Centro Svizzero di Calcolo Scientifico

C S C S

Swiss Scientific Computing Center

CSCS-TR-95-03

ETH Eidgenössische Technische Hochschule Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Tool-Supported Development of Parallel Application Kernels

C. Clémençon, K.M. Decker, A. Endo, V.R. Deshpande,
J. Fritscher, N. Masunda, A. Müller, R. Rühl,
W. Sawyer, B.J.N. Wylie and F. Zimmermann

April 1995

Technical Report

Tool-Supported Development of Parallel Application Kernels

CSCS TR-95-03

April 1995

Abstract: Our overall goal is to ease the parallelization of applications on distributed-memory parallel processors. Part of our team is implementing parallel kernels common to industrially significant applications using High Performance Fortran (HPF) and the Message Passing Interface (MPI). They are assisted in this activity by a second group developing an integrated tool environment consisting of a parallelization support tool, a parallel debugging tool, and a performance monitor and analyzer. From close interaction the design of the tools is inherently application-driven: application developers define requirements and evaluate prototypes of the tool environment.

This paper describes goals, achievements and perspectives of the project, illustrating with three application kernels how the tool environment assists in the parallelization process: development effort and resulting performance are discussed.

C. Clémençon **K. M. Decker** **A. Endo***
V. R. Deshpande **J. Fritscher** **N. Masuda***
A. Müller **R. Rühl** **W. Sawyer**
B. J. N. Wylie **F. Zimmermann***

Section of Research and Development (SeRD)
Swiss Scientific Computing Center (CSCS)
Swiss Federal Institute of Technology Zurich
Via Cantonale, CH-6928 Manno, Switzerland

* NEC European Supercomputer Systems, Swiss Branch

Technical Report

Table of Contents

List of Figures

List of Tables

1 Introduction

Distributed-memory parallel processor (DMPP) systems with scalable interconnection networks offer computing power and memory which can scale to meet foreseen demands. They are widely considered as enabling technology for progress in science, engineering, and business. Although there are a growing number of production applications running on DMPPs, they have not yet lived up to their expectations in the scientific and engineering communities.

The major reason for this short-coming is the current difficulty of programming scientific applications. Comfortable development environments, as found on sequential machines, don't yet exist on DMPPs. Since the level of abstraction from the complex hardware provided by the tools which are available is insufficient, the user suffers from uncomfortable, time-consuming, and error-prone programming. Despite considerable work on tool environments for parallel machines, user response has not been enthusiastic.

It is the goal of the Joint CSCS-ETH/NEC Collaboration in Parallel Processing to develop the integrated tool environment *Annai* [CDE+94, CEF+95] for convenient programming of DMPPs in a user-oriented and application-driven way. In this paper we demonstrate how the current *Annai* prototype eases effective parallelization of common application kernels.

The paper is organized as follows: in Section 2 we introduce our project from the application user's perspective. We present our development strategy for application kernels, an give an overview of *Annai*, and discuss the interaction of the partners in the collaboration. Section 3 elaborates on the parallelization of application kernels and discusses three examples: the iterative linear solver BiCGSTAB, an eigensolver for sparse linear systems, and the NAS Multigrid kernel. Not only the overall performance is discussed, but also the development effort for the parallel versions—a topic which is of primary concern for the application developer Finally we summarize our experiences using the tool environment and identify important functionality for parallelizing full-scale applications.

2 Project Overview

2.1 Development Strategy for Application Kernels

We first revisited the spectrum of applications in high-performance computing running on the facilities at CSCS and at other supercomputing centers. Application fields taken into consideration include combinatorial optimization, elementary particle physics, fluid dynamics, molecular dynamics, plasma physics, quantum chemistry, and structural mechanics. We then analyzed this application spectrum with respect to the computational methods used; among them, linear system and eigenvalue solvers for dense and sparse systems, fast Fourier transforms, random number generators, branch-and-bound search algorithms, and algorithms for dynamic finite-element mesh refinement.

This list is clearly only partial, but its contents cover a very wide range of applications. Currently there are no public-domain parallel libraries available to solve the above problems, with the possible exception of the ScaLAPACK [CDPW94] library which is currently under development. Thus our efforts are currently concentrated on implementing libraries for these algorithmic classes, and incorporating them into applications from the previously mentioned fields. Three important codes in this development are presented in this paper.

From the developer's perspective portability of code is also an important issue, as yet insufficiently addressed. A developer does not want to be bound to a particular platform and compiler, which may disappear before the useful life of the application has ended. The immediate solution is to adhere to accepted standards wherever possible. Thus we base our approach on the defined specifications for High Performance Fortran (HPF) [HPF93] and a Message Passing Interface (MPI) [MPI94].

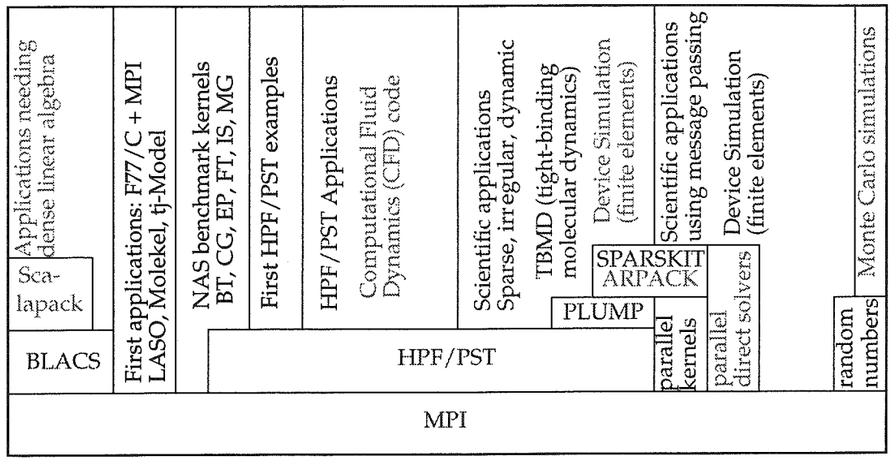


Figure 1: *Our efforts originated with an in-house implementation of MPI and a design of the tool environment (represented in magenta). First several NAS benchmark kernels were parallelized with MPI and using an older data-parallel extended Fortran, for performance comparison, along with several small applications in MPI alone (shown in blue). Our recent efforts (shown in red) include development of the PLUMP library, parallelization of certain solvers from SPARSKIT, a collaboration on the MPI-BLACS, and parallel random number generators. In the future (shown in green), we will parallelize eigensolvers from ARPACK, collaborate on the development of sparse direct solvers, and implement several full-scale applications in extended High Performance Fortran using existing libraries.*

The goals of our project go well beyond the examples presented in this paper. An overview of the parallelization work being performed is illustrated in Figure 1. At the lowest level of parallel programming is either F77 or C with explicit MPI message passing. At a higher level we try to formulate applications in data-parallel HPF, a task widely recognized to be difficult for unstructured applications [SMC90, CZM94]. In such cases we define and utilize extensions to HPF. We have previously compared the performance of NAS benchmark kernels programmed with explicit message passing to versions parallelized using Oxygen [Rüh92, CDE+94], the base of our current Parallelization Support Tool (PST). In addition, several small applications were implemented with MPI and their results discussed [FHM+94]. More recently, we have parallelized certain routines from SPARSKIT [Saa90] and LAPACK [ABD+92] in HPF/PST.

