and comprehensively and thus demand for further studies. As far as the level of parallel implementation is concerned, the programming support environment developed is an aid in such investigations by facilitating the production of experimental program versions.

In order to study the impact on the performance, parallel versions of the nested Benders decomposition algorithm can be varied in the following characteristics. The problem size is defined by the number of time stages, the number of successors of a single node, the number of financial contracts in a portfolio, and the number of local constraints. Various mappings occurring in this context have to be considered. The *decomposition* of the whole optimization

problem is described by a mapping from the set of node problems onto the set of subproblems, i.e., a subproblem can comprise more than one node problem. The *distribution* specifies the mapping of subproblems onto compute nodes of a parallel system. It can vary during runtime through dynamic rebalancing of the tree which includes the migration of subproblems. Parts of the tree which will not contribute to the final solution can be pruned. Distribution strategies can also be characterized by their objective, such as minimal communication and optimal load balancing. The *scheduling* includes the mapping of subproblems onto threads (a thread can be responsible for more than one subproblem), the mapping of threads onto compute nodes and onto physical processors of an SMP node, respectively, and the order of visits of subproblems by a thread.

Different synchronization schemes are possible and can also be combined. As an example, the coordination of compute nodes follows the asynchronous scheme, and the coordination of subproblems within a compute node the synchronous one. Likewise, subproblems handled by the same thread can be coordinated asynchronously, whereas subproblems handled by different threads interact in a synchronous manner. Furthermore, the overhead of buffers can be minimized, e.g., data structures of the linear program solver could be carefully employed as communication buffers.

A comparison with parallelizations of the nested Benders algorithm employing HPF, MPI, and OpenMP would yield further insights in the suitability of Java as a language for high performance computing.

The sequential decomposition procedure can be optimized through the deletion of irrelevant cuts of a subproblem, the prevention of resending identical solutions and cuts, including the optimal specification of the equality of cuts in terms of numerical accuracy. The performance of the decomposition algorithm can be enhanced also by a variety of optimized linear problem solving techniques, including “warm” starts.

At the level of the optimization model to be solved, a number of objective functions are interesting with respect to both numerical and performance results, including quadratic and nonlinear convex objectives and risk measures, e.g., variances, mean absolute deviation (MAD), and Value at Risk (VaR). Some of these need universal variables in the node problems and thus raise a challenge with respect to communication requirements.

The Monte Carlo simulation can be enhanced by a multitude of known variance reduction techniques which aim at sequential performance improvement, such as control variates, stratified sampling, and importance sampling. The pricing algorithms can be applied to multi-dimensional lattices, which, e.g., represent different foreign interest rates. A potential is seen in the development of pricing algorithms exploiting special cash-flow characteristics and the parallelization of these. Further subject to parallelization are approximative backward induction techniques as well as scenario tree generation procedures.

Numerical studies will expose the optimal choice of performance-critical parameters, e.g., the size of the time step and the number of paths in the Monte Carlo simulation, with respect to the accuracy of the result required. An optimal calibration of the lattice to market data is important in order to minimize model errors.

The user interface of the AURORA Financial Management System is subject to improvement via—possibly XML based—input languages both for the specification of financial instruments and the optimization model. On the output side, some visualization component is desirable, illustrating both the solution process and the results of the optimization.

The study and development of object-oriented models of the application domain including financial products, asset price models, and algorithms is a real challenge. The integration of pricing and optimization procedures into (remotely accessible) problem solving environments, e.g., in the form of web services, is a further issue.

The Fortran 95 path notation is to be formally specified and implemented within a compiler or preprocessor. The distribute active tree model can be generalized to arbitrary graph structures in a straightforward way. Finally, the specification of a mapping from UML state diagrams onto (Java) code and automatic code generation would represent a means for reducing the time-consuming and error-prone programming effort of synchronizing concurrent processes and would pass the way to model driven application development for financial applications.

