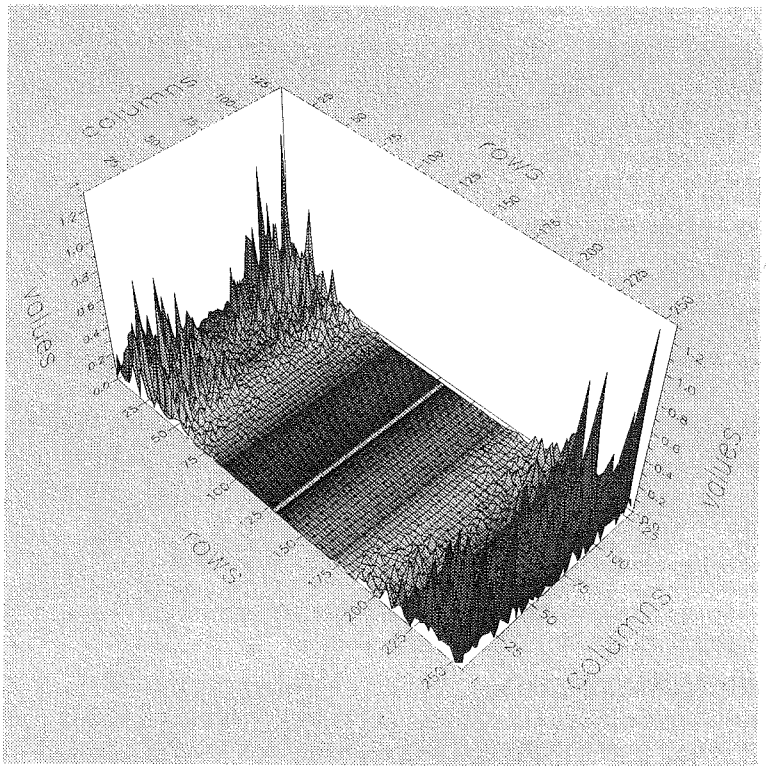
Centro Svizzero di
Calcolo Scientifico

Swiss Center for
Scientific Computing

ETH

Tools-supported HPF and MPI Parallelization of the NAS Parallel Benchmarks

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



TECHNICAL REPORT

TR-96-02

March 1996

OTHER PUBLICATIONS BY CSCS/SCSC

Annual Report: yearly review of activities and projects
CrosSCutS (triannually): newsletter featuring announcements relevant to our users as well as research highlights in the field of high-performance computing
Speedup Journal (biannually): proceedings of the SPEEDUP Workshops on Vector and Parallel Computing, published on behalf of the SPEEDUP Society
User's Guide: manual to hardware and software at CSCS/SCSC

To receive one or more of these publications, please send your full name and complete address to:

Library
CSCS/SCSC
via Cantonale
CH-6928 Manno
Switzerland

Fax: +41 (91) 610 8282

E-mail: `library@cscs.ch`

Technical Reports are also available from:
`http://www.cscs.ch/Official/Publications.html`
A list of former IPS Research Reports is available from:
`http://www.cscs.ch/Official/IPSreports.html`

Tools-supported HPF and MPI Parallelization of the NAS Parallel Benchmarks

C. Clémençon[1] K. M. Decker[1] V. R. Deshpande[1]
A. Endo[2] J. Fritscher[1] P. A. R. Lorenzo[1]
N. Masuda[2] A. Müller[1] R. Rühl[1]
W. Sawyer[1] B. J. N. Wylie[1] F. Zimmermann[2]

TR-96-02, March 1996

Abstract. High Performance Fortran (HPF) compilers and communication libraries with the standardized Message Passing Interface (MPI) are becoming widely available, easing the development of portable parallel applications. The recently developed *Annai* tool environment supports programming, debugging and tuning of both HPF- and MPI-based applications. Considering code development time to be as important as final performance, we address how sequential versions of the familiar NAS Parallel Benchmark kernels can be expediently parallelized with appropriate tool support. While automatic parallelization of scientific applications written in traditional sequential languages remains largely impractical, Annai provides users with high-level language extensions and integrated program engineering support tools. Respectable performance and good scalability in most cases are obtained with this straightforward parallelization strategy on the NEC Cenju-3 distributed-memory parallel processor, even without recourse to platform-specific optimizations or major program transformations.

Keywords. Application Parallelization Strategies, Parallel Program Engineering Tools, HPF and MPI Comparison, NAS Parallel Benchmarks

[1] Swiss Center for Scientific Computing (CSCS/SCSC)
[2] NEC European Supercomputer Systems, Swiss Branch
Via Cantonale, CH-6928 Manno, Switzerland
`decker@cscs.ch`

Contents

List of Figures

List of Tables

1 Introduction

Expedient exploitation of distributed computer systems, in a manner which ensures application developer's investments, requires appropriate consideration of the parallelization effort and desired portability. Standard specifications have recently emerged which address these considerations for the two most common parallel programming paradigms: 'data-parallel' programming with High Performance Fortran (HPF) and explicit use of Message Passing Interface (MPI) communication primitives. While MPI implementations and HPF compilers are currently available, tools supporting portable parallel programming have not yet adequately addressed the development issues of redesigning algorithms, optimizing data distributions, using optimized libraries, and re-engineering critical sections. An effective parallel debugger must provide convenient facilities for interacting with programs distributed on multiple processors, such as insightful global views of distributed data structures. Performance analysis tools similarly need to transparently manage program execution information, as users search for critical regions of inefficient computation, communication and load distribution, which can then be targetted for further detailed investigation and tuning.

The *Annai* environment and component tools for HPF and MPI support the parallelization of sequential applications as a step-wise refinement ranging from straightforward data-parallel to hand-tuned asynchronous message-passing programs. Using our HPF compilation system, we can exploit the virtues of the global name space provided by HPF and the optimized communication routines and widespread availability of MPI for a wide range of applications.

The benchmarks developed for the Numerical Aerodynamic Simulation (NAS) Program are especially designed to evaluate and compare the performance of parallel computers [1]. While much time and effort has been invested by computer vendors to attain the highest possible computer performance on specific systems, little consideration has been paid to the parallelization effort required by typical programmers — a significant issue in the development of real high-performance applications. In contrast to other work on the NAS Parallel Benchmarks (NPB), we regard them as representative of applications for which portable parallel versions with scalable performance are desired. As such, the existing Fortran sequential NPB implementations are suitable candidates to be parallelized in HPF and MPI using an expedient tools-supported approach, where development time is considered as important as optimization.

Section 2 describes the *Annai* tool environment support for the parallelization of applications like the NAS kernels. The parallelization of the individual NAS kernels with the help of the tools is then described in section 3, and performance results are presented. Finally, our results are summarized in section 4.

2 Tool support for data-parallel and message-passing program engineering

High Performance Fortran [2] and the Message Passing Interface [3] are proposed standards for two paradigms for distributed memory parallel computing: data-parallel and message-passing. Message-passing programming is generally tedious and error-prone because data must be distributed explicitly and all communication primitives have to be manually inserted. Local data is referenced through local indices, which may have a complex relationship to the global array indices in the sequential program. Data-parallel languages retain the global name space of the sequential program and compiler directives specify data distribution, dependencies, etc. The compiler parallelizes loops which operate on distributed data and generates communication primitives as required.

There are, however, known deficiencies in the available HPF compilers, which attempt to automatically generate communication patterns for given loop and data distributions. HPF-implemented applications have typically been an order of magnitude slower than their message-passing equivalents, and the current HPF specification makes it very difficult to express all problems in a straightforward data-parallel way. The slow progress and disappointing performance have lead the HPF Forum to discuss extensions to the current HPF specification [4, 5, 6], to support a larger range of applications, such as those based on irregular meshes.

As part of the *Annai* tool environment [7, 8] developed within the *Joint CSCS-NEC Collaboration in Parallel Processing* an HPF compilation system has been implemented which uses MPI for underlying communication and offers the possibility to combine data-parallel with message-passing programming. An existing HPF compiler from NEC is augmented by the *Parallelization Support Tool* (PST) [9], which realizes extensions [4] to the existing HPF specification in order to ease program development and code generation, and to improve HPF performance. The trade-off between performance and ease of programming can be avoided by utilizing the virtues of both — the HPF global name space together with optimized MPI communication routines.