We believe that future users of parallel machines will rely heavily on public-domain libraries, as they have in the past with vector and scalar machines. Such libraries are a result of collaborations of many institutions. We therefore support the development of ScaLAPACK for dense linear algebra problems, and participate actively in the porting of the underlying communication mechanism BLACS [DW95] to MPI.

While continuing to parallelize commonly used application kernels, we are also parallelizing several applications in the fields of molecular dynamics, device simulation and computational fluid dynamics with *Annai* and the parallel libraries already available, and are currently planning work on several others.

2.2 The *Annai* Integrated Tool Environment

A number of the tools typically available on DMPPs were investigated and user experience with them evaluated: ParAide [Int93] for the Intel Paragon, TotalView [BBN94] available on the Cray Research T3D and other systems, and Prism [ABJ+91] for the Connection Machine from Thinking Machines Corporation. While some high-level programming features are provided, many essential features are missing, and, crucially, none of them support portable application development with the standard languages and machine interfaces expected by the user community.

There are three component tools within the *Annai* environment: a Parallelization Support Tool (PST), a Parallel Debugging Tool (PDT), and a Performance Monitor and Analyzer (PMA), sharing a common user interface (UI). Their inter-relationship is shown in Figure 2.

Annai accepts high-level data-parallel HPF programs and low-level message-passing source code. PST acts mainly as a compiler for both paradigms. PMA and PDT are designed to assist the user by providing information at different levels of abstraction. The lowest level of abstraction, providing the most detailed information, is as close as possible to the DMPP hardware. Since one of our objectives is the development of a portable tool environment, this lowest level of abstraction is MPI.

PST [MR94], the Parallelization Support Tool, extends the current HPF definition by providing language constructs and extensive run-time support for the parallelization of irregular computations. Along with comprehensive compilation support for mixed-language program sources, applications can also be instrumented to generate debugging and performance information for the other tools. PST supplements NEC's HPF compiler.

PDT [CFR94], the Parallel Debugging Tool, is a conventional source-level debugger extended with control- and data-breakpoints with global break conditions. At the data-parallel level, PDT provides coherent graphical representations of large, distributed data-sets. At the message-passing level, PDT assists programmers with deadlock detection, race-condition detection and deterministic execution replay.

PMA [WE94], the Performance Monitor and Analyzer, exploits trace information from interactively specified source code regions where instrumentation is inserted and data collected during the execution of a parallel program. It then assists with the performance tuning and interpretation of program execution through visualization and analysis of this information. Different levels of abstraction are supported, from execution summary profiles and charts of time-varying behavior down to views of individual processes and analysis of communication events and memory utilization.

The OSF/Motif UI provides a common interface between *Annai*'s components and the user. It primarily consists of a source code and program structure browser, which can also be directed and annotated by PMA and PDT to show features or source regions of interest. Output from a running program is also displayed under the control of the UI in a separate window. UI directs the operation of the other tools and controls parallel program construction and execution.

PDT and PMA have a common interface to the parallel computing platform via the Tool Services Agent (TSA), which provides basic, low-level functions for controlling parallel program execution, currently based on the Free Software Foundation's `gdb` debugger.

Annai's portability is addressed by relying on standards, i.e., HPF and MPI as programming models, OSF/Motif for the graphical user interface, and classical debugger technology for TSA.

2.3 Interaction between Collaboration Partners

An important characteristic of our approach is the close interaction with a computer vendor. In the development process, besides tool and application developers, system designers are also involved. While tool developers design and implement *Annai* prototypes, application developers evaluate them and return

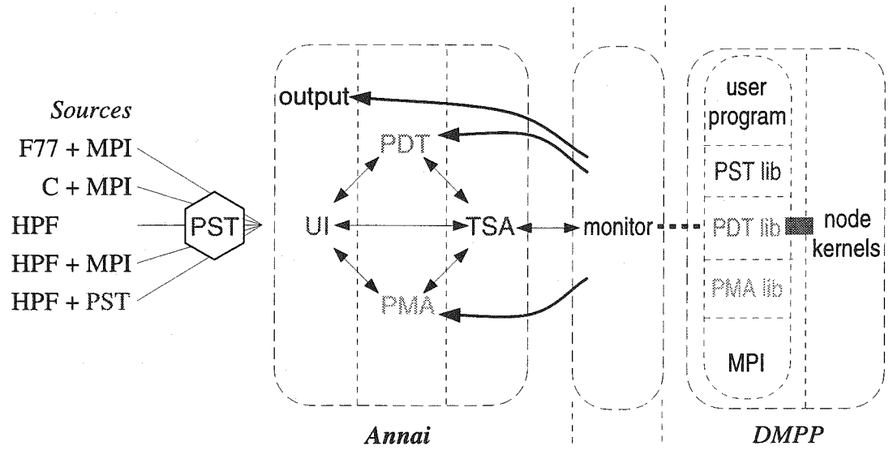


Figure 2: *Annai tool environment overview.*

requests for modifications and suggestions for possible enhancements. Application developers debug with PDT and use PMA to determine performance problems, and provide feedback to system designers if these bottlenecks are due to inefficient hardware or system software (e.g., for communication latency bound applications). On the other hand, tool developers have fundamental requirements of hardware and system software, such as a high-resolution global system clock to facilitate consisting event time-stamping.

To help attain the goals of the collaboration, NEC has provided CSCS with a Cenju-3 [DWMN94] and system software. Each of the 128 VR4400SC RISC processors in the machine has 32 Kbytes on-chip cache, 1 Mbyte of second-level cache, and 64 Mbytes of main memory. The CPUs are MIPS-compatible 64-bit processors clocked at 75 MHz. They communicate via a packet-switched multi-stage interconnection network composed of 4×4 crossbar switches. The machine is hosted by a VR4400SC-based workstation.

Although our major development platform is the Cenju-3, the tools and applications kernels presented are portable: they are also running on SUN workstations which are used for code development and debugging.

