High Performance Numerical Pricing Methods *

Hans Moritsch[1] and Siegfried Benkner[2]

[1]Department of Business Studies
University of Vienna, Brünner Strasse 72, A-1210 Vienna, Austria
moritsch@finance2.bwl.univie.ac.at

[2]Institute for Software Science
University of Vienna, Liechtensteinstrasse 22, A-1090, Vienna, Austria
sigi@par.univie.ac.at

1 The Aurora Financial Management System

The *Aurora Financial Management System* developed at the University of Vienna is a decision support tool for portfolio and asset liability management. An investor chooses a portfolio of various assets or asset classes, in such a way that some objective [9], including a risk measure, is maximized, subject to uncertainty of future markets' development and additional constraints (e.g. budget restrictions). The system is based on pricing models for financial instruments, and a multivariate Markovian birth-and-death model for liabilities. The core of the system is a large scale linear optimization problem whose solution is the main outcome of the tool.

In this paper we discuss the parallelization of a major computational kernel from this system utilizing HPF+, an extended version of HPF. In Section 1.1 we give a brief overview of numerical pricing methods. In Section 2 we describe the parallelization of the backward induction algorithm using HPF+. This is followed by a brief description of proposed HPF extensions for clusters of SMPs in Section 3. Section 4 presents experimental results of the kernel on a Beowulf cluster and on a vector parallel supercomputer, followed by conclusions in Section 5.

1.1 The Pricing Module

Since the investor in the financial planning problem bases his decision (whether to buy, sell or keep a contract) on the prices for different instruments, the financial management system contains a pricing module which determines prices at specific times and states of the economic environment. It computes prices of different classes of financial instruments. In particular it performs the pricing of derivative and interest rate dependent products. A *derivative* is a financial instrument whose value depends on other, so called underlying securities (e.g. stock options). We concentrate on the pricing of interest rate dependent products, whose payments depend on actual or past interest rates (e.g. variable coupon bonds). The pricing problem can be stated as follows: what is the price today of an instrument which will pay some cash flows in the future, which depend the development of interest rates? For simple cases analytical formulas are available, but in general numerical techniques have to be applied.

Products, whose cash flows depend on a value of a financial variable in the past (*path dependent* products), are priced with Monte Carlo simulation techniques (see [5],[7]). Instruments without path

*The work described in this paper was partially supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund and by NEC Europe Ltd. within the co-operation project ADVANCE of the C&C Research Laboratories, St. Augustin, Germany, with the University of Vienna. Published in: Proceedings of the 4th HPF User Group Meeting, Tokyo, Japan, 2000.

dependences can be priced with the more efficient backward induction algorithm (see [11]). Both methods are based on a discrete representation of a stochastic process.

The Hull and White trinomial tree for the One-factor Hull and White interest rate model (see [11]) describes the development of the short term interest rate, discrete in time (with time increment Δt) and interest rate value. Figure 1 shows an example for a Hull and White interest rate tree. Each node represents a state (time, rate for maturity Δt), and has three successor nodes, representing increasing, constant remaining, and decreasing interest rates. Arcs are labeled with transition probabilities p_{up}, p_{mid}, p_{down}. A state can be reached via more than one predecessors. This recombining property establishes a lattice structure. At each node the interest rates for different maturities for that state are calculated during the construction of the tree.

Every instrument is characterized by a stream of cash flows, which are paid by that instrument. Currently, we focus on the pricing of variable-coupon bonds and interest rate derivatives. The sequence of cash flows at discrete points in time is specified as a function of a reference interest rate, e.g. the LIBOR (London Inter-bank Offered Rate). For path dependent products the cash flows also depend on the past development of interest rates.

1.2 Numerical Pricing Methods

The numerical pricing algorithms are based on the *present value* of an instrument at a node, which is defined as a sum of discounted future cash flows.

The *Monte Carlo simulation* algorithm selects a number of paths in the Hull and White tree from the root node to some final node (see Figure 1). Along each path, it iteratively computes, backwards from the final node to the root node, the present value at each node, using the cash flow generated by the instrument along this path. For path dependent products, the cash flows depend on the interest rates at predecessor nodes. Discounting is performed using the interest rates along this path. The price corresponding to a specific path is defined as the present value at the root node. The price of the instrument is the mean present value over all selected paths.