Like HPF, PST provides the user with a global name space. In contrast to HPF, however, PST does not provide a single thread of execution, but is based on the SPMD (Single Program Multiple Data) paradigm: by default, all statements are executed by all processors. Where users want only selected processors to execute parts of the code, they must specify that explicitly by using loop distribution directives, or by executing code depending on the processor identifier. One advantage of the SPMD model over the single-thread model is that routines of the underlying message-passing library can be called. PST uses MPI *communicators* to avoid that user-inserted message-passing primitives interfere with compiler-generated communication. Also variables are by default replicated and private; only distributed arrays are part of the global name space, and their consistency is enforced across processors.

Arrays can be distributed using all regular HPF distribution and alignment directives. In addition, PST provides user-defined data distributions. `BLOCK_GENERAL`, a generalization of the `BLOCK` distribution, allows variable block lengths, and permits *gaps*: i.e., parts of the array may remain unmapped and are not accessible. Arrays can also be distributed

via a mapping array, or with integer valued functions which map global indices to local indices and owning processor.

The HPF/PST compilation system is only one component of *Annai*. The integrated environment also provides debugging, performance monitoring and analysis tools with complete source reference, for comprehensive application development support.

The *Parallel Debugging Tool* (PDT) [10] is a conventional source-level symbolic debugger, enhanced to support different levels of abstraction. At the data-parallel level, PDT provides a *Distributed Data Visualizer* to graphically represent large, distributed data-sets (both views of the data values and the data distribution itself), and control- and data-breakpoints with global break conditions. At the message-passing level, PDT assists programmers with deadlock and race detection, and deterministic execution replay.

The *Performance Monitor and Analyzer* (PMA) [11] manages program instrumentation and subsequent execution information analysis. Instrumentation inserted by the compilation system, or incorporated within the communication library, can be interactively configured to generate summary profiles or detailed execution traces. Users may also specify additional information about their application or select regions for different instrumentation.

Statistics can be accumulated and summarized during program execution, for interactive browsing with the *Execution Statistics Display*. Critical regions of computation and communication can be identified from such profiles, along with memory utilization and the balance among processors. The time-varying behavior of selected parts of a program is captured in an execution trace from every processor. When visualized and browsed with the *Evolution Time-line Display*, inter-processor dependencies and communication intensive regions can be examined in detail.

The primary hardware platform for the project is the NEC Cenju-3, equipped with 128 VR4400 RISC processors providing a peak performance of 6.4 GFlops (50 MFlops per node) where each processing element (PE) has its own 64 MByte local memory. In addition to 32 kByte on-chip primary cache, all processors have a secondary cache of 1 MByte. The processing elements of the NEC Cenju-3 are connected by a high-speed multi-stage interconnection network which is built of 4×4 crossbar switches. The system is connected to a front-end NEC EWS4800 workstation which also handles the I/O.

3 The NAS parallel benchmarks

The aim of the benchmarks which have been issued by the NAS Program is to measure the actual performance of highly parallel computer systems and to compare them with conventional supercomputers. The original benchmark suite (NPB 1.0) is defined [1, 12] in a "pencil and paper" fashion, i.e., the choice of data structures, algorithms, processor allocation and memory usage are left up to the implementor of the benchmark (as far as the specification allows). Of course, for a given set of initial data there must exist a unique solution. Although originally related to computational fluid dynamics, the benchmarks are also representative of other parallel applications. The biannually published NPB report results [13] are well accepted as a comprehensive overview about the comparative performance of parallel machines.

The NAS Parallel Benchmarks 1.0 consist of eight codes, of which the five kernels are the subject of this study: Embarrassingly Parallel (EP), Multigrid (MG), Conjugate Gradient (CG), three-dimensional Fast Fourier Transform (FT), and Integer Sort (IS). The other three, the LU-solver (LU), Penta-diagonal Solver (SP), and Block Tridiagonal Solver (BT) are simulated computational fluid dynamics (CFD) applications and considerably more complex. It would have required a bigger programming effort to consider them here. Various problem classes — a "Sample Class", "Class A" and "Class B" (in NPB 2.0 also "Class C") — are defined corresponding to the different sizes of the principal arrays and the estimated number of operations.

EP: In the Embarrassingly Parallel kernel pairs of uniform pseudo-random numbers are generated and transformed into two-dimensional Gaussian deviates. The number of Gaussian pairs in successive square annuli are tabulated. The kernel belongs to the class of parallel applications with concurrent tasks independently executed on the individual processors and with little or no interprocessor communication involved. This way, NAS-EP provides an estimate of the highest achievable floating point performance of a given platform without significant communication. The parallelization effort for both, the data parallel and the message-passing approach, is minimal.

MG: The Multigrid kernel calculates an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ using four iterations of the V-cycle multigrid algorithm on a $n \times n \times n$ grid with periodic boundary conditions. The four most important routines are: the residual projection, the smoother, the trilinear interpolation of the correction and the residual calculation. The communication is highly structured and goes through a fixed sequence of regular patterns, but the original arrays do not easily fit into the standard HPF distributions.

CG: Using the inverse power method and the iterative conjugate gradient algorithm, the NAS-CG kernel finds an estimate of the smallest eigenvalue of a sparse, symmetric, positive definite matrix with a random pattern of non-zeros. The key operation is a sparse matrix-vector multiplication which employs irregular communication patterns that are typical for unstructured grid computations. Data locality and partitioning are major concerns for an efficient parallel implementation with balanced processor load and low communication overhead.

FT: The three-dimensional complex Fast Fourier Transform is applied to solve the heat equation $\partial u/\partial t = \alpha \nabla^2 u$. Fourier transforms are, e.g., essential for many spectral calculations. The data in a parallel Fourier transform on a d-dimensional array is most easily distributed block-wise according to the d-direction. Multiple one-dimensional FFTs are then computed without interprocessor communication for the first $d-1$ dimensions. An array transposition with respect to the first and last dimension must be performed, before the final set of one-dimensional FFTs can be calculated locally again. The efficiency of a parallel implementation relies on the performance of the transposition, where each processor in a global all-to-all communication phase exchanges data with every other processor.

IS: The Integer Sort kernel determines the rankings of N keys in the range $[0, B_{max}-1]$. It is an important operation, e.g., in many "particle method" codes. The keys are equally (block-wise) distributed. Besides communication performance, the NAS-IS kernel is a test for integer computation.

While a number of implementations based on various message-passing libraries have been presented (for a list of references see [12]), data-parallel versions have proven more difficult and are only starting to appear [14, 15]. In contrast to highly optimized implementations, we consider the NPB here as representative applications and focus on practical parallel implementations which should achieve reasonable performance and scalability without undue parallelization effort.

The techniques used for the parallelization of the NAS kernels exploit HPF/PST and MPI in the following ways:

- A parallel algorithm in a data-parallel style, possibly starting with an existing sequential implementation, is sought from the NPB specification.
- Data structures in the program are carefully analyzed to select a suitable HPF distribution. Whenever one is not provided by the current HPF specification [2], a PST `BLOCK_GENERAL` distribution — or even a user-defined mapping function or array — is used. Alternative distributions may be investigated to determine the most appropriate distribution.
- Within HPF/PST programs, communication may be coded explicitly in MPI, for performance or memory reasons, using HPF/PST *private* subroutines which are not analyzed for inter-processor dependencies.
- Explicitly-coded standard procedures may be substituted by optimized libraries written in HPF/PST, Fortran or C, together with MPI.
- In MPI versions of functions developed from HPF originals, collective operations have to be explicitly coded using MPI equivalents. Global communication is retained for simplicity, rather than optimizing point-to-point communication and restructuring to overlap computation and communication.

This general strategy meets the wish of program developers to parallelize codes step-by-step: starting with a simple data-parallel program, then exploring the critical regions of computation and communication with the appropriate tools and evaluating possibilities for improvements (e.g., better data partitioning). Incorporation of optimized library functions or explicitly-coded message-passing routines are further options.

Absolute maximum performance is not one of our primary goals in this study. Within our strategy further optimizations are still possible in some cases, but due to inefficiences in the algorithms, and inherent overheads within the HPF compilation system, they are not necessarily expected to match the optimum performance.

We present the performance results — according to the tradition of NPB — as the ratio of the parallel execution time to the current best execution times of the sequential F77 code on one processor of both the Cray Y-MP and C90 [13]. This evaluation allows

a comparison of the largest problem classes with an optimized sequential version [1]. Furthermore, we want to consider the performance under the aspect of scalability as well as communication overhead.

3.1 Embarrassingly Parallel (EP) kernel

Algorithm: The Embarrassingly Parallel benchmark generates a large number of pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs lying in successive square annuli. As in other NAS benchmarks, a linear congruential random number generator with modulus $m = 2^{46}$ and multiplier $a = 5^{13}$ is used to generate a sequence of uniformly distributed random numbers $0 < r_k < 1$ $(1 \leq k \leq n)$: $i_k = a\, i_{k-1} \bmod m$, and $r_k = i_k/m$. Any particular integer i_k of these uniformly distributed random numbers can be calculated directly from the initial seed i_0 with the help of the binary algorithm for exponentiation in only $O(\log(n))$ steps, a useful fact for the parallel implementation.