3 Parallelization Strategy for Application Kernels

The central characteristic of our approach is that we follow a tools-supported parallelization strategy. Instead of manually parallelizing one application after the other, resulting in the repetition of elementary editing steps, we develop parallelization tools and libraries which then allow a comfortable parallelization of all applications in a specific class. One of the major benefits of this approach is that parallelization is carried out at a higher level of abstraction.

Another facet of our approach is using accepted sequential public-domain libraries as a basis for the parallel kernels. Such libraries include LAPACK for dense linear algebra problems, SPARSKIT for the treatment of sparse problems, including several iterative linear solvers, ARPACK [SKSL94] for sparse eigenvalue problems, and FFTpack for one-dimensional Fast Fourier Transforms. The software developer can justifiably expect that the parallel routines for solving these problems have similar or identical library interfaces to the sequential versions, minimizing the effort needed to migrate to a

parallel architecture. Our goal is therefore to alter the outward appearance of the parallelized library routines as little as possible.

In the remainder of this section we discuss the parallelization of the iterative Krylov method BiCGSTAB from SPARSKIT, a Lanczos eigensolver for symmetric problems arising from a finite-element mesh, which uses parallelized LAPACK routines, and a unit-cube Multigrid solver from the NAS benchmark kernels. As the success of our parallelization strategy relies heavily on the tools, their key features are indicated in the individual examples.

An iterative linear solver and an eigensolver have been chosen since these are two of the most commonly used algorithms in applications, and ones for which developers most often resort to existing libraries. The third example is a NAS Multigrid benchmark kernel commonly used in evaluating parallel machines and data-parallel compilers. In addition, the Multigrid method is an important algorithmic component in preconditioners for iterative solvers. While this selection only represents a small cross-section of the work in the Joint Collaboration, it illustrates characteristic use of the tool environment for parallelizing application kernels and yields indicative performances.

3.1 BiCGSTAB solver from SPARSKIT

Problem Specification The BiCGSTAB solver [vdV92] belongs to a growing class of Krylov Subspace Methods [BBC+94] to solve large, sparse non-symmetric linear systems. SPARSKIT includes several such solvers. BiCGSTAB was chosen as a starting point due to its success in real applications [PR94].

In the linear system $Ax = b$ to be solved by BiCGSTAB, A is never explicitly referenced; the central operation in the calculation is a matrix-vector multiplication (hereafter Ax) to be provided and parallelized by the user. In this case, the input matrix is a banded random matrix of 16,384 rows and a total bandwidth of 201. Each BiCGSTAB iteration requires the computation of two matrix-vector products and some vector-vector operations. From those vector-vector operations the computation of four inner products requires global reduction operations. We subsequently discuss the parallelization of BiCGSTAB, and leave the parallelization of Ax to Section 3.2.

Parallelization Since the matrix A never needs to be constructed by the algorithm, the only data structures to be distributed are vectors needed for the right-hand side, the solution x, and several intermediate work vectors. The choices for distribution are,

1. **BLOCK** Vectors are partitioned into rectangular pieces of equal size, with every processor 'owning' one of these blocks.

2. **CYCLIC** Vector elements are assigned to processors in a round-robin fashion. In BiCGSTAB there is no reason to expect that `CYCLIC` would provide better performance than `BLOCK` distribution.

3. **BLOCK_GENERAL** This scheme is a PST extension of the HPF `BLOCK` distribution. Vectors are distributed in a block-wise fashion, but the blocks can have variable size and can contain *gaps*, i.e., unused elements. Using an 'oversized' array and leaving gaps can be useful if the problem size varies dynamically during the program execution.

4. **DYNAMIC** The most general PST data distribution. Every single element is individually mapped to a processor. This mapping can be specified in two ways: using a mapping array or using mapping functions. The first possibility is efficient but memory consuming, the latter saves memory at the cost of performance.

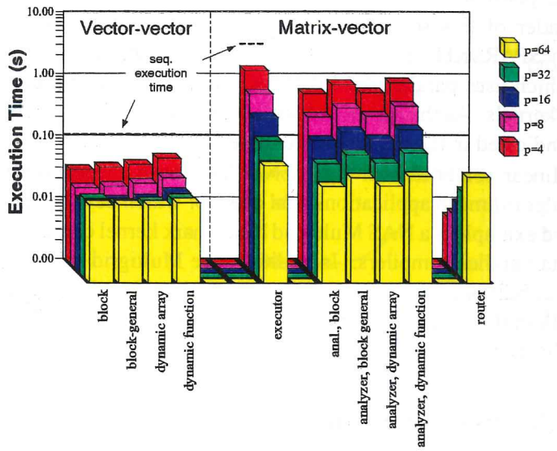


Figure 3: *Execution time on several Cenju-3 configurations of different SPARSKIT components accumulated over the first BiCGSTAB iteration for different data distributions, all emulating a block-wise distribution. During the first iteration, 8 vector-vector operations are carried out, and the executor of the matrix-vector product is called twice. Analyzer and router are only invoked once. Both router and executor performance depend only slightly on the actual distribution directive.*

In addition, vectors could be replicated on all processors and only the internal format of A distributed. This is only a viable alternative if the number of vector entries is small compared to the number of non-zero matrix values (i.e., the matrix is fairly dense) and thus will not be considered further.

Once the distribution of the vectors is determined, all loops over distributed vectors are parallelized explicitly using PST's `ALIGN` directive which assigns a loop iteration to the owner of a particular array element. The alignment is straightforward since all loops are over entire vectors. This is an important addition to the HPF compiler which performs the parallelization of loops automatically, but which fails for irregular computations or dynamic data distributions.

A parallel version of the LAPACK DDOT is constructed which globally sums the contributions of the local scalar products. This task is again easily performed with the help of loop alignment directives. Unless the matrix is explicitly redistributed, the data distribution does not change between iterations (i.e., the underlying code segments are *start-time schedulable* [SMC90]). Communication patterns can therefore be reused with PST's `SAVECOM` directive. This important feature is described more thoroughly in [MR94].

Results Figure 3 summarizes our BiCGSTAB measurements on several Cenju-3 configurations using HPF `BLOCK` and PST `BLOCK_GENERAL` and `DYNAMIC` distributions (using both mapping arrays and mapping functions). It shows both the performance of the matrix-vector product, which requires communication, and the vector-vector operations which do not. The vector-vector operations, however, consist of short aligned loops. Due to the implementation of such loops, performance is best with the regular `BLOCK` distribution and worst with dynamic mapping functions. The performance sensitivity decreases with increasing number of processors, because the global reduction operation in the inner product becomes the major bottleneck with large machine configurations.

The performance results of Table 1 show an approximate scaling behavior and that most of the

execution time is spent in the matrix-vector multiplication.