Bibliography

[1] E. Alba, editor. *Parallel Metaheuristics*. Parallel and Distributed Computing. Wiley, Hoboken, New Jersey, USA, 2005.

[2] T.W. Archibald, C.S. Buchanan, K.I.M. McKinnon, and L.C. Thomas. Nested Benders decomposition and dynamic programming for reservoir optimisation. *Journal of the Operational Research Society*, 50(5):468–479, 1999.

[3] K.A. Ariyawansa 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.

[4] S. Beer. What has cybernetics to do with operational research? *Operational Research Quarterly*, 10:1–21, 1959.

[5] J.F. Benders. Partitioning procedures for solving mixed-variable programming problems. *Numer. Math.*, 4:238–252, 1962.

[6] S. Benkner. HPF+—High Performance Fortran for advanced scientific and engineering applications. *Future Generation Computer Systems*, 15(3):381–391, 1999.

[7] S. Benkner. Optimizing irregular HPF applications using halos. *Concurrency: Practice and Experience*, 12:137–155, 2000.

[8] S. Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, 2000.

[9] S. Benkner and T. Brandes. Initial specification of HPF extensions for clusters of SMPs. Technical Report ADVANCE Deliverable 4a, University of Vienna, August 2000.

[10] S. Benkner, L. Halada, and M. Lucka. Parallelization strategies of three-stage stochastic program based on the BQ method. In R. Trobec, P. Zinterhof, M. Vajtersic, and A. Uhl, editors, *Parallel Numerics'02, Theory and Applications*, pages 77–86, October 2002.

[11] S. Benkner, E. Laure, and H. Zima. HPF+: An extension of HPF for advanced industrial applications. Technical Report TR99-01, Institute for Software Technology and Parallel Systems, University of Vienna, 1999.

[12] A. Berger, J. Mulvey, and R.J. Vanderbei. Solving multistage stochastic programs with tree dissection. Technical report, Statistics and Operations Research, Princeton University, 1991.

[13] D.P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, New York, 1982.

[14] D. Bienstock and J.F. Shapiro. Optimizing resource acquisition decisions by stochastic programming. *Management Science*, 34:215–229, 1988.

[15] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA machines. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, November 2000.

[16] J.R. Birge. Decomposition and partitioning methods for multistage stochastic linear programs. *Operations Research*, 33:989–1007, 1985.

[17] J.R. Birge, C.J. Donohue, D.F. Holmes, and O.G. Svintsitski. A parallel implementation of the nested decomposition algorithm for multistage stochastic linear programs. *Mathematical Programming*, 75(2):327–352, 1996.

[18] F. Black, E. Derman, and W. Toy. A one-factor model of interest rates and its application to treasury bond options. *Financial Analysts Journal*, Jan-Feb:33–39, 1990.

[19] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–654, 1973.

[20] J. Blomvall. A multistage stochastic programming algorithm suitable for parallel computing. *Parallel Computing*, 29:431–445, 2003.

[21] P.P. Boyle. Options: A Monte Carlo approach. *Journal of Financial Economics*, 4:323–38, 1977.

[22] P.P. Boyle, M. Broadie, and P. Glasserman. Monte Carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21:1267–1321, 1997.

[23] S.P. Bradley and D.B. Crane. A dynamic model for bond portfolio management. *Management Science*, 19:139–151, 1972.

[24] T. Brandes and S. Benkner. Hierarchical data mapping for clusters of SMPs. Technical Report ADVICE-2 Deliverable 4a, GMD, June 2000.

[25] M.J. Brennan, E.S. Schwartz, and R. Lagnado. Strategic asset allocation. *Journal of Economic Dynamics and Control*, 21:1377–1403, 1997.

[26] G. Campolieti and R. Makarov. Parallel lattice implementation for option pricing under mixed state-dependent volatility models. In *IEEE Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pages 170–176, May 2005.

[27] F. Cappello and D. Etieble. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, 2000.

[28] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: Data parallel extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.