In a second step, pairs of random numbers $(x_k = 2r_{2k-1}-1, y_k = 2r_{2k}-1)$ are converted into a set of independent Gaussian deviates (X_j, Y_j) with mean zero and variance one. Finally, the computation is verified by the comparison of a checksum and the count of pairs (X_j, Y_j) that lie in the square annulus $l \leq max(|X_j|, |Y_j|) < l+1$ with given reference numbers for $0 \leq l \leq 9$.

Parallelization: The NAS-EP code is typical for task parallelism. Since the seeds of the random number generator can be propagated over n numbers by an efficient $\log(n)$ method, the computation of the first two steps can be performed concurrently. In the HPF code, the computation of the checksums and the counter is distributed over all processors, and the partial sums and counters are finally collected (Fig. 4).For the MPI code, the outer loop is split into parts of equal size to be computed locally on every processor. The only communication involved is the collection of the local sums and counters on processor 0 with an `MPI_Reduce` (Fig. 5).

Qualitative Discussion: As for all subsequent kernels we first consider the profile provided by the PMA Execution Statistics Display (Fig. 6). The execution time of the program components represented here does not incorporate sub-components accounting (i.e., accumulation is exclusive).

In the Fortran MPI code, about one-third of the time is spent generating the random numbers with the efficient Parallel Random Number Generator (PRNG), and the preparation of the Gaussian deviates (GAUSS) takes about two-thirds of the CPU time. In the HPF code the time spent in the Fortran PRNG library is, of course, the same. The GAUSS part, on the other hand, has a lower performance, because the HPF compiler is passing the Fortran code through a Fortran-to-C translator to a C-compiler resulting in a less efficient machine code. We measure a performance improvement of a factor 1.5 for the compilation with the Fortran compiler instead of the usage of the HPF compiler. Whereas the collection time (COLLECT) in MPI is negligible, the corresponding

[1]The largest NPB problem classes generally do not fit on one Cenju-3 PE.

HPF-segment shows a considerable 5s in the Class B benchmark on 64 processors of the Cenju-3. The contribution for the calculation of the seed (CALC_SEED) is invisible in both versions.

In a real parallel application, one might be interested in Gaussian deviates with mean $(0.5M, 0.5M)$ and variance $(0.2M, 0.2M)$ that might be assigned to a `(BLOCK,BLOCK)` distributed array. As revealed by the PDT Distributed Data Visualizer view (Fig. 1) of that array we get a 2-dimensional Gaussian distribution with varying bin size depending on M. Here, we chose $M = 16$ and $M = 64$, respectively. Qualitatively, we learn that the bin size with $M = 64$ is too small for a *smooth* Gaussian distribution for the available statistics. The debugger also supports tabulation and inspection of the actual values.

Performance Results: The communication overhead for NAS-EP is negligible, and from 1 to 128 processors we have a completely scalable application (Fig. 12). As shown in Table 1, we get close to the peak performance of the Cenju-3 (about 43% with the HPF code and 61% with MPI). We consider the NAS-EP MPI and HPF performance as the highest achievable and compare subsequent kernel performances to those results.

3.2 MultiGrid (MG) kernel

Algorithm: The V-cycle multigrid algorithm is applied to a $n \times n \times n$ grid to approximate the solution of the discretized Poisson problem $\nabla^2 u = v$ over the unit cube with periodic boundary conditions, with an irregular grid-point initialization. Provided that n is a power of 2, i.e., $k = \log_2 n$, each of the iterations consists of the two steps, the residual evaluation $(r = v - Au)$ and the correction with the multigrid V-cycle operator M $(u = u + M^k r)$. Here, A denotes the trilinear finite-element discretization of the Laplacian ∇^2. The V-cycle consists of four recursively applied steps, the projection (residual restriction), the relaxation (smoother), the interpolation (prolongation), and the residual evaluation. The detailed description of the algorithm can be found in [16].

Parallelization: The parallelization of this benchmark arises directly from the nature of the problem: the cubic grid is partitioned into blocks. The communication required is then only the exchange of elements between adjacent block faces. Our parallel implementation uses a partitioning into "matchsticks", i.e., all grid points in one dimension are on one processor and distributed block-wise in the other two dimensions.[2]

Although communication overhead for NAS-MG is not negligible, it is restricted to the exchange of planes at the PE boundaries and thus grows only as $O(n^2)$, whereas the calculation grows as $O(n^3)$. The MPI version was written from scratch in C, while the HPF/PST implementation uses the original Fortran code as basis for parallelization. In the original code, all levels of the grid are kept in one long array and the amount of data for each subsequent grid decreases exponentially. In order to distribute each grid level in this long array regularly in the matchstick fashion, we employ a user-defined (bijective)

[2]The NPB 2.0 partitions the grid block-wise in all three directions, by successively halving the grid in the x-, y-, and z-directions. The resulting Fortran/MPI code is roughly 1.7 faster than our implementation for two reasons: Fortran implementations are found to usually be 20–50% faster than C implementations and the `(BLOCK,BLOCK,BLOCK)` distribution requires less communication between PEs.

mapping of each global grid index to the index of the local data element in the appropriate grid and its owning process number. Since the mapping is only occasionally used when accessing one pointer to the current grid level, it is not expensive to use function calls instead of mapping arrays. The communication patterns are determined and stored in the first iteration and inherited to the subsequent iterations.

Qualitative Discussion: Looking at profile information (Fig. 6), it is apparent that each of the four major subroutines (RESID, PROJECT, SMOOTH, PROLONG) runs 2–10 times faster in the MPI version although the communication overhead and idle times are comparable. This difference can be partially explained by the run-time data consistency analysis performed by HPF/PST to generate the communication pattern for each grid level. The four V-cycles of Class A be seen in the trace overview in Fig. 7. The first V-cycle takes roughly 200% more time than the rest of the three because of the time spent to generate the necessary communication patterns.

Since multigrid is an iterative process, it is of particular interest to investigate the convergence of the solution. We can use the PDT Distributed Data Visualizer to visualize (Fig. 2) the convergence for a two-dimensional slice of the residual after 1 (left) and 2 (right) iterations. A well-known feature of the multigrid approach can be observed, namely that high frequency residual components of the residual, i.e., the initial discontinuities, are smoothed out after a small number of iterations, leaving low frequency residuals which will be eliminated on a coarser grid.

Performance Results: The MPI version scales well for large numbers of PEs (Fig. 12). Although the HPF/PST version is hard to evaluate due to lack of data points[3] it also scales well for smaller problem sizes. As shown in Table 1 the MPI version attains roughly 25% of MPI NAS-EP performance and the NPB 2.0 Fortran-MPI code achieved 35%. The HPF/PST version delivers 10% of HPF NAS-EP performance. The poorer performance of the latter is partially explained by the need for extensive run-time data dependency analysis to generate the communication patterns for all public routines at every grid level.

[3]The Class B timings for 128 PEs come from an earlier version of PST.

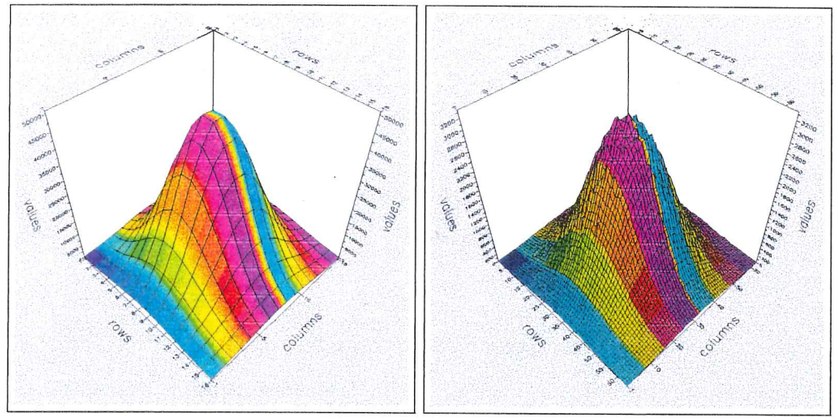


Figure 1: 3D visualization of Gaussian distributions generated with different bin sizes in the NAS-EP kernel, `(BLOCK,BLOCK)`-distributed on 64 processors.