# PEs	1	4	8	16	32	64
Ax analyzer (s)	0.00	0.57	0.29	0.14	0.07	0.04
Ax executor (s)	26.34	15.54	7.79	3.83	1.82	1.00
Other operations (s)	1.10	0.43	0.29	0.29	0.19	0.17
MFlop/s	5.9	10.0	19.9	40.5	85.2	155.1

Table 1: *Performance for the linear system solution $Ax = b$ with BiCGSTAB, where A is a band matrix with bandwidth 201 and 16,384 rows. All vectors are distributed* `BLOCK_GENERAL`. *Although Ax is executed* 20 *times, the PST analysis only needs to be done the first time. Superlinear scaling occurs due to caching effects.*

Development Effort The BiCGSTAB solver, and indeed all solvers in SPARSKIT, are based on the use of n-vectors, i.e., vectors of problem size length n. The vectors need to be distributed and the corresponding loops aligned. In addition, the index mapping for the `BLOCK_GENERAL`, `DYNAMIC_ARRAY`, or `DYNAMIC_FCT` distribution have to be programmed. Although this task is sometimes complex, mapping problems are easily identified using features of PDT demonstrated later.

Since the global indexing remains in the code, its parallelization is fundamentally simple, and the BiCGSTAB routine can be rewritten quickly in HPF/PST. Few additional optimizations are necessary to avoid PST-specific inefficiencies. For instance, the expensive passing of array sections to subroutines is avoided by separating the original work array in the program into n-length vectors. This requires somewhat more time to program and some debugging with PDT, but was justified by the performance benefits.

The programming effort for the band Ax operation is minimal in this case, due to its simplicity. The programming effort for other Ax will be considered in the next section.

3.2 Eigensolver for Unstructured Problems

Problem Specification We consider the important class of applications in which the basic data structure can be described as a weakly interconnected graph with relatively few interactions between nodes and with dynamic changes in time. Such an application could use adaptive finite elements, e.g., for device simulation [Kor93], or could simulate the behavior of liquid silicon with a tight-binding model [L. 94], in which atomic interactions are constantly changing as the atoms drift.

In such sparse irregular problems, linear solvers — such as BiCGSTAB described in Section 3.1 — and eigensolvers are commonly needed. The algorithmic class of eigensolvers must again be split up into problems requiring just a few eigenvalues (e.g., the smallest, largest, or the ones closest to the imaginary axis), and a class in which most or all of the eigenvalues and eigenvectors are required. The former problem requires such Arnoldi and Lanczos-based eigensolvers as found in the ARPACK. In the symmetric case of the latter problem, the matrix can be tridiagonalized, either with Householder transformations or implicitly (using only Ax) with the Lanczos process. It can then be diagonalized using either the QR algorithm [GL89] or, more efficiently, with a divide-and-conquer approach [DS87], using routines which are available in LAPACK2.0.

As in Section 3.1 we consider here the symmetric case where only Ax is available, resulting from a summation of element matrices, e.g., $Ax = \sum_{\forall el} A_{el}x$. We thus first apply the Lanczos algorithm to tridiagonalize the matrix. It is known that round-off errors quickly lead to loss of orthogonality

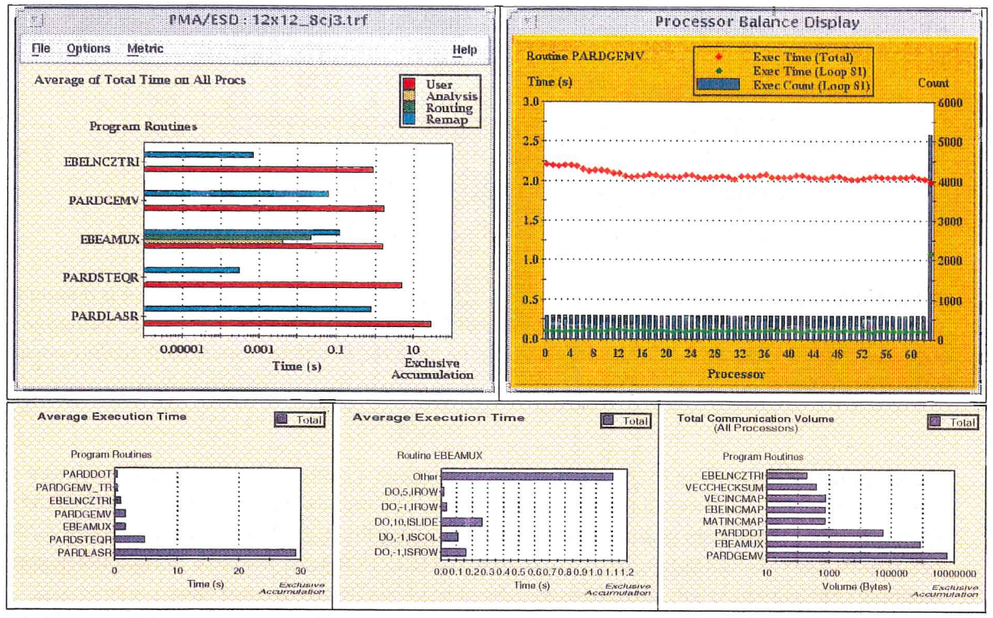


Figure 4: *PMA Execution Statistics Displays showing various profile summaries of the eigenvalue solver. Upper left is a detailed ranked profile of the five most important routines, with execution time separated into user and parallel overhead components. The* EBEAMUX *routine has the only significant overheads as it is responsible for setting up the data distributions. (Such overheads originally made this routine dominant.) The small graphs at the bottom show an initial execution time profile of the program routines, a detailed profile of the* EBEAMUX *routine (identifying the loop structure), and a communication summary (which shows that most of the communication is seen to occur in* PARDGEMV *and* EBEAMUX*, and that* PARDLASR *requires no communication). Upper right is the Processor Balance Display, which shows the variation and imbalance of selected metrics on the different processors in the* PARDGEMV *routine, identifying an inefficient unbalanced distributed loop.*