The core step in the *backward induction* algorithm is the computation of the present value at a node through discounted cash flows and present values at all direct successor nodes. Thus it incorporates the information about the future of a node by propagating it backwards in time. Through the iterative application of this step starting at the final nodes, the algorithm computes the price of the instrument as the present value at the root node. This method can be employed only, if the future information used is fully determined by the state represented by the node. In the case of path dependent products, the future cash flows, are specific to the history of the state, hence the backward induction techniques is restricted to instruments without path dependence.

The computational complexity of the backward induction is $O(n^2)$, as opposed to $O(3^n)$ for the MC simulation, where n is the number of time steps.

For pricing a subclass of path dependent instruments, the computation effort can be significantly reduced. Our pricing method for these instruments evaluates much shorter paths than the Monte Carlo simulation. The improvement is based on the following property of certain path dependent instruments: the path dependence is *limited*, if the cash flows depend on the interest rates at previous times reaching back in history no longer than a certain period. The *length* of the path dependence is the maximum difference in time steps between the earliest time a cash flow depends on, and the time it is paid. These instruments can be priced using a method which combines backward induction with sampling of paths of the size of the dependence length at every node. The computational effort for pricing limited path dependent instruments thus is reduced to $O(3^{length})$.

2 Parallelization of the Backward Induction Algorithm

In this section we discuss the parallelization of one of the main computational kernels of the pricing module. The backward induction kernel has been parallelized based on HPF+ [3], an extended version of HPF, and compiled with the VFC compiler [2]. The HPF+ version of the backward induction algorithm is shown in Figure 2.

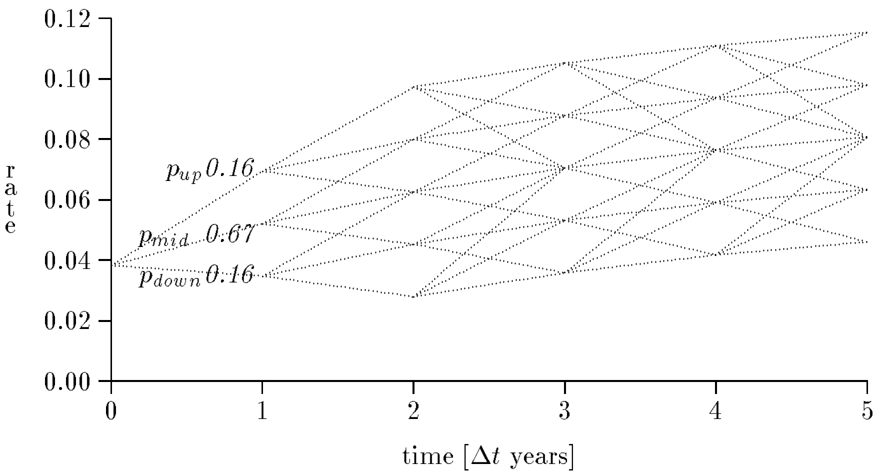

Figure 1: Hull and White interest rate tree

The main data structure of this kernel is the Hull and White tree which is represented by means of three arrays, `rate`, `probability` and `successor`. The arrays `rate` and `presentVal` are distributed in the first dimension, while the arrays `probability` and `successor` are distributed in the second dimension.

The algorithm comprises an outer time-step loop which is to be executed sequentially. Within each time-step, the subroutine `computeColumn()` is called in order to compute the present value at step `m`. Subroutine `computeColumn()` operates only on those parts of the tree that are actually required in time-step `m`. To enable an efficient parallelization with VFC, one-dimensional temporary arrays (`pvCol`, `pvColOld`, `rateCol`) have been introduced for storing the corresponding slices of the arrays `presentValue` and `rate` required in time-step `m`. These temporary arrays are distributed as well.

Moreover, on each processor the set of non-local elements of `pvColOld` required from its neighbors has been described by means of the HPF+ `HALO` attribute [1]. The halo information is computed at runtime and represented by means of the arrays `HH` and `HP`. The communication associated with the halo of array `pvColOld` is triggered explicilty by means of the HPF+ `UPDATE_HALO` directive. Because of the `HALO` attribute, no communication is generated by VFC within subroutine `computeColumn()`.