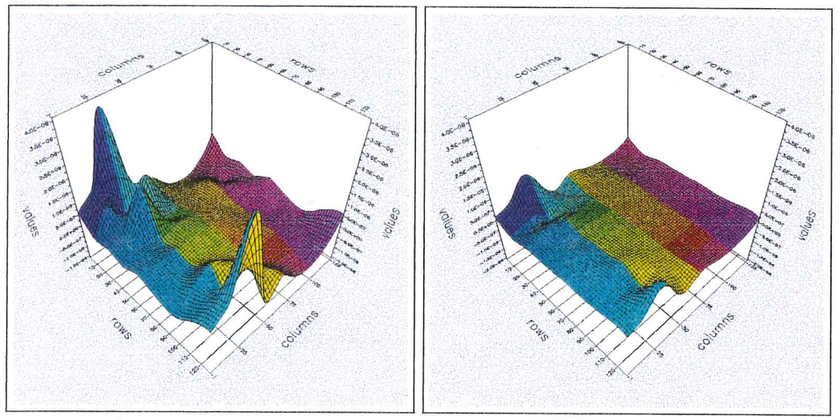


Figure 2: Evolution of the residual in the NAS-MG kernel after 1 and 2 iterations, respectively, for the $128 \times 128 \times 128$ problem, `(BLOCK,BLOCK)`-distributed on 64 processors.

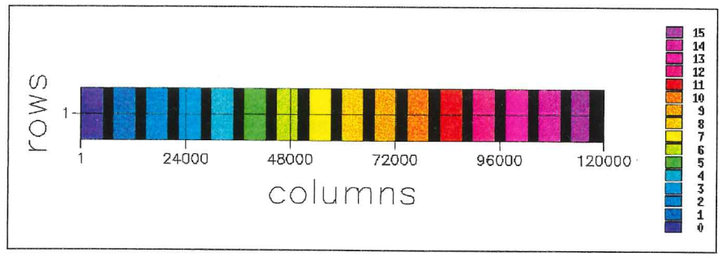


Figure 3: The `BLOCK_GENERAL` distribution of the matrix entries A in the NAS-CG kernel is illustrated. Different colors represent PEs, while gaps (unmapped entries in the distribution) are indicated as narrow black strips. Load balancing of the array makes the blocks and gaps approximately equal.

Figure 4: HPF — Embarrassingly Parallel (EP).

```
      PROGRAM EP_HPF
      ...
      PARAMETER (m=28, mk=16, mm=m-mk, nn=2**mm, nk=2**mk, nq=10)
C     seed and multiplier for the random generator LCG(a,2**46)
      PARAMETER (a=1220703125.d0, seed=271828183.d0)

      DIMENSION  dsx(nn), dsy(nn), dq(nn,nq), q(nq), x(2*nk)
!HPF$ DISTRIBUTE dsx(BLOCK), dsy(BLOCK)
!HPF$ DISTRIBUTE dq(BLOCK,*)
C     Initialization ...                                          INIT
      dsx(1:nn) = 0.d0
      dsy(1:nn) = 0.d0
      dq(1:nn,1:nq) = 0.d0

      CALL rand_init(seed, a)

!HPF$ INDEPENDENT
      DO k = 1, nn

C     Skip ahead (k-1) * 2*nk random numbers                      CALC_SEED
         CALL rand_skip (seed, a, 2*nk*(k-1))

C     Compute 2*nk random numbers                                 PRNG
         CALL rand_vect (2*nk, seed, a, x)

C     Compute Gaussian deviates by acceptance-rejection method
C     and tally counts in concentric square annuli.               GAUSS

         DO i = 1, nk
            x1 = 2.d0 * x(2*i-1) - 1.d0
            x2 = 2.d0 * x(2*i) - 1.d0
            t1 = x1 * x1 + x2 * x2
            IF (t1 .LE. 1.d0) THEN
               t2 = SQRT (-2.d0 * LOG (t1) / t1)
               t3 = x1 * t2
               t4 = x2 * t2
               l  = MAX(ABS(t3),ABS(t4)) + 1
               dq(k,l) = dq(k,l) + 1.d0
               dsx(k) = dsx(k) + t3
               dsy(k) = dsy(k) + t4
            ENDIF
         ENDDO
      ENDDO
C                                                                 COLLECT
      DO i = 1, nq
         q(i) = 0.0d0
         DO j = 1, nn
            q(i) = q(i) + dq(j,i)
         ENDDO
      ENDDO
      sx = 0.0D0
      sy = 0.0D0
      DO i = 1, nn
         sx = sx + dsx(i)
         sy = sy + dsy(i)
      ENDDO
      ...
      END
```


Figure 5: MPI — Embarrassingly Parallel (EP).

```
      PROGRAM EP_MPI
      ...
      PARAMETER (m=28, mk=16, mm=m-mk, nn=2**mm, nk=2**mk, nq=10)
C     seed and multiplier for the random generator LCG(a,2**46)
      PARAMETER (a=1220703125.d0, seed=271828183.d0)

      DIMENSION  x(2*nk)
C     Initialization ...                                          INIT
      CALL MPI_Init (ierr)
      CALL MPI_Comm_rank (MPI_COMM_WORLD, myid, ierr)
      CALL MPI_Comm_size (MPI_COMM_WORLD, numpe, ierr)

      istart = nn*myid/numpe + 1
      iend   = nn*(myid+1)/numpe
      IF (iend .GT. nn) iend = nn

      sx = 0.0D0
      sy = 0.0D0

      CALL rand_init(seed, a)
C     Skip ahead (istart-1)*(2*nk) random numbers                 CALC_SEED
      CALL rand_skip (seed, a, (2*nk)*(istart-1))

      DO k = istart, iend

C     Compute 2*nk random numbers                                 PRNG
         CALL rand_vect (2*nk, seed, a, x)

C     Compute Gaussian deviates ...                               GAUSS
         DO i = 1, nk
            x1 = 2.d0 * x(2*i-1) - 1.d0
            x2 = 2.d0 * x(2*i) - 1.d0
            t1 = x1 * x1 + x2 * x2
            IF (t1 .LE. 1.d0) THEN
               t2 = SQRT (-2.d0 * LOG (t1) / t1)
               t3 = x1 * t2
               t4 = x2 * t2
               l  = MAX(ABS(t3),ABS(t4)) + 1
               q(l) = q(l) + 1.d0
               sx = sx + t3
               sy = sy + t4
            ENDIF
         ENDDO
      ENDDO

C     Collect sums ...                                            COLLECT
      CALL MPI_Reduce (sx, sx, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
     &                 0, MPI_COMM_WORLD, ierr)
      CALL MPI_Reduce (sy, sy, 1, MPI_DOUBLE_PRECISION, MPI_SUM,
     &                 0, MPI_COMM_WORLD, ierr)
      CALL MPI_Reduce (q, q, nq, MPI_DOUBLE_PRECISION, MPI_SUM,
     &                 0, MPI_COMM_WORLD, ierr)
      ...
      END
```