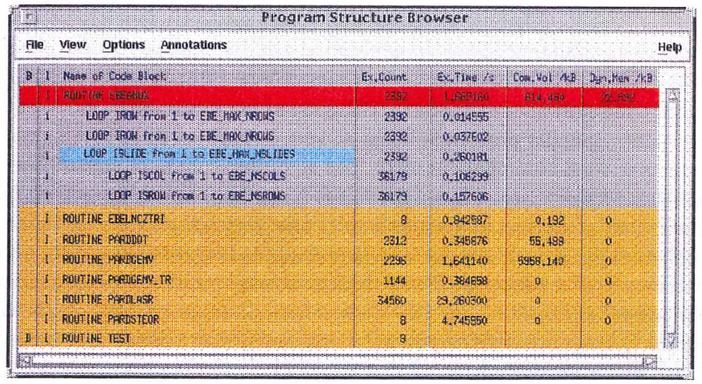
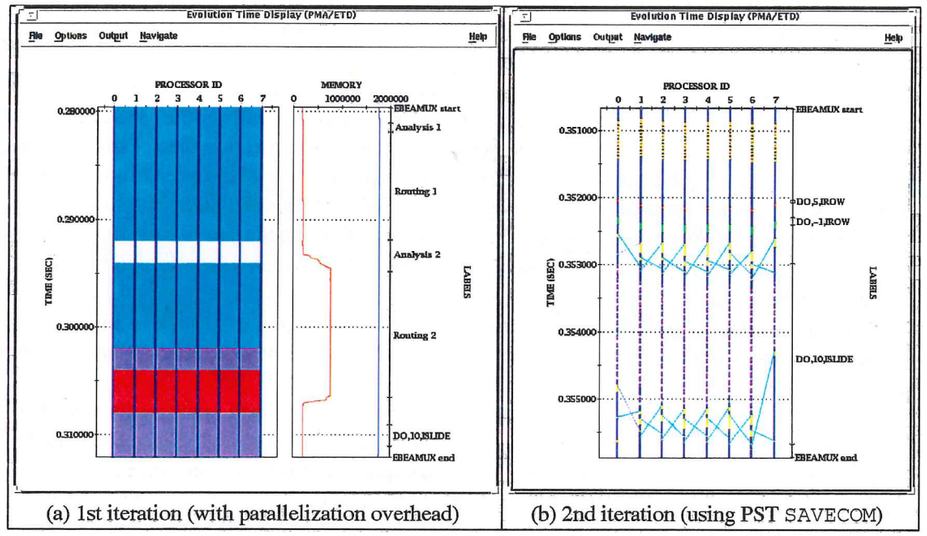


Figure 5: *PMA annotations in the* Annai *UI Program Structure Browser show execution count, total execution time, communication volume and dynamic memory usage for eigensolver program routines.*



(a) 1st iteration (with parallelization overhead) (b) 2nd iteration (using PST SAVECOM)

Figure 6: *PMA Evolution Time-line Displays providing a scrollable and rescalable chart of program execution. The vertical time axis is annotated with the major 'landmarks' — at more detailed scales additional landmarks are also shown. Within this framework various graphical displays of time-varying behavior can be shown. The figure shows individual processor states as separate threads, with either (a) a summary of the amount of communication in progress at any instant, or (b) a complete display of communication events. Standard metric graphs, such as that of memory utilization are also available, here showing PST temporary data allocation during the communication analysis. (NB: The graphs have been scaled differently to completely show the target routine,* EBEAMUX*, in each case.)*

of the Lanczos vectors [Seh89] used in this iterative process. We therefore employ the partial re-orthogonalization algorithm of Simon [Sim84] to re-stabilize the method at a lower cost than the classical approach of re-orthogonalizing each new vector against all the previous ones (so-called full re-orthogonalization).

Parallelization We use a flexible distributed format for A supported in our Parallel Library for Unstructured Mesh Problems (PLUMP) [BLM+94], which is based on the PST `BLOCK_GENERAL` distribution. The gaps described in Section 3.1, are left in the data structure to allow dynamic addition (or removal) of nodes to (from) the underlying graph, such as happens during mesh refinement. The vector x and all other vectors used in the Lanczos tridiagonalization are distributed in a `BLOCK_GENERAL` fashion.

For the second phase of the eigensolver, we use a parallelized routine which is almost identical to the LAPACK routine DSTEQR, except for the fact that the eigenvectors are also distributed in a `BLOCK_GENERAL` fashion. Only two of the LAPACK routines called from DSTEQR which refer to the eigenvectors have to be modified (as well as DSTEQR itself) — all other routines remain the same.

Assuming the diagonal and sub-diagonal vector are duplicated on all processors, there is no communication in the diagonalization, and thus we can expect excellent speed-up figures for this phase of the eigensolver.

Performance Monitoring and Analysis with PMA To understand the execution behavior and locate performance bottlenecks in the eigensolver, PMA instrumentation incorporated by the PST compilation system is activated. Dynamic configuration of this latent instrumentation allows execution information to be collected where required at the desired detail.

Figure 4 shows a selection of graphical execution profiles of the eigensolver provided by PMA's Execution Statistics Display. From investigation of the initial overall profile, selecting the different performance metrics and view options interactively explores the execution data. Principal components are isolated by masking those with insignificant contributions. Most of the execution time is seen to be currently spent in the computation-intensive PARDLASR routine, with parallelization overheads only found in routine EBEAMUX. Unbalanced loop iteration assignment to processors leading to inefficient utilization is also identified in routine PARDGEMV, though in this case it is not found to be significant for larger problem sizes.

PMA also interacts with *Annai*'s UI Program Structure Browser, as shown in Figure 5, where the program structure is presented in a tabular display, and performance annotations are incorporated as additional columns.

Originally, the EBEAMUX routine was found to be dominant, and analysis showed that this was due to large parallelization overheads every time the routine was executed. Since the data distribution is static, PST's determination of the communication required during every instance could be saved after the first iteration and reused on subsequent iterations. The current profiles were generated after the appropriate directives were specified.

Figure 6 from PMA's Evolution Time-line Display shows how the overheads still present during the first iteration are eliminated completely during the second (and subsequent iterations), giving a 6-fold performance improvement in the execution time of this routine.

The targeted instrumentation facilities provided by PMA help to ensure that the desired performance information can be obtained at the required level of detail with little effort and minimal intrusion. Source-reference to the original program structure and source code is retained throughout, and repeatedly exploited by PMA's profile summaries and chart displays, providing the deep insight into program execution necessary for effective tuning within a familiar framework.

Results The performance of the two phases of the eigensolver are listed in Table 2.

# PEs	1	2	4	8	16	32	64
Tridiagonalization (s)	1504.2	729.6	372.0	201.0	101.4	64.4	52.1
Diagonalization (s)	8656.9	2482.6	1227.7	620.0	301.4	145.4	68.5
Total (s)	10161.1	3212.2	1599.8	821.1	402.8	209.9	102.6
MFlop/s	4.2	13.2	27.2	51.7	105.3	202.9	351.7

Table 2: *All eigenvalues and eigenvectors of a test problem of a square* 40×40 *mesh (1600 nodes, 1521 elements) of rectangular finite elements with symmetric element matrices. Note that super-linear speed-up is observed in some cases due to cache effects.*

Development Effort As in the BiCGSTAB solver, the Lanczos tridiagonalization only makes use of n-vectors. Therefore the same ease of parallelizing of the sequential version applies in this case. [1]

[1] A sequential implementation of the Lanczos method with partial re-orthogonalization was not available to us and therefore had to be written.