For the parallelization of the loop in subroutine `computeColumn()` an `INDEPENDENT` directive is required, because array `pvColOld` is indirectly accessed through the index array `successor`. Due to the indirect data accesses, the VFC compiler applies runtime parallelization techniques to parallelize the j-loop. Since the access patterns of all distributed arrays are invariant for all executions of `computeColumn()`, the HPF+ `REUSE` clause 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 `computeColumn()` is executed the first time. Moreover, due to the use of halos, no communication is required within `computeColumn()`.

According to the structure of the lattice, single element communication (between processors) takes place, which, depending on the computational complexity of the function `computeCoupon()`, can overwhelm the performance win gained by the parallization. However, this effect can be signficantly reduced on architectures with nodes consisting of several processors communicating through shared memory such as clusters of SMPs. Processors within the same node perform the exchange of elements of `pVColOld` through shared memory accesses, thus reducing the communication overhead. In order to optimize the HPF+ program for clusters of SMPs the hierarchical structure of clusters is specified by means of a processor mapping directive, which specifies a distribution of a processor arrangement to a nodes arrangement. These features are discussed in more detail in the next section.

An experimental evaluation of the backward induction kernel on a PC cluster as well as on a parallel vector supercomputer is presented in Section 4.

```
SUBROUTINE BW(HH, HP)
      INTEGER, DIMENSION(:)                          :: HH, HP
      INTEGER, DIMENSION(3,2*maxJ+1)                 :: successor
      DOUBLE PRECISION, DIMENSION(3,2*maxJ+1)        :: probability
      DOUBLE PRECISION, DIMENSION(2*maxJ+1, nrSteps+1) :: presentValue, rate
      DOUBLE PRECISION, DIMENSION(2*maxJ+1)          :: pVCol, pVColOld, rateCol
      ...
!HPF$ PROCESSORS P(number_of_processors())
!HPF+ NODES N(number_of_nodes())              ! abstract nodes arrangement of an SMP cluster
!HPF+ DISTRIBUTE P(BLOCK) ONTO N              ! processor mapping

!HPF$ DISTRIBUTE (*,BLOCK) ONTO P :: probability, successor
!HPF$ DISTRIBUTE (BLOCK,*) ONTO P :: presentValue, rate
!HPF$ DISTRIBUTE (BLOCK)   ONTO P :: pVCol, rateCol

!HPF+ DISTRIBUTE (BLOCK) ONTO P, HALO(HH(HP(I):HP(I+1)-1)) on P(I) :: pVColOld
      ...
      DO m = nrSteps, 1, -1                    ! time-step loop
         pVColOld(:) = presentValue(:,m+1)
         rateCol(:)     = rate(:,m)
!HPF+  UPDATE_HALO :: pVColOld
         CALL computeColumn()
         present_value(:,mm) = pVCol(:)
      END DO
 ...
CONTAINS

 SUBROUTINE computeColumn()

!HPF+ INDEPENDENT, NEW(c), ON HOME(pVCol(j)), REUSE
      DO j = 1, 2*maxJ+1
         c  = computeCoupon(rateCol(j))   ! compute coupon payment at node
         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 computeColumn
```

Figure 2: HPF+ Backward Induction Kernel

3 HPF Extensions for Clusters of SMPs

In this section we briefly discuss HPF extensions for clusters of SMPs which are currently being developed at the University of Vienna, Austria, in cooperation [4, 6] with GMD, St. Augustin, Germany. These extensions are currently being implemented within the HPF+ compiler VFC [2, 3], which we also used for an experimental evaluation of these features.

The main goal of these extensions is to optimize HPF for clusters of SMPs by enhancing the functionality of the HPF mapping mechanisms such that the hierarchical structure of SMP clusters can be taken into account. An HPF compiler can then adopt a hybrid parallelization strategy where distributed-memory parallelism based on message-passing, e.g. MPI [12], is exploited across the nodes of a cluster, while within SMP nodes shared-memory parallelism is exploited relying on multi-threading, e.g. OpenMP [13].