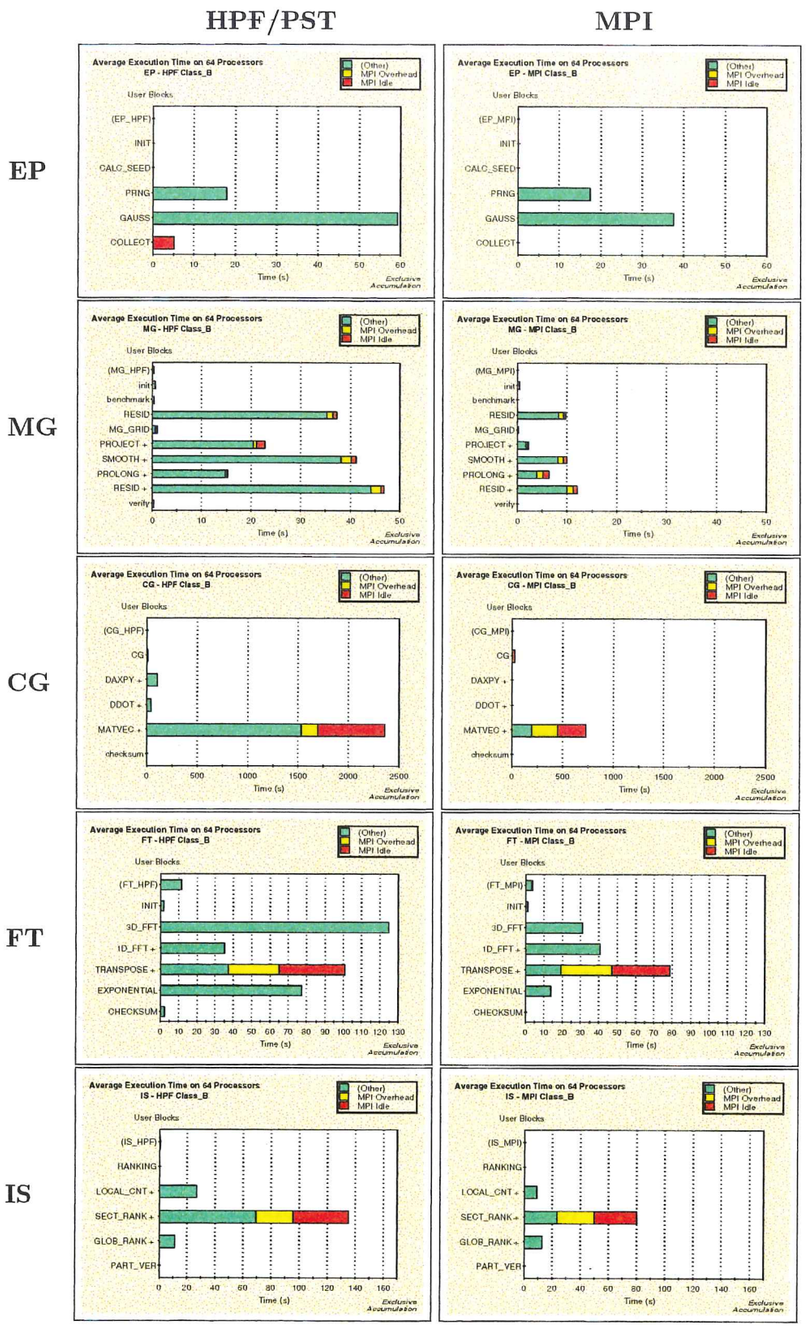


Figure 6: Execution time profiles of NPB Class B runs on 64 processors of the NEC Cenju-3. (Exclusive accumulation into user-defined tasks, breakdown by MPI utilization.)

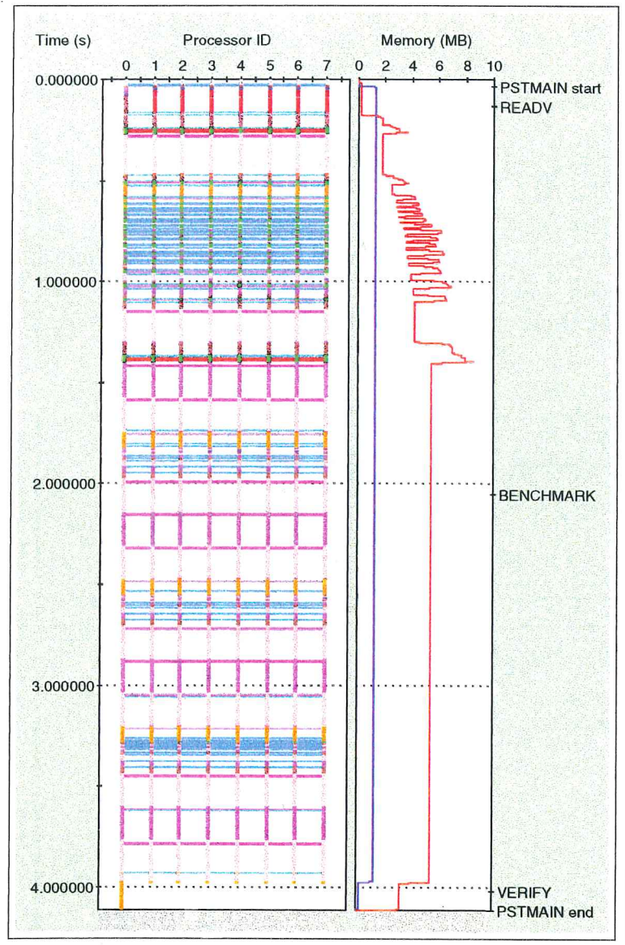


Figure 7: Trace of the NAS-MG kernel HPF/PST version from 8 processors: the first iteration (0.2–1.6s) is much longer than the other three (e.g., the second cycle 1.6–2.3s) due to run-time data consistency analysis.

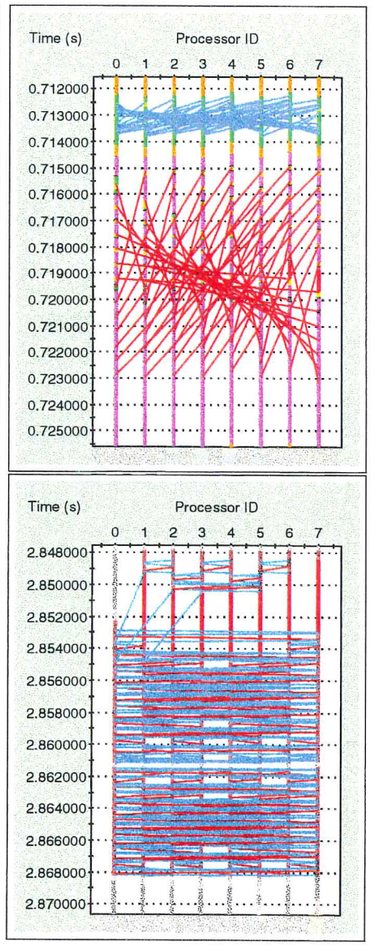


Figure 8: Execution traces from HPF/PST (top) and MPI (bottom) code segments of the NAS-IS kernel containing all-to-all key density exchanges on 8 processors.

Figure 9: Conjugate gradient and power method (CG program).

```
      PROGRAM CGM
      ...
c     Generate n × n sparse matrix A with nnz non-zeroes stored in a, rowidx, colstr
      CALL makea (n, nnz, a, rowidx, colstr, ...)

c     x = [1, 1, ..., 1]^T
      x(1:nn) = 1.0D0
c     Do niter iterations of the Power Method (Timed Benchmark)
      DO it = 1, niter
c        Solve the system Az = x by the CG method and return residual norm: resid = ||r||
         CALL cgsol(n, a, rowidx, colstr, nnz, x, z, resid, nitcg, r, p, q)
c        Eigenvalue estimate: ζ = λ + 1/(x^T z)
         zeta = lambda + 1.0D0/DDOT(n, x, z)
         PRINT *, it, resid, zeta
c        Renormalize x: x = z/||z||
         znorminv = 1.0D0/SQRT(DDOT(n, z, z))
         CALL DSCAL(n, znorminv, z, x)
      ENDDO
      ...
      END
```

```
c     Conjugate Gradient Method to solve Az = x
      SUBROUTINE cgsol(n, a, rowidx, colstr, nnz, x, z, resid, nitcg, r, p, q)
      ...
c     z = 0;   r = x;   p = r;   ρ = r^T r
      z(1:nn) = 0.0D0
      CALL DCOPY(n, x, r)
      CALL DCOPY(n, r, p)
      rho = DDOT(n, r, r)

c     Do nitcg iterations of the Conjugate Gradient Method
      DO iter = 1, nitcg
c        Sparse matrix - vector multiplication q = Ap
         CALL matvec(n, nnz, a, rowidx, colstr, p, q)
c        α = ρ/(p^T q)
         alpha = rho/DDOT(n, p, q)
c        z = z + αp
         CALL DAXPY(n, alpha, p, z)
c        ρ0 = ρ
         rho0 = rho
c        r = r − αq
         CALL DAXPY(n, -alpha, q, r)
c        ρ = r^T r
         rho = DDOT(n, r, r)
c        β = ρ/ρ0
         beta = rho/rho0
c        p = r + βp
         CALL DAYPX(n, beta, r, p)
      ENDDO

c     Calculate residual norm: resid = ||r|| = ||x − Az||
      CALL matvec(n, nnz, a, rowidx, colstr, z, r)
      CALL DAYPX(n, -1.0D0, x, r)
      resid = SQRT(DDOT(n, r, r))

      END
```