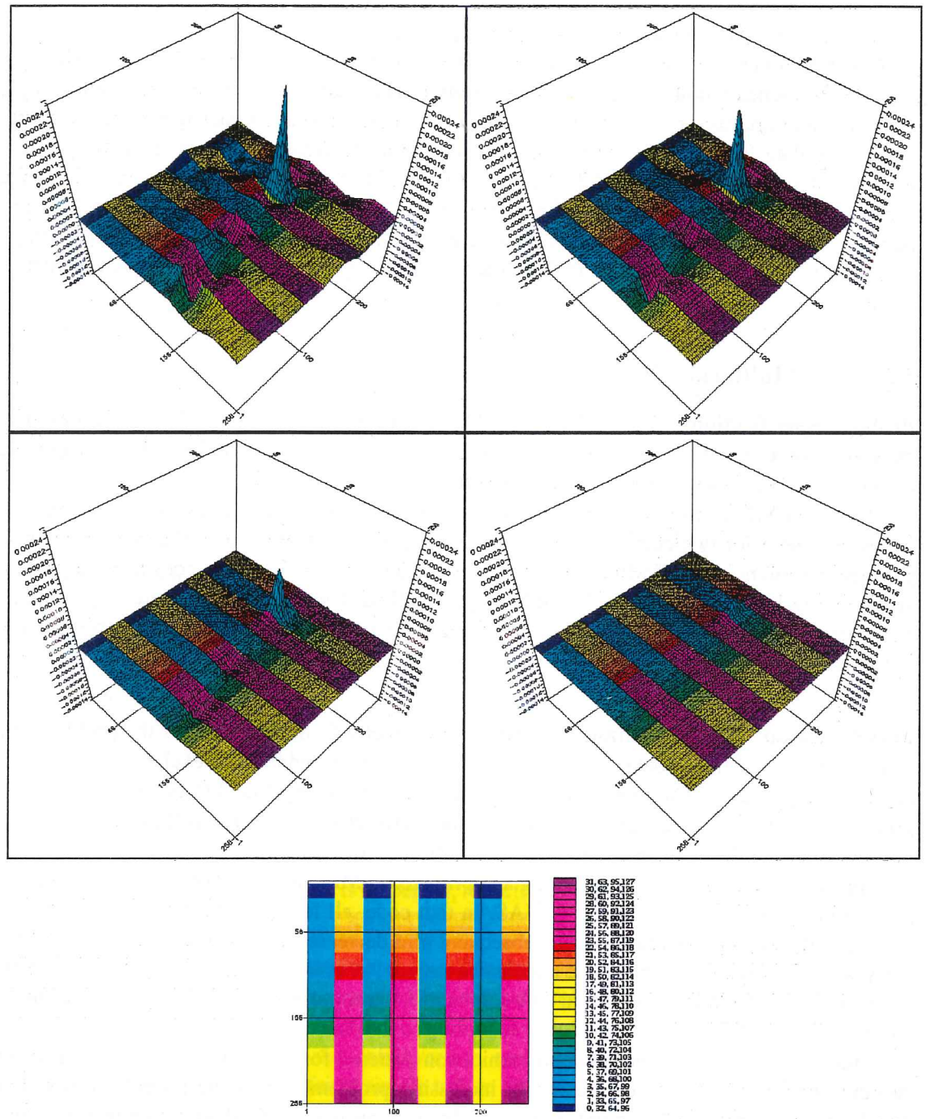


Figure 7: *Four PDT distributed data views show the value of the residual of the CLASS_B problem* ($256 \times 256 \times 256$) *for a given* x*-coordinate after 1, 2, 4, and 8 V-cycles of the MG iteration. The high frequency error components (i.e., the discontinuities) are efficiently removed in a few V-cycles. The lower views show the* (BLOCK, BLOCK) *distribution map of the cross-section and the data value browser.*

The LAPACK routines DLASR and DSWAP, called by DSTEQR, which operate on the matrix eigenvectors could be rewritten with ease since the matrix is distributed in one dimension as every vector is, namely `(BLOCK_GENERAL,*)`, and all operations are performed on entire eigenvectors.

Overall, the parallelization and debugging of the sequential Lanczos tridiagonalization and six LAPACK routines was much less than the effort of writing this non-trivial sequential code.

The parallelization of the constituent Ax routines is not counted in the above. While simple sparse Ax operations are easy to support (see Section 3.1), we are unaware of packages which support flexible distributed data formats required by unstructured mesh problems. Thus we are putting considerable investment into creating the PLUMP library to parallelize such operations as insertion of element matrices into a distributed global stiffness matrix, redistribution of that matrix when load imbalance occurs, and reordering to minimize bandwidth and maximize data locality. These algorithms are non-trivial, even with the support from PST, and accordingly require much more time to implement than the solvers.

3.3 NAS Multigrid (MG)

Problem Specification The NAS Multigrid kernel [BBDS92] (hereafter MG) calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle Multigrid algorithm on an $n \times n \times n$ grid with periodic boundary conditions.

Although MG in itself does not fit into any of the algorithmic classes defined in Section 2, we consider it here for two compelling reasons. First, MG is known to be a difficult problem for data-parallel compilers, most of which cannot accommodate the V-cycle which goes from large data sets to small and back [CZM94]. Secondly, the Multigrid method is widely used in several algorithmic classes, e.g., linear solvers and preconditioners for iterative methods, thus making it of interest for our future library developments.

Parallelization The parallelization of MG arises directly from the nature of the problem: the cubic grid is partitioned into blocks. Communication is then only required to exchange elements between adjacent block faces. The most efficient blocks were empirically found to be "matchsticks": all grid points in one dimension are on one processor and distributed block-wise in the other two dimensions, in other words an HPF `(*,BLOCK,BLOCK)` distribution.

An extensive effort went into writing an optimized C/MPI version. The PST version was based on the original Fortran code supplied by NAS. In that code, all levels of the grid were kept in one long array and the amount of data for each subsequent grid decreased exponentially. Each grid level in this array was distributed regularly in the matchstick fashion. A user-defined mapping was used to map each global index to the index of the local data segment in the appropriate grid. Loop distribution directives were applied to all of the $n \times n \times n$ loops according to the data distribution.

The benchmark uses multiple communication patterns for its critical code segments: the main subroutines are called with decreasing or increasing problem size as the program proceeds through one V-cycle. Since a large number of V-cycles is typically required to compute the solution of a given problem (the "Class B" problem is defined to use 20 V-cycles) the overhead to generate such communication patterns in the first V-cycle becomes small compared to the overall execution time.