Two sets of HPF extensions for clusters of SMPs are being developed [4]. The first set of extensions has been designed in order to optimize existing HPF codes for clusters of SMPs without requiring the user to modify existing mapping directives. This is achieved by introducing the concept of abstract node arrays and mechanisms for specifying a mapping from an abstract node array to an abstract HPF processor array. Such mappings are referred to as *processor mappings* (see Example 1).

The second set of HPF extensions enables the specification of *hierarchical data mappings* in two steps. First, data arrays are mapped to the nodes of an SMP cluster by means of the usual HPF data distribution mechanisms. Such mappings are referred to as *inter-node data mappings*. Second, for the local data of nodes an *intra-node data mapping* may be specified which is utilized by the compiler to derive the work sharing for the threads executing concurrently on the processors within the nodes of a cluster.

Moreover, a new local extrinsic model, `MODEL = 'NODE_LOCAL'` is proposed, in order to allow the integration of OpenMP routines or other local routines that operate on the local data of nodes.

3.1 Abstract Nodes Arrangements and Processor Mappings

Processor mappings are used to describe the hierarchical structure of SMP clusters by mapping abstract processor arrangements to abstract node arrangements. Abstract node arrangements are specified by means of a new directive, the `NODES` directive, which declares one or more rectilinear *node arrangements*. For the specification of processor mappings a subset of the HPF data distribution mechanisms as provided by the `DISTRIBUTE` directive are utilized. The new intrinsic function `NUMBER_OF_NODES()` may be used to inquire about the total number of nodes used to execute the program.

Example 1: Node arrangements and processor mappings

```
!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 R(4,8)
!hpf$ nodes M(4)
!hpf$ distribute R(*, block) onto M

      real B(NB1,NB2)
!hpf$ distribute B(cyclic, block) onto B
```

(b)

Example (a) specifies the hierarchical structure of an SMP cluster, consisting of 8 nodes with 2 processors each. Example (b) defines an SMP cluster with 4 nodes each of which comprises 4x2 processors. ■

Processor mappings provide a simple means for optimizing existing HPF applications for SMP clusters. Using processor mappings the hierarchical structure of SMP clusters may be explicitly specified, without the need to change existing HPF directives. Based on a specified processor mapping an HPF compiler can adopt a cluster-specific parallelization strategy in order to efficiently exploit distributed-memory parallelism across the nodes of a cluster, and shared-memory parallelism within the nodes of a cluster.

3.2 Distributed-Memory versus Shared-Memory Parallelism

If a dimension of a processor arrangement is distributed by `BLOCK` or `GEN_BLOCK`, contiguous blocks of processors are mapped to the nodes in the corresponding dimension of the specified node arrangement. As a consequence of such a processor mapping, for all data array dimensions that are mapped to such a processor array dimension both distributed-memory parallelism and shared-memory parallelism may be exploited. However, if for a dimension of a processor arrangement an "*" is specified as distribution format, all abstract processors in this dimension are mapped to the same node of a node arrangement, and thus only shared-memory parallelism may be exploited.

In Example 1(a) both distributed and shared memory parallelism may be exploited for array `A`. In Example 1(b) shared memory parallelism may be exploited across the first dimension of array `B`, while across the second dimension of `B` both shared-memory and distributed-memory parallelism may be exploited.

3.3 An HPF Execution Model for SMP Clusters

If an HPF program contains `NODES` directives and processor mappings, the program is transformed by the VFC compiler as follows. First, VFC translates the original HPF program into an SPMD message passing node program by distributing data and work to the nodes of the cluster and inserting message-passing calls in order to communicate non-local data between nodes. In a second phase, those loops for which the compiler can determine that shared-memory parallelism can be exploited, are transformed in such a way that additional thread parallelism within the nodes is exploited by inserting corresponding OpenMP directives.

Example 2: VFC code generation under the cluster execution model

```
!hpf$ processors P(number_of_processors())
!hpf$ nodes N(number_of_nodes())                         ! nodes 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
-----------------------------------------------------------------------------------------
      ...                                                ! pseudo-code generate by VFC
      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
```

■

Currently the VFC compiler inserts simple OpenMP directives (e.g. `PARALLEL DO`) to exploit shared memory parallelism (see Example 2). More sophisticated code transformation strategies are currently under development in order to implement OpenMP work sharing mechanisms in accordance to the intra-node mapping derived from the HPF distribution.