[29] Y. Censor and S.A. Zenios. *Parallel Optimization. Theory, Algorithms and Applications.* Oxford University Press, 1997.

[30] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic data distributions in Vienna Fortran. In *Proceedings of the Supercomputing '93 Conference*, Portland, Oregon, November 1993.

[31] B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with features for locality control. In *Proceedings of the ECMWF Workshop "Towards Teracomputing—The Use of Parallel Processors in Meteorology"*, Reading, England, November 1998.

[32] L. Clewlow and C. Strickland. *Implementing derivative Models*. John Wiley & Sons, 1998.

[33] P.D. Coddington and A.J. Newell. JAPARA—A Java parallel random number generator library for high-performance computing. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, April 2004.

[34] A. Consiglio and S.A. Zenios. Designing portfolios of financial products using integrated simulation and optimization models. *OR*, 47:195–20, 1999.

[35] Cornell Theory Center. *Glossary of High Performance Computing Terms*, March 2006. `www.tc.cornell.edu/NR/shared/Edu/Glossary`.

[36] J. Cox, J.E. Ingersoll Jr., and S. Ross. A theory of terms of interest rates. *Econometrica*, 53:385–407, 1985.

[37] J. Cox, S. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

[38] CPLEX Optimization, Inc., Incline Village, NV. *Using the CPLEX Callable Library*, 1995.

[39] H. Dahl, A. Meeraus, and S.A. Zenios. Some financial optimization models: I. risk management, II. financial engineering. In Zenios [158], pages 3–36, 37–71.

[40] G.B. Dantzig. Linear programming under uncertainty. *Management Science*, 1:197–206, 1955.

[41] G.B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[42] R. Das and J. Saltz. A manual for PARTI runtime primitives—Revision 2. Internal research report, University of Maryland, December 1992.

[43] M.A.H. Dempster, editor. *Stochastic Programming*. Academic Press, New York, 1980.

[44] M.A.H. Dempster. On stochastic programming II: Dynamic problems under risk. *Stochastics*, 25:15–42, 1988.

[45] M.A.H. Dempster and A.M. Ireland. A financial expert decision support system. In G. Mitra, editor, *Mathematical Models for Decision Support*, NATO ASI Series F48, pages 631–640. Springer, Heidelberg, 1988.

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

[47] E. Dockner and H. Moritsch. Pricing constant maturity floaters with embedded options using Monte Carlo simulation. In A.M. Skulimowski, editor, *Proceedings of the 23rd Meeting of the EURO Working Group on Financial Modelling*, Krakow, 1999. Progress & Business Publishers.

[48] J. Dupacova. Multistage stochastic programs: The state-of-the-art and selected bibliography. *Kybernetika*, 31:151–174, 1995.

[49] Y. Ermoliev and R.J.-B. Wets, editors. *Numerical Techniques for Stochastic Optimization*. Springer, Berlin, 1988.

[50] 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.

[51] T. Fahringer, P. Blaha, A. Hössinger, J. Luitz, E. Mehofer, H. Moritsch, and B. Scholz. Development and performance analysis of real-world applications for distributed and parallel architectures. *Concurrency and Computation: Practice and Experience*, 13:841–868, 2001.

[52] T. Fahringer, M. Geissler, G. Madsen, H. Moritsch, and C. Seragiottio. Semi-automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *Proceedings of the 11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP2003)*, Genoa, Italy, February 2003.

[53] T. Fahringer and A. Jugravu. JavaSymphony: New directives to control and synchronize locality, parallelism, and load balancing for cluster and GRID-computing. In *Proceed-*

ings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI '02), pages 8–17, Seattle, Washington, USA, 2002. ACM Press.

[54] T. Fahringer, A. Jugravu, B. Di Martino, S. Venticinque, and H. Moritsch. On the evaluation of JavaSymphony for cluster applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, Illinois, September 2002.