Extra optimizations of the PST code are required to improve performance. For instance, inner loops are programmed as separate subroutines and compiled with a standard Fortran compiler. This is worth the effort because PST does not yet recognize loop-invariant calculations and cannot yet optimize address calculations. In addition, loops need to be rearranged such that non-local elements are only accessed once. Otherwise the same value would be transferred multiply (in the same message) which

does not cause incorrect results but degrades performance. In due course of the project, the PST run-time system will be able to detect multiple remote data accesses.

Distributed Data Visualization and Debugging with PDT MG can only be debugged effectively if its huge distributed data structures may be examined and manipulated as a whole, without either particular knowledge or interest in the current data distribution. In this respect, PDT provides a convenient mechanism for visualizing entire arrays and array segments allocated across processors.

As an example, Figure 7 shows different PDT representations of a slice (with a given x coordinate) of the MG residual at four different iterations. Both data values and data distribution of the 2-dimensional array representing the slice are depicted. The picture at the bottom shows the HPF `(BLOCK,BLOCK)` distribution of the slice on 128 PEs, colors representing the PEs on which the blocks are located. The 3-dimensional graphs at the top show data values of the slice after 1, 2, 4, and 8 V-cycles of the MG iteration, and depict how the algorithm converges. PDT supports interactive rotation and zooming of the graphs to suit the user's preferred view. A value browser (not shown in the figure) could also be used to look at the numerical values stored in the array.

Additionally, PDT supports control- and data-breakpoints (also known as watchpoints) with global break conditions for stopping the program at interesting points in the computation. If there were an apparent anomaly in one of the MG global views of Figure 7, it could be investigated by setting a conditional watchpoint on the array. For example, program execution would be stopped when any element of the array becomes larger than a certain value.

Watchpoints are notoriously slow because most debuggers implement them by single-stepping the program and evaluating the condition at each step. PDT uses a more efficient mechanism based on watching all memory updates [WLG93] — a technique requiring instrumentation of the target program. Figure 8 shows costs of memory watching, i.e., instrumentation overhead introduced by PDT when the MG C/MPI version is compiled without optimization. The intrusiveness of the watching mechanism depends on the *number* of memory regions (of arbitrary sizes) watched. In the common case where only a few (less than 10) memory regions have to be watched, performance is degraded by upto a factor of three. This is generally still acceptable in a debugging session.

Results The performance of the C/MPI and HPF/PST versions are compared in Table 3. The C/MPI version is faster because all communication is concentrated into one MPI message exchange, whereas in the HPF/PST version communication is dispersed, reflecting the structure of the sequential code.

Development Effort The parallelization and optimization of the MPI version of MG took a student eleven weeks, in which the code was redesigned to overlap calculation and computation, and rewritten in C with explicit message passing.

Given the existing NAS MG F77 sequential version, and the fact that an HPF/PST implementation retains the global name space, it is a fairly simple task to implement a straightforward parallel version using the simple `(*,BLOCK,BLOCK)` data distribution. In addition, each distributed grid in the V-cycle, from fine to coarse and back, requires a separate work space, requiring the definition of a `DYNAMIC` mapping to access the appropriate grid at any given level. The definition of this mapping is already the most difficult aspect of the 'naive' parallelization.

The additional optimizations to the HPF/PST code were few in number and quickly implemented. The main obstacle is the need to understand how to optimize the code—a task which requires some knowledge of the behavior of the PST compiler. This effort took considerably more time than the 'naive' parallelization, but still an order of magnitude less time than the C/MPI implementation.

		HPF/PST			C/MPI
# PEs	Problem	$T(i=1)$	$T(i=2)$	MFlop/s	MFlop/s
64	64	1.72	0.20	159	405
	128	3.78	1.00	249	718
	256	14.95	6.45	311	856
128	64	3.06	0.15	208	586
	128	4.92	0.66	377	1197
	256	12.81	4.11	488	1568

Table 3: *Performance of MG when parallelized with PST on 64 and 128 processor Cenju-3 configurations. Execution times of the first and second V-cycle are shown and MFlops for the second cycle (when the iteration enters steady state, and PST-generated communication patterns are reused) are compared to what is achieved with a manually parallelized C code.*

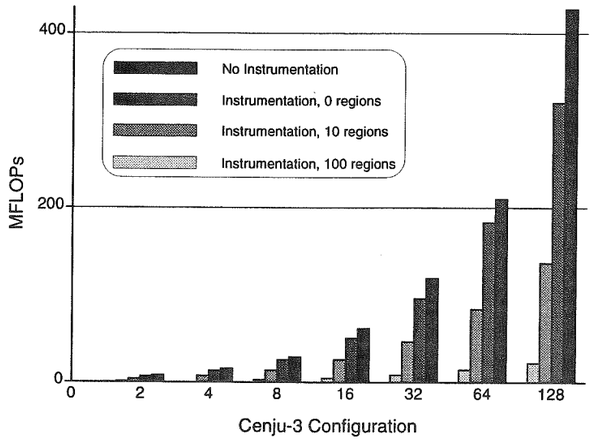


Figure 8: *Performance in MFLOPs of the NAS MG kernel ($n = 128$) parallelized manually with C/MPI, compiled without optimization and running in parallel on Cenju-3 configurations of different sizes. Several levels of instrumentation are considered (with simultaneous watching of 0, 10 and 100 memory regions, respectively) and compared to non-instrumented code.*

4 Conclusions

We have considered a variety of scientific fields and have isolated several algorithmic classes which recur in numerous applications. From these classes we have presented three example codes—the BiCGSTAB iterative linear solver from SPARSKIT, a Lanczos eigensolver in part from LAPACK, and the NAS Multigrid kernel—whose parallelization has been realized with the help of the *Annai* tool environment. We have analyzed aspects of the parallelization and have discussed the effort required in each case. The results are favorable in terms of development effort and overall performance.

Several important features of *Annai* have been identified, such as the need for a `BLOCK_GENERAL` distribution in PST to supplement the existing HPF `BLOCK` distribution. The tracing and analysis facilities of PMA were key to determining and alleviating performance bottlenecks in all of the implementations discussed. Finally, parallelization would not have been possible in such short time without the parallel debugging facilities of PDT.

From the results presented, we have shown a viable approach to the parallelization of applications for distributed memory machines, namely the use of an advanced tool environment and the parallelization of kernels into libraries commonly used in a wide range of applications. We strongly believe that this approach to parallelization allows quicker and easier exploitation of high performance architectures, realizing applications which produce the leading-edge results which developers seek.