Figure 10: Matrix-vector multiplication in NAS-CG: HPF/PST (top) and MPI.

```
      EXTRINSIC (PST_LOCAL) SUBROUTINE matvec(n, nnz, a, rowidx, colstr,
     >                                        x, y, matmap, myid)
!PST$ SUB_SPEC PUBLIC SAVECOM

      INTEGER rowidx(*), colstr(*), n, nnz, matmap(*)
      DOUBLE PRECISION a(*), x(*), y(*)
      INTEGER i, j, k
      DOUBLE PRECISION tmp, ytmp(n)

!HPF$ DISTRIBUTE ytmp(BLOCK)
!PST$ DISTRIBUTE a(BLOCK_GENERAL(matmap))
!PST$ DISTRIBUTE rowidx(BLOCK_GENERAL(matmap))

      i = matmap(myid+1) + 1

!HPF$ ALIGN 20 WITH ytmp(j)
      DO 20 j = 1, n
         tmp    = 0.0d0
         DO 10 k = 1, colstr(j+1) - colstr(j)
            tmp = tmp + a(i)*x(rowidx(i))
            i = i + 1
 10      CONTINUE
         ytmp(j) = tmp
 20   CONTINUE

c     The only communication takes place here!!
      DO 30 j = 1, n
         y(j) = ytmp(j)
 30   CONTINUE

      END
```

```
      SUBROUTINE matvec(n, nnz, a, rowidx, colstr, x, y,
     >                  rcounts, displ, ist_p, iblock_p, iend_p)

      INCLUDE 'mpi.incl'

      INTEGER n, nnz, rowidx(*), colstr(*)
      INTEGER rcounts(*), displ(*), ist_p, iblock_p, iend_p
      DOUBLE PRECISION  a(*), x(*), y(*)

      INTEGER i, j, k, ierror
      DOUBLE PRECISION tmp

      CALL MPI_Allgatherv (x(ist_p), iblock_p, MPI_DOUBLE_PRECISION,
     >        x, rcounts, displ, MPI_DOUBLE_PRECISION, idoc, ierror)

      i = 1
      DO j = ist_p, iend_p

         tmp = 0.0d0
         DO k = 1, colstr(j+1) - colstr(j)
            tmp = tmp + a(i)*x(rowidx(i))
            i = i + 1
         ENDDO
         y(j) = tmp

      ENDDO

      END
```

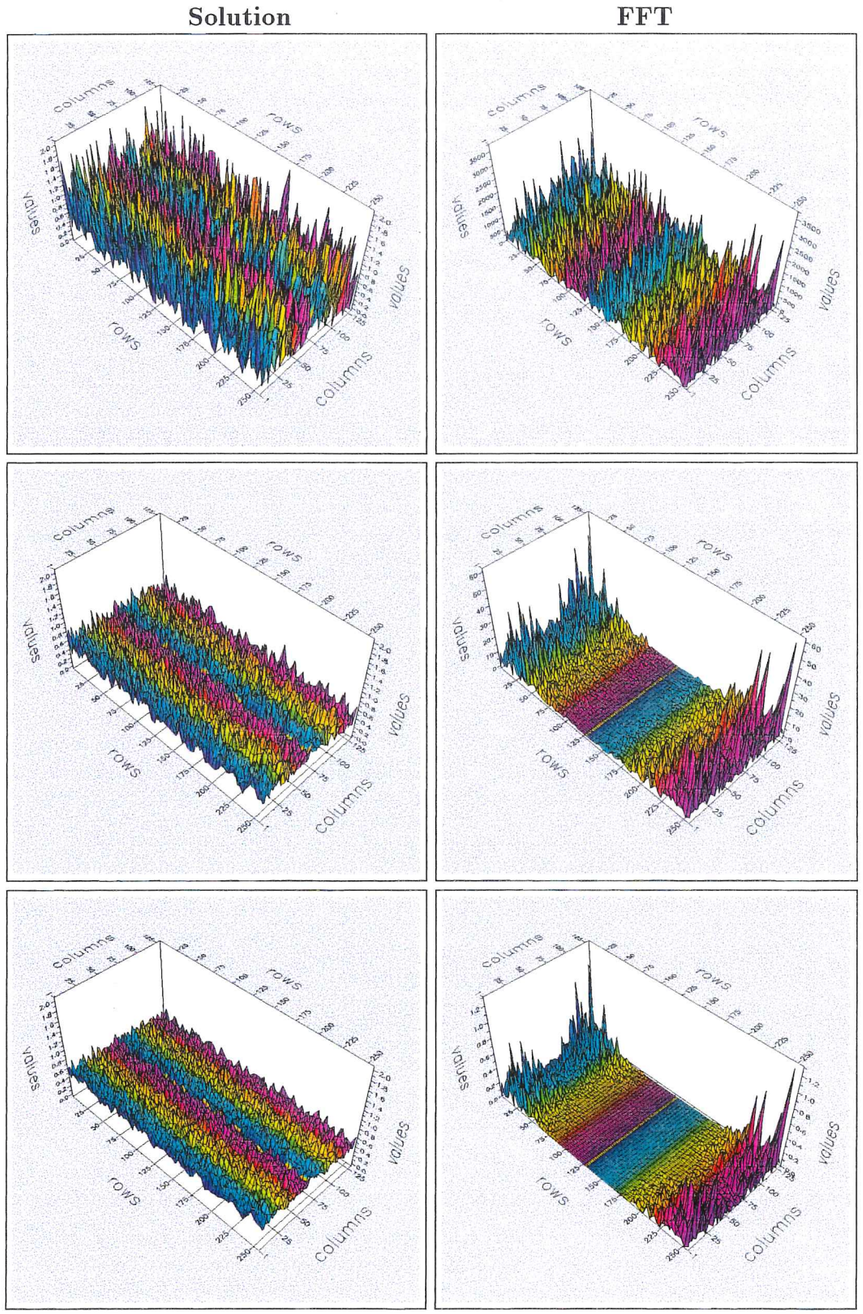


Figure 11: Evolution of the solution and its Fourier transform in the NAS-FT kernel for $t = 0$ (top), $t = 3$ (center), and $t = 6$ (bottom) as squared absolute value slices.

3.3 Conjugate Gradient (CG) kernel

Algorithm: The NAS-CG kernel uses the conjugate gradient algorithm in combination with the inverse power method to approximate the smallest eigenvalue of a sparse, symmetric, positive definite matrix A, which is generated with a random pattern of nonzeroes. *niter* iterations of the power method are employed to approximate the smallest eigenvalue. Within each iteration *nitcg* iterations of the conjugate gradient method are performed to solve the linear system $Ax = z$ (Fig. 9). Each NAS-CG iteration consists of 3 vector updates (DAXPY), 2 scalar products (DDOT), and, additionally, one matrix-vector multiplication (MATVEC). The method is further described in the literature [17].

Parallelization: The sparse matrix A is generated — as in the sequential code — by sorting elements by their column index. It is stored in the Compressed Sparse Row (CSR) format, i.e., matrix values are stored in `a`, their row index in `rowidx`, and the pointer, where column i starts, in `colstr`(i). Matrix columns are distributed block-wise over all the processors using the `BLOCK_GENERAL` distribution of HPF/PST which allows arbitrary sizes of the local blocks and gaps (Fig. 3). In the HPF/PST implementation, each processor keeps a full copy of all n-vectors (five of them are necessary for the conjugate gradient method). The matrix-vector multiplication $y = Ax$ is calculated by performing scalar products of matrix rows and the incoming vector x, therefore the only communication needed is for gathering updated vector portions of the resultant vector y on all processors so that each has its own copy (Fig. 10).

Other HPF/PST versions with distributed vectors were also tested: (a) duplicating x on all processors or (b) only the needed segments of x on the requesting processors before calculating the proper matrix-vector multiplication. In the latter time is saved in calculating the vector updates but lost when collecting partial sums in the scalar products (DDOT). Although the MPI implementation of this method turned out to be faster (Fig. 10), in the HPF/PST code the benefit of having distributed vectors is lower, i.e., the parallel scalar product using an optimized Fortran BLAS routine is faster than with the corresponding HPF/PST code. Furthermore, since the NAS-CG problem under investigation is fairly dense, the cost of having local n-vectors is not appreciable when compared to the matrix-vector multiplication. The execution time for MATVEC is about 5–10% higher with distributed vectors; this applies for both cases, (a) and (b). Thus for the HPF/PST code, the lack of data locality favors the implementation of global communication and the duplication of the vectors.

Much of the sequential program has been rewritten to reference the data structure with local indices in the MPI version. We can take advantage of the `MPI_Allgatherv` global communication routine to gather the vector x.

Qualitative Discussion: The execution profile (Fig. 6) reveals that more than 90% of the time is spent for the matrix-vector multiplication (MATVEC), and therein most (HPF/PST 40%, MPI 70%) for MPI Idle and MPI Overhead which are associated with communication. Time spent for the other vector operations DAXPY and DDOT and computation of the checksum CHECKSUM is negligible, explaining the previously mentioned small performance differences between the duplicated or distributed vector versions.