[55] T. Fahringer, A. Pozgaj, J. Luitz, and H. Moritsch. Evaluation of P^3T+: A performance estimator for distributed and parallel applications. In *IEEE Proceedings of the International Parallel and Distributed Processing Symposium*, Cancun, Mexico, 2000.

[56] T. Fahringer, K. Sowa, P. Czerwinski, J. Luitz, and H. Moritsch. On using SPiDER to examine and debug real-world data-parallel applications. In *Proceedings of the 6th International Conference on Parallel Computing Technologies, PACT-2001*, Novosibirsk, Russia, September 2001.

[57] Österreichische Gesellschaft für Operations Research, March 2006. `www.oegor.at`.

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

[59] A.V. Gerbessiotis. Architecture independent parallel binomial tree option price valuations. *Parallel Computing*, 30:301–316, 2004.

[60] H.M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität, Bonn, 1989.

[61] H.M. Gerndt and H. Moritsch. Parallelization for multiprocessors with memory hierarchies. In H.P. Zima, editor, *Parallel Computation. First International ACPC Conference 1991*, Lecture Notes in Computer Science, No. 591, pages 89–101. Springer, 1992.

[62] M. Giles and P. Glasserman. Smoking adjoints: Fast evaluation of Greeks in Monte Carlo calculations. Technical Report NA-05/15, Numerical Analysis Group, Oxford University, July 2005.

[63] L.J. Gittman. *Principles of Managerial Finance*. Pearson/Addison-Wesley, Boston, 11th edition, 2006.

[64] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, New York, 2004.

[65] J. Gondzio and R. Kouwenberg. High performance computing for asset liability management. *Operations Research*, 49:879–891, 2001.

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

[67] S.L. Graham, M.Snir, and C.A. Patterson, editors. *Getting Up to Speed: The Future of Supercomputing*. The National Academic Press, 2005.

[68] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of SC 2000: High Performance Networking and Computing Conference*, 2000.

[69] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[70] M. Hitz, G. Kappel, E. Kapsammer, and W. Retschitzegger. *UML@Work*. dpunkt.verlag, Heidelberg, 2005.

[71] R. Hochreiter, C. Wiesinger, and D. Wozabal. Large-scale computational finance applications on the Open Grid service environment. In *Advances in Grid Computing—EGC 2005: European Grid Conference*, Lecture Notes in Computer Science, No. 3470, pages 891–899. Springer, 2005.

[72] P. Hoel, S. Port, and Ch. Stone. *Introduction to Stochastic Processes*. Houghton-Mifflin, Boston, 1972.

[73] K. Hoyland and S.W. Wallace. Generating scenario trees for multistage decision problems. *Management Science*, 47:295–307, 2001.

[74] K. Huang and R.K. Thulasiram. Parallel algorithm for pricing american asian options with multi-dimensional assets. In *IEEE Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pages 177–185, May 2005.

[75] J.C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, 6th edition, 2006.

[76] J.C. Hull and A. White. One factor interest rate models and the valuation of interest rate derivative securities. *Journal of Financial and Quantitative Analysis*, 28:235–254, 1993.

[77] J.M. Hutchinson and S.A. Zenios. Financial simulations on a massively parallallel connection machine. *The International Journal of Supercomputer Applications*, 5(2):27–45, 1991.

[78] P. Hutchinson and M. Lane. A model for managing a certificate of deposit portfolio under uncertainty. In Dempster [43], pages 473–496.

[79] IBM Corporation, Kingston, New York. *Optimization Subroutine Library (OSL) Release 2*, 4th edition, 1992.

[80] IEEE/ANSI. *POSIX Threads Extensions*. IEEE/ANSI 1003.1c-1995.

[81] ISO. *Fortran 90 Standard*, 1991. ISO/IEC 1539:1991 (E).

[82] ISO. *Fortran 95 Standard*, 1997. ISO/IEC 1539:1997.