Acknowledgements We would like to thank all the students of the 1993 and 1994 CSCS Summer Student Internship Programs, J. Blandy, M. T. Nyeu, U. Kühn, I. Beg, U. Krishnaswamy, E. La Cognata, M. Meehan, P. Przybyszewski, T. Schrøder, T. Toupin, and Wu Ling, who helped parallelize codes and develop parts of the tool environment. In addition, we are indebted to collaborators J. Nievergelt, M. Müller, C. Wirth, J. Korvink of the ETH Zürich, and L. Colombo of the Università di Milano, for their useful input and constructive comments which influenced the design of the algorithmic libraries.

References

[ABD+92] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, USA, 1992.

[ABJ+91] D. Allen, R. Bowker, K. Jourdenais, J. Simons, S. Sistare, and R. Title. The Prism programming environment. In *Proc. Supercomputer Debugging Workshop '91*, pages 1–7, Albuquerque, NM, USA, November 1991.

[BBC+94] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Publications, 1994.

[BBDS92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmarks Results. Technical Report RNR-92-002, NASA Ames Research Center, CA, Dec 1992.

[BBN94] BBN. *TotalView User's Guide*. BBN Systems and Technologies, Apr. 1994.

[BLM+94] I. Beg, W. Ling, A. Müller, P. Przybyszewski, R. Rühl, and W. Sawyer. PLUMP: Parallel Library for Unstructured Mesh Problems. In *Proceedings of the IFIP WG 10.3 International Workshop and Summer School on Parallel Algorithms for Irregularly Structured Problems (Geneva, Switzerland)*. Kluwer Academic Publishers, Aug. 1994.

[CDE+94] C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, B. J. N. Wylie, and F. Zimmermann. Application-Driven Development of an Integrated Tool Environment for Distributed Memory Parallel Processors. In V. K. Prasanna, V. P. Bhatkar, L. M. Patnaik, and S. K. Tripathi, editors, *Proceedings of the First International Workshop on Parallel Processing (Bangalore, India, December 27–30)*, pages 110–116. Tata McGraw-Hill, New Delhi, India, 1994. ISBN 0-07-462332-X.

[CDPW94] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. A User's Guide to BLACS v1.0. World-Wide Web documentation `<http://www.netlib.org/scalapack/index.html>`, 1994.

[CEF+95] C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In H. El-Rewini and B. D. Shriver, editors, *Proceedings of the 28th Hawaii International Conference on System Sciences, Volume II (Maui, Hawaii, USA, 3–6 January, 1995)*, pages 242–251. IEEE Computer Society Press, Jan. 1995. ISBN 0-8186-6935-7.

[CFR94] C. Clémençon, J. Fritscher, and R. Rühl. Execution Control, Visualization and Replay of Massively Parallel Programs with Annai's Debugging Tool. Technical Report CSCS-TR-94-09, CSCS, CH-6928 Manno, Switzerland, 1994.

[CZM94] B. Chapman, H. Zima, and P. Mehrotra. Extending HPF for Advanced Data-Parallel Applications. *IEEE Parallel & Distributed Technology*, 2(3):59–70, Fall 1994.

[DS87] J. J. Dongarra and D. C. Sorensen. A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem. *Scientific and Statistical Computing*, 8(2):139–154, Mar. 1987.

[DW95] J. J. Dongarra and R. C. Whaley. A User's Guide to BLACS v1.0. DRAFT LAPACK Working Note, Oak Ridge National Laboratory, TN, USA, 1995.

[DWMN94] S. Doi, T. Washio, K. Muramatsu, and T. Nakata. Implementing a CFD Solver on Cenju-3 Parallel Computer. In *Preprints of Parallel CFD '94 (Kyoto Institute of Technology, Japan, May 1994)*, pages 31–36, May 1994.

[FHM+94] P. Flükiger, E. Heeb, N. Masuda, W. B. Sawyer, C. Stern, and F. Zimmermann. Parallelization of Scientific Applications on Cenju-3. *CrosSCutS (CSCS Newsletter)*, 3(3):1,3–5, Dec. 1994.

[GL89] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins, 2nd edition, 1989.

[HPF93] HPFF (High Performance Fortran Forum). High Performance Fortran Language Specification: Version 1.0. *Scientific Programming*, 2(1&2), 1993.

[Int93] Intel Corporation. *Paragon Software Tools User's Guide*. Intel SSD, 1993.

[Kor93] J. Korvink. *An Implementation of the Adaptive Finite Element Method for Semiconductor Sensor Simulation*. PhD thesis, ETH-Zürich, Nov. 1993. Verlag der Fachvereine Zürich, Bericht Nr. 8.

[L. 94] L. Colombo. Tight-Binding Molecular Dynamics: Present Status and Perspectives. In *Proceedings of the 6th International Conference on Physics Computing (PC'94)*, pages 231–238. European Physical Society, Geneva, Switzerland, Aug. 1994.

[MPI94] MPIF (Message Passing Interface Forum). MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3&4):157–416, 1994.

[MR94] A. Müller and R. Rühl. Extending High Performance Fortran for the Support of Unstructured Computations. Technical Report CSCS-TR-94-08, CSCS, CH-6928 Manno, Switzerland, 1994. accepted for publication in ACM International Conference on Supercomputing, July 1995, Barcelona, Spain.

[PR94] C. Pommerell and R. Rühl. Migration of Vectorized Iterative Solvers to Distributed Memory Architectures. In *Colorado Conference on Iterative Methods (Breckenridge, CO, USA, April 1994)*, 1994. Preliminary proceedings, accepted for publication in SIAM J. Sci. Comput.

[Rüh92] R. Rühl. *A Parallelizing Compiler for Distributed-Memory Parallel Processors*. PhD thesis, ETH-Zürich, 1992. Published by Hartung-Gorre Verlag, Konstanz, Germany.

[Saa90] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computation. Technical Report CSRD Report no. 1029, University of Illinois, IL, USA, Aug. 1990.

[Seh89] N. S. Sehmi. *Large Order Structural Eigenanalysis Techniques*. Ellis Horwood Ltd., 1st edition, 1989.

[Sim84] H. D. Simon. The Lanczos Algorithm With Partial Reorthogonalization. *Mathematics of Computation*, 42(165):115–142, 1984.

[SKSL94] W. Sawyer, G. Kreiss, D. Sorensen, and J. Lambers. Arnoldi Method Applied to Burgers' Equation. Technical Report CSCS-TR-94-04, CSCS, CH-6928 Manno, Switzerland, May 1994.

[SMC90] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *Journal of Parallel and Distributed Computing*, April 1990.

[vdV92] H. A. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, Mar. 1992.

[WE94] B. J. N. Wylie and A. Endo. Design and Realization of the Annai Integrated Parallel Programming Environment Performance Monitor and Analyzer. Technical Report CSCS-TR-94-07, CSCS, CH-6928 Manno, Switzerland, Aug. 1994.

[WLG93] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. *SIGPLAN Notices*, 28(6):1–12, June 1993.