Performance Results: As expected from the discussion above, the performance completely depends on the efficiency of the parallel implementation of the sparse matrix-vector multiplication. The absolute performance of the MPI and HPF/PST versions is low (Table 1), and the scalability of the algorithm is reasonable for small numbers of PEs, but flattens out for large numbers (Fig. 12). This deficiency can be partially attributed to the irregular communication pattern in that problem. The results for the MPI implementation might be improved by restructuring the matrix-vector multiplication such that computation and communication are overlapped. Due to the lack of data locality, another data decomposition might be more efficient for both parallel codes.

3.4 Fourier Transform (FT) kernel

Algorithm: In the NAS-FT kernel a set of 3-D fast-Fourier transforms is employed to numerically solve the heat equation: $\partial_t u(x,t) = \alpha \nabla^2 u(x,t)$. By applying a 3-dimensional Fourier transform to each side, we get $\partial_t \tilde{u}(p,t) = -4\alpha\pi^2 |p|^2 \tilde{u}(p,t)$, where $\tilde{u}(p,t)$ is the spatial Fourier transform of $u(x,t)$. The time t dependent solution in Fourier space is a simple multiplication with an exponential factor: $\tilde{u}(p,t) = \exp(-4\alpha\pi^2 |p|^2 t)\ \tilde{u}(p,0)$. The NAS-FT kernel solves the discretized version of this PDE where $u(x,t)$ becomes a $n_1 \times n_2 \times n_3$ dimensional complex array $U(k,l,m,t)$ (for some fixed time t). As initial configuration $U(k,l,m,0)$ $2n_1n_2n_3$ random numbers are generated. After one forward transform, for each step t the multiplication with the exponential factor raised to the power of t and the backward transform must be calculated. The results are verified by computing a certain checksum at each iteration.

Parallelization: The parallelization of the generation of pseudo-random numbers was already outlined for the NAS-EP kernel. The three-dimensional array U is distributed in the third dimension, with blocks of planes assigned to each processor. In the HPF/PST version this results in a `(*,*,BLOCK)` distribution of the 3-dimensional arrays, or equivalently, in MPI with slices which are n_3/n_p thick on each processor.

In a first step, multiple 1-dimensional FFTs can be performed locally in the first two dimensions. Before the final set of 1-dimensional FFTs in the last dimension can be performed, the 3-dimensional array must be transposed. This transposition implies an all-to-all exchange of data, where each processor must send sub-arrays to every other processor. In HPF, the transpose is supported by a library function[4]; with MPI we can exchange the data with the help of the global communication function `MPI_Alltoall`. After multiplying the transformed array with an exponential term raised to the power of t the inverse Fourier transform has to be performed. To halve the communication time we calculate the Fourier transform on the transposed field and compute the exponential factor in accordance with the transposed field. Another optimization for computer systems with large local memories is to compute the exponential term only once to the power $t = 1$ and store it into an additional array for convenience. This method was applied for larger processor numbers with a small speed-up of about 3–4%.

[4]As a PST-extension to HPF, the transpose is implemented directly in MPI.

Qualitative Discussion: As seen from the profile (Fig. 6) the execution time for the random initialization of the solution (INIT) and the computation of the checksum (CHECKSUM) are negligible.[5] The calculation of the 3-dimensional FFT (3D_FFT) consists of data assignments, the separately-shown one-dimensional FFT (1D_FFT) and the the transposition (TRANSPOSE). Since for both the HPF/PST and MPI versions the 1D_FFT is calculated within an optimized Fortran library function, the execution time is about the same in both cases. For Class B on 64 processors, the intrusion of the instrumentation is high (about 4s) because of about 7 million calls of 1D_FFT. In HPF/PST, the transposition is realized with a call to a library function, whereas MPI uses a slightly faster `MPI_Alltoall`-based TRANSPOSE. The MPI-Overhead and MPI-Idle time in the transposition are roughly equal. The contribution of "other" (copying to buffers) is higher in the HPF/PST library function. As seen from the 3D_FFT bar, the MPI code has significantly fewer data assignments, because the 1D_FFT calls operate with pointers directly on the three-dimensional array, while in the HPF implementation the data-segments are copied before and after to and from one-dimensional arrays.

We can easily use the PDT Distributed Data Visualizer to explain the time-evolution of the solution. In Fig. 11, 2-dimensional slices of the solution and the Fourier transform are displayed for $t = 0, 3, 6$ as squared absolute values. The initial random solution, with values scattering in the interval $(0, 2)$ flattens out after 6 time steps. The explanation can be seen from the Fourier transform: the multiplication with the exponential factor results in a spatial concentration in the corners, which due to period boundary conditions can be identified as the coordinate origin.

Performance Results: Memory requirements limit the processor range to a minimum of 16 for Class A and 64 for Class B, respectively. Since the explicit calculation of the exponential factor saves one three-dimensional array, the MPI program even runs on 8 (Class A) and 32 (Class B) processors, respectively.[6] In the above mentioned ranges the program is completely scalable (Fig. 12), despite the fact that the most significant routine, the transposition, is communication intensive. The underlying all-to-all communication itself scales to large numbers of PEs in this problem, since individual PEs have progressively less data to exchange. As revealed by Table 1, we get about 1/3 (1/5) of the NAS-EP performance for the MPI (HPF/PST) version. The performance of our MPI implementation is slightly better (about 25% for Class A and 10% for Class B) than the code distributed with NPB 2.0. This is in accordance with our previous experience with FFTPACK from NETLIB [18] compared with the original FFT.[7]

[5]The evaluation of the checksum causes a small imbalance of the processors, due to unequal distribution of the 1024 numbers on the PEs.

[6]As observed for larger processor numbers, storing the exponential term into an additional array yields to a negligible speed-up of about 3-4%.

[7]For each dimension, one copy of the FFTPACK initialization routine (CFFTI) and forward and inverse transform of a complex periodic sequence (CFFTF, CFFTB) is modified to only accept array sizes which are powers of two. FFTPACK (Version 4, April 1985) is by the National Center for Atmospheric Research, Boulder, Colorado.

3.5 Integer Sort (IS) kernel

Algorithm: The NAS-IS kernel sorts an array of N keys, which we refer to as *keys*, in the range $[0, B_{max} - 1]$. The keys are sorted when $(\forall i : K_{i-1} \leq K_i \leq K_{i+1})$. The benchmark requires only a ranking of the unsorted sequence, not that the keys are actually exchanged. The fundamental algorithm used is the bucket sort, e.g., described in [19]. The proper ranking of the keys is verified in an untimed post-processing step.

Parallelization: In the parallel implementation, the keys are distributed block-wise as described in [1], namely, into arrays *keys* of length $N_{pp} = N - (p-1)N_p$ (on the last processor) and $N_p = \lfloor N/p + 0.5 \rfloor$ on all others. A key density array, *key_den* of length B_{max} is allocated in each processor.

A local bucket sort first determines the density of the local keys (LOCAL_CNT). In the next step (SECT_RANK), each processor is assigned to a block of the density array and collects the local density information from every other PE for that block. In the MPI version, the latter is performed with a `MPI_Alltoall`, in the HPF/PST version, the communication is generated automatically. In order to find the cumulative number of global keys, the *nprocs*-vector containing the number of keys sent from each PE for their given block is summed over all PEs (in the MPI version with `MPI_Allreduce`). This vector then denotes the rank of the first sorted key in any sector. Subsequently each PE calculates the global key density for its given block. The blocks of global key density values are communicated back to every PE which then, in the final step (GLOB_RANK), completes the global ranking of its allotted keys. Afterwards, the ranking is partially verified by a comparison with reference values (PART_VER). The full verification of the proper ranking of all keys was not performed for Class A and Class B.

Qualitative discussion: From the PMA profile presented with the Execution Statistics Display (Fig. 6) it is clear that the local bucket sort (LOCAL_CNT) and the final global ranking (GLOB_RANK) are communication-free, as indicated by the lack of MPI overhead and idle time in both the HPF/PST and MPI versions.

In the section ranking (SECT_RANK), the communication overhead and idle time for both the MPI and HPF/PST versions are similar, reflecting the similar choice of collective communication primitives. In the MPI version, a combination of `MPI_Alltoall` and `MPI_Allreduce` primitives is used, while in the HPF/PST version the communication pattern is generated at run-time during the data-dependency analysis. The difference in the underlying communication patterns can be seen in the program execution traces in Fig. 8. The difference between the relatively complicated implementation of `MPI_Alltoall` (bottom) contrasts with the rather simplistic automatically generated communication from the HPF/PST version (top) in which every PE posts a send to every other PE. It is also apparent from the execution traces that the HPF/PST-generated all-to-all communication is competitive with the optimized MPI primitive.