The code generated by the VFC compiler, is compiled with an OpenMP Fortran compiler (currently pgf90 -mp) and linked with the MPI library. The resulting program is executed as an MPI program. Since each MPI process spawns parallel threads, process parallelism combined with thread parallelism. More precisely, an HPF program is executed by a set of parallel processes, each of which executes on a separate node of an SMP cluster within its own local address space. Each of these processes employs a set of threads which execute concurrently in the shared address space of a node. Process parallelism, data partitioning, and message-passing communication is utilized between the nodes of a cluster, while within a node additional parallelism is exploited by means of multiple threads concurrently executing in a shared address space.

4 Experimental Evaluation

For the performance experiments, we computed the price of a variable coupon bond with a maturity of 10 years on a one-day basis, resulting in a tree of length 3650 and height 2690. The backward induction kernel shown in Figure 2 has been parallelized with the VFC compiler and executed on a node of an NEC SX-5Be as well as on a Beowulf cluster consisting of 8 nodes, each equipped with two Pentium II (400MHz) processors. On the SX-5, the MPI/F90 code generated by the VFC compiler was compiled with the NEC sxf90, while on the PC cluster pgf90 from Portland Group Inc. was used as backend compiler of VFC.

The execution times measured on the SX-5 for the MPI kernel generated by VFC are shown in Figure 3. Special care has been taken in order to enable the sxf90 compiler used as VFC's backend to fully vectorize the loop in subroutine `computeCol()`, the function `computeCoupon()` has been inline-expanded. The performance and scaling behaviour is quite satisfactory.

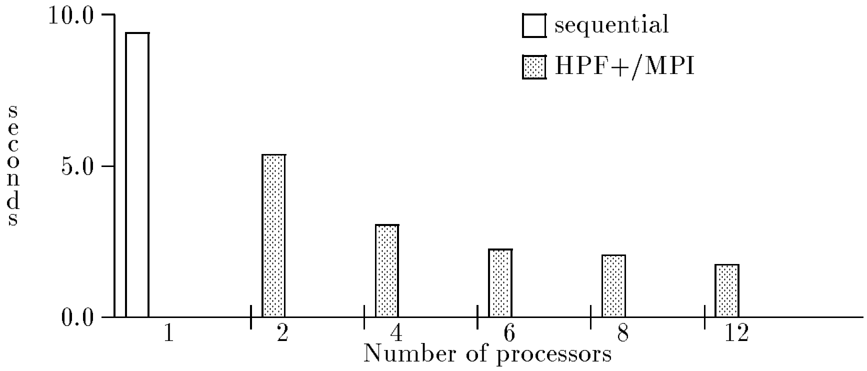

Figure 3: Execution times on NEC SX-5Be.

On the PC cluster two different parallel code version were generated by VFC from the HPF+ backward induction kernel. The first version generated is a pure MPI message passing code, while the second version employs a hybrid execution model utilizing MPI process parallelism across the nodes of the cluster and OpenMP thread parallelism within nodes. The pure MPI version of the generated code has been measured in two different ways which are labeld in Figure 4 MPI-P and MPI-N, respectively. MPI-P refers to the MPI code generated by the VFC and executed on 2 to 16 processors, where on each node both processors were utilized. In contrast, MPI-N refers to the MPI version of the generated code executed on 2 to 8 nodes, where on each node only one processor was utilized.

MPI-OpenMP refers to the hybrid parallel code generated by VFC which utilizes MPI accross nodes and OpenMP within nodes. This version was generated by VFC by taking into account the HPF extensions described in Section 3 (i.e. the `NODES` directive and processor mapping).

As Figure 4 shows, the MPI-OpenMP code generated by VFC outperforms the MPI-P variant significantly. One reason for this performance difference seems to be the communication overhead induced by MPI communication[1] within the nodes of the cluster. The times measured for the MPI-OpenMP and MPI-N variants were almost the same, however, the MPI-N version required twice as many nodes.