[83] F. Jamshidian. Forward induction and construction of yield curve diffusion models. *Journal of Fixed Income*, 1:62–74, 1991.

[84] J.E. Ingersoll Jr. *Theory of Financial Decision Making. Studies in Financial Economics*. Rowman & Littlefield, 1987.

[85] P. Kall and S.W. Wallace. *Stochastic Programming*. Wiley & Sons, Chichester, 1994.

[86] 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.

[87] A. Knapp and S. Merz. Model checking and code generation for uml state machines and collaborations. In G. Schellhorn and W. Reif, editors, *Proceedings of the 5th Workshop on Tools for System Design and Verification(FM-TOOLS 2002)*, Report 2002-11, Reisensburg, Germany, July 2002. Institut für Informatik, Universität Augsburg.

[88] R. Kouwenberg. Scenario generation and stochastic programming models for asset liability management. *European J. of Operations Research*, 134:279–292, 2001.

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

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

[91] 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*, pages 590–597, Delft, The Netherlands, April 2000. Imperial College Press.

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

[93] E. Laure. OpusJava: A Java framework for distributed high performance computing. *Future Generation Computer Systems*, 18(2):235–251, October 2001.

[94] 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.

[95] E. Laure, E. Mehofer, H. Moritsch, V. Sipkova, and A. Świętanowski. HPF in financial management under uncertainty. Technical Report AURORA TR1999-21, Vienna University, October 1999.

[96] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous computing with data parallel tasks. *Parallel Processing Letters*, 9(2):275–289, June 1999.

[97] E. Laure and H. Moritsch. A high performance decomposition solver for portfolio management problems in the AURORA financial management system. Technical Report TR01-13, Institute for Software Science, University of Vienna, October 2001.

[98] E. Laure and H. Moritsch. Portable parallel portfolio optimization in the AURORA financial management system. In *Proceedings of the SPIE ITCom 2001 Conference: Commercial Applications for High-Performance Computing*, Denver, Colorado, August 2001.

[99] D. Lea. *Concurrent Programming in Java.* Addison-Wesley, Reading, Mass., 1997.

[100] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP—A programming model for SMP clusters. In *Eight International Workshop on Compilers for Parallel Computers*, pages 229–238, Aussois, France, January 2000.

[101] M.-P. Leong, C.-C. Cheung, C.-W. Cheung, P.P.M. Wan, I.K.H. Leung, W.M.M. Yeung, W.-S. Yuen, K.S.K. Chow, K.-S. Leung, and P.H.W. Leong. CPE: A parallel library for financial engineering applications. *IEEE Computer*, 38(10):70–77, 2005.

[102] R. Levkovitz and G. Mitra. Solution of large-scale linear programs: A review of hardware, software and algorithmic issues. In T.A. Ciriani and R.C. Leachman, editors, *Optimization in Industry*, pages 139–171. John Wiley, Chichester, 1993.

[103] J.X. Li and G.L. Mullen. Parallel computing of a quasi-Monte Carlo algorithm for valuing derivatives. *Parallel Computing*, 26:641–653, 2000.

[104] M. Lobosco, C. Amorim, and O. Loques. Java for high-performance network based computing: a survey. *Concurrency and Computation: Practice and Experience*, 14:1–31, 2002.

[105] I.J. Lustig, R.E. Marsten, and D.F. Shanno. Interior point methods for linear programming. *ORSA J. on Computing*, 6:1–14, 1994.

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

[107] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 1.1*, June 1995.

[108] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[109] M. Mizuno, G. Singh, and M. Neilsen. A structured approach to develop concurrent programs in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proceedings of UML 2000—The Unified Modeling Language. Advancing the Standard. Third International Conference*, pages 451–465, York, UK, October 2000. Springer.

[110] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[111] H. Moritsch. A backward induction algorithm for pricing instruments with limited path dependence. Technical Report AURORA TR2003-05, Vienna University, 2003.