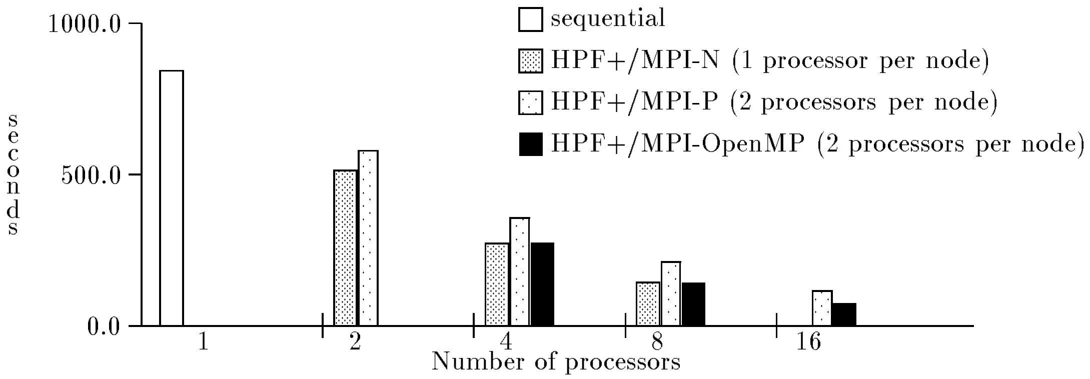

Figure 4: Execution times on Beowulf cluster with dual-processor nodes.

[1] The MPI version used on the PC cluster did not offer an optimized intra-node communication via shmem.

5 Conclusion

In this paper we presented the pricing component of the AURORA Financial Management System which is being developed at the University of Vienna. We have presented the parallelization of a numerical pricing kernel based on HPF with special extensions for clusters of SMPs as provided by the VFC [2] compiler. Our performance experiments indicate that these extensions can be utilized by an HPF compiler in order to optimize HPF programs for SMP clusters by adopting a hybrid parallelization strategy and execution model that combines distributed-memory parallelism with shared-memory parallelism.

References

[1] S.Benkner. "Optimizing Irregular HPF Applications Using Halos." *Concurrency: Practice and Experience*,12, pp.137-155, John Wiley & Sons Ltd, 2000.

[2] S. Benkner. "VFC: The Vienna Fortran Compiler", *Scientific Programming*, Vol.7, No.1, pp. 67-81, IOS Press, 1999.

[3] S. Benkner. "HPF+ – High Performance Fortran for Advanced Scientific and Engineering Applications", *Future Generation Computer Systems*, Vol 15 (3), pp. 381-391, 1999.

[4] S. Benkner and T. Brandes. Initial Specification of HPF Extensions for Clusters of SMPs. Technical Report, ADVANCE Deliverable 4a, University of Vienna, August 2000.

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

[6] T. Brandes and S. Benkner. Hierarchical Data Mapping for Clusters of SMPs. ADVICE-2 Deliverable 4a, GMD, St. Augustin, Germany, June 2000.

[7] L. Clelow, C. Strickland. *Implementing Derivative Models.* John Wiley & Sons, 1998.

[8] E. Dockner, H. Moritsch. *Pricing Constant Maturity Floaters with Embedded Options Using Monte Carlo Simulation.* Technical report AURORA TR1999-04.

[9] E. Dockner, H. Moritsch, G.Ch. Pflug and A. Swietanowski. *The AURORA financial management system: From Model Design to Implementation.* Technical report AURORA TR1998-08. University of Vienna, June 1998.

[10] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, Department of Computer Science, Rice University, January 1997.

[11] J. C. Hull. *Options, Futures, and Other Derivatives.* Prentice Hall, April 1997.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Vers. 1.1, June 1995. MPI-2: Extensions to the Message-Passing Interface, 1997.

[13] The OpenMP Forum. OpenMP Fortran Application Program Interface. Version 1.1. November 1999. `http://www.openmp.org/`.

[14] 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.

[15] G.Ch. Pflug and A. Świętanowski. Dynamic asset allocation under uncertainty for pension fund management. Technical report AURORA TR1998–15, Vienna University, 1998. To appear in *Control and Cybernetics*.

[16] G.Ch. Pflug and A. Świętanowski. Parallel decision support for financial management under uncertainty. Technical report AURORA TR1998–07, Vienna University, 1998. to appear in: PARCO, Elsevier, 1999.

[17] G.Ch. Pflug, A. Świętanowski. Selected parallel optimization methods for financial management under uncertainty. *Parallel Computing*, 2000.