[112] H. Moritsch. A programming model for tree structured parallel and distributed algorithms. In *Proceedings of the International Conference on Parallel Computing (ParCo2003)*, Dresden, Germany, September 2003.

[113] H. Moritsch and S. Benkner. High performance numerical pricing methods. *Concurrency and Computation: Practice and Experience*, 14:665–678, 2002.

[114] H. Moritsch, G. Ch. Pflug, and E. Dockner. Test problem generation for multiperiod optimization in the AURORA financial management system. Technical Report AURORA TR2001-14, Vienna University, November 2001.

[115] H. Moritsch and G.Ch. Pflug. Java implementation of asynchronous parallel nested optimization algorithms. In *Third Workshop on Java for High Performance Computing*, 2001. Sorrento, Italy, June.

[116] H. Moritsch and G.Ch. Pflug. Polynomial algorithms for pricing path dependent contracts. Technical Report AURORA TR2002-32, Vienna University, December 2002.

[117] 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.

[118] H.W. Moritsch and G.Ch. Pflug. Using a distributed active tree in Java for the parallel and distributed implementation of a nested optimization algorithm. In *Proceedings of the 5th Workshop on High Performance Scientific and Engineering Computing with Applications (HPSECA-03)*, Kaohsiung, Taiwan, ROC, October 2003.

[119] J.M. Mulvey and W.T. Zimba. Asset and liability management systems for long-term investors: discussion of the issues. In J.M. Mulvey and W.T. Ziemba, editors, *Worldwide Asset Liability Management*. Cambridge University Press, 1998.

[120] S.S. Nielsen and S.A. Zenios. Scalable parallel Benders decomposition for stochastic linear programming. *Parallel Computing*, 23:1069–1088, 1997.

[121] The OpenMP Forum. *OpenMP Fortran Application Program Interface, Version 1.1*, November 1999. `www.openmp.org`.

[122] K. Pang and C. Strickland. Pricing interest rate exotics using term structure consistent short rate trees. In C. Strickland L. Clewlow, editor, *Exotic Options: The State Of The Art*. International Thomson Business Press, London, 1997.

[123] Giorgio Pauletto. Parallel Monte Carlo methods for derivative security pricing. In *Numerical Analysis and Its Applications, Second International Conference, NAA 2000*, Lecture Notes in Computer Science, No. 1988, pages 650–657. Springer, 2000.

[124] M.V.F. Pereirea and L.M. Pinto. Multi-stage stochastic optimization applied to energy planning. *MPr*, 52:359–375, 1991.

[125] S.C. Perry, R.H. Grimwood, D.J. Kerbyson, E. Papaefstathiou, and G.R. Nudd. Performance optimization of financial option calculations. *Parallel Computing*, 26:623–639, 2000.

[126] 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.

[127] G. Ch. Pflug. Optimal scenario tree generation for multiperiod financial planning. *Mathematical Programming*, 89:251–271, 2001.

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

[129] G.Ch. Pflug, A. Świętanowski, E. Dockner, and H. Moritsch. The AURORA financial management system: Model and parallel implementation design. *Annals of Operations Research*, 99:189–206, 2000.

[130] G.Ch. Pflug and A. Świętanowski. The AURORA financial management system documentation. Technical Report AURORA TR1998-09, Vienna University, June 1998.

[131] G.Ch. Pflug and A. Świętanowski. Dynamic asset allocation under uncertainty for pension fund management. *Control and Cybernetics*, 28(4):755–777, 1999.

[132] G.Ch. Pflug and A. Świętanowski. Asset-liability optimization for pension fund management. In *Operations Research Proceedings 2000*, pages 124 – 135. Springer, 2000.

[133] S. Pichler. *Bewertung von Zahlungsströmen mit variabler Verzinsung*. Vienna, September 1998. Habilitationsschrift.

[134] R. Prodan, T. Fahringer, F. Franchetti, M. Geissler, G. Madsen, and H. Moritsch. On using ZENTURIO for performance and parameter studies on clusters and grids. In *Proceedings of the 11-th Euromicro Conference on Parallel Distributed and Network based Processing (PDP2003)*, Genoa, Italy, February 2003.

[135] A. Ruszczyński. A regularized decomposition method for minimizing a sum of polyhedral functions. *Mathematical Programming*, 35:309–333, 1986.

[136] A. Ruszczyński. An augmented Lagrangian decomposition method for block diagonal linear programming problems. *Operations Research Letters*, 8:287–294, 1989.

[137] A. Ruszczyński. Parallel decomposition of multistage stochastic programming problems. *Mathematical Programming*, 58:201–228, 1993.

[138] A. Ruszczyński. Decomposition methods in stochastic programming. *Mathematical Programming*, 79:333–353, 1997.

[139] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[140] E.S. Schwartz. The valuation of warrants: Implementing a new approach. *Journal of Financial Economics*, 4:79–94, 1977.

[141] Silicon Graphics Inc. *MIPSpro Power Fortran 77 Programmer's Guide: OpenMP Multiprocessing Directives*, 1999. Document 007-2361-007.

[142] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[143] L. Somlyody and R.J.-B. Wets. Stochastic optimization models for lake eutrophication management. *Operations Research*, 36:660–681, 1988.

[144] Spezialforschungsbereich (SFB) AURORA "Advanced Models, Applications and Software Systems for High Performance Computing", University of Vienna. *Report on the First Research Period*, October 1999.

[145] Thomas Sterling. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge, MA, USA, 2001.

[146] U.H. Suhl. MOPS - Mathematical Optimization System. *European J. of Operations Research*, 72:312–322, 1994.

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

[148] H. Troung, T. Fahringer, G. Madsen, A. Malony, H. Moritsch, and S. Shende. On using SCALEA for performance analysis of distributed and parallel programs. In *Proceedings of the 9th IEEE/ACM High-Performance Networking and Computing Conference, SC'2001*, Denver, Colorado, November 2001.

[149] E.P.K. Tsang and S. Martinez-Jaramillo. Computational finance. *IEEE Computational Intelligence Society Newsletter*, pages 8–13, August 2004.

[150] C. van Reeuwijk, A. van Gemund, and H. Sips. Spar: A set of extensions to Java for scientific computation. *Concurrency and Computation: Practice and Experience*, 15(3-5):277–297, 2003.

[151] R.J. Vanderbei. LOQO: An interior point code for quadratic programming. Technical Report SOR-94-15, School of Engineering and Applied Science, Departament of Civil Engineering and Operations Research, Princeton University, 1994.

[152] H. Vladimirou and S.A. Zenios. Scalable parallel computations for large-scale stochastic programming. *Annals of Operations Research*, 90:87–129, 1999.

[153] Wikipedia. *"Finance"*, March 2006. `en.wikipedia.org/wiki/Finance`.

[154] 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.

[155] S.J. Wright. *Primal Dual Interior-Point Methods*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[156] D. Yang and S.A. Zenios. A scalable parallel interior point algorithm for stochastic linear programming and robust optimization. Technical Report 95-07, Departament of Public and Business Administration, University of Cyprus, February 1995.

[157] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency and Computation: Practice and Experience*, 10(11-13):825–836, 1998.

[158] S. Zenios, editor. *Financial Optimization*. Cambridge University Press, 1993.

[159] S.A. Zenios. Parallel Monte Carlo simulation of mortgage-backed securities. In Zenios [158], pages 325–343.

[160] S.A. Zenios. High-performance computing in finance: The last ten years and the next. *Parallel Computing*, 25:2149–2175, 1999.

[161] G. Zhong. A numerical study of one-factor interest rate models. Master's thesis, Graduate Department of Computer Science University of Toronto, 1998.

[162] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran—A language specification. Internal Report 21, ICASE, 1992.

[163] H.P. Zima and B.M. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesely, 1990.