From this analysis we have good reason to believe that the parallel performance of the two versions will be similar, however the initial version of the HPF/PST code is in fact still an order of magnitude slower than the MPI version and is unable to calculate the large Class A and Class B problems sizes. This HPF/PST overhead comes from the

management of large auxiliary data structures which help determine the proper communication pattern. Some work went into a step-by-step refinement of the HPF/PST code to optimize communication based on the above analysis techniques.

Performance Results: As anticipated from the qualitative analysis above, the performance (Fig. 12) of the MPI and HPF/PST versions is quite similar. The scalability of the algorithm is reasonable for small numbers of PEs, but poor for large numbers. This reflects the fact that the amount of integer calculation is relatively small compared to the communication overhead and collective communication used does not scale well to large numbers of PEs. The fact that the results [13] for other machines are considerably more scalable than these is an indication that a relatively "naive" algorithm was employed in this case and that there was no attempt at overlapping communication and calculation.

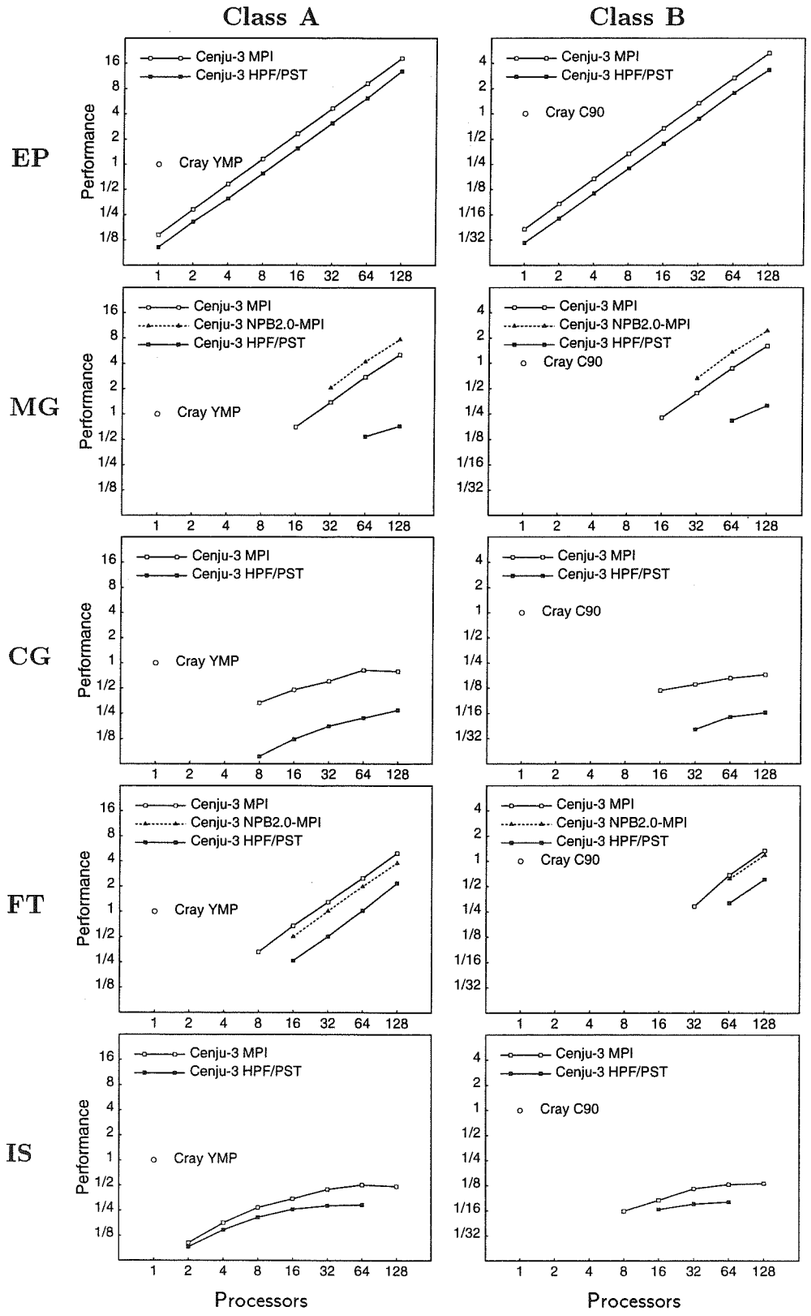


Figure 12: Performance of the NAS kernels on Cenju-3 relative to Cray reference [13].

Table 1: Performance of the NAS kernels on the Cenju-3. For MG and FT, results for NPB 2.0 MPI Fortran code are also quoted in square brackets. (* in the FT-MPI code indicates explicit calculation of the exponential factor.)

Kernel Size MInst.	PEs	**MPI** CPU time (s)		MFlop/s		**HPF/PST** CPU time (s)	MFlop/s
	128	27.4		3670.8		43.5	2318.3
	64	54.8		1840.8		82.4	1224.4
EP	32	109.6		920.5		168.5	598.6
Class B	16	219.1		460.3		331.5	304.0
100900	8	438.3		230.1		658.5	153.2
	4	876.7		115.0		1311.8	76.9
	2	1753.4		57.5		2619.1	38.5
	1	3506.9		28.8		5091.7	19.8
	128	6.9		3876.1		9.7	2729.5
	64	13.7		1945.5		20.4	1301.4
EP	32	27.4		973.2		40.8	652.4
Class A	16	54.8		486.7		81.7	326.5
26680	8	109.6		243.4		163.3	163.3
	4	219.1		121.7		326.6	81.6
	2	438.3		60.8		617.3	43.2
	1	876.7		30.4		1234.7	21.6
MG	128	21.1	[13.9]	922.3	[1398.2]	108.2	179.8
Class B	64	38.8	[25.1]	501.5	[774.1]	163.2	119.2
19461.6 (NPB2)	32	76.4	[51.3]	254.7	[379.1]		
	16	149.5		130.1			
MG	128	4.4	[2.9]	873.2	[1307.5]	31.0	123.9
Class A	64	8.1	[5.3]	474.3	[721.5]	41.1	93.4
3842.3 (NPB2)	32	16.1	[10.9]	238.6	[355.9]		
	16	31.7		121.2			
CG	128	681.9		80.5		1936.9	28.3
Class B	64	751.3		73.1		2190.5	25.1
54890	32	889.2		61.7		3065.1	17.9
	16	1056.3		52.0			
	128	15.0		100.0		43.5	34.6
CG	64	14.4		103.9		53.9	27.9
Class A	32	19.5		76.7		67.5	22.3
1508	16	24.8		60.6		96.9	15.5
	8	35.6		42.2		155.1	9.7
FT	128	82.9	[93.5]	1133.9	[1005.8]	182.1	516.1
Class B	64	161.4	[179.5]	582.4	[524.0]	349.1	269.3
94053.1 (NPB2)	32	*383.3		*245.3			
	128	5.8	[7.6]	1294.5	[999.0]	13.2	570.9
FT	64	11.5	[14.5]	655.2	[520.8]	28.2	358.0
Class A	32	22.3	[28.6]	337.8	[263.0]	57.0	132.1
7541.4 (NPB2)	16	42.6	[57.5]	176.6	[131.0]	111.5	67.7
	8	*87.8		*85.9			
	128	96.8		32.5			
IS	64	99.8		31.5		162.1	19.4
Class B	32	112.4		28.0		170.7	18.4
3150	16	154.0		20.4		199.1	15.8
	8	210.1		15.0			
	128	23.6		33.0		40.2	19.4
	64	22.6		34.4		39.3	19.8
IS	32	25.6		30.4		40.2	19.4
Class A	16	33.3		23.4		44.6	17.5
781.2	8	42.3		18.4		55.9	14.0
	4	64.9		12.0		78.6	9.9
	2	113.1		6.9		125.5	6.2

4 Summary

HPF and MPI versions of the NAS Parallel Benchmark kernels EP, MG, CG, FT, and IS have been presented. We have demonstrated the features of the *Annai* integrated tool environment which has been used extensively in the development of the parallel implementations. The possibility of step-wise parallelization from a simple HPF data-parallel formulation to a progressively improved hybrid data-parallel and explicit message-passing version has been sketched through analysis techniques applied to the final HPF/PST and MPI versions.

Parallel software development with *Annai* can be summarized as follows: starting with a sequential Fortran version of the code, a 'naïve' parallel version is written in HPF. Valuable language extensions to complement the current HPF specification are provided by the Parallelization Support Tool, such as a comprehensive set of data distributions and support for explicit loop distribution. The Parallel Debugging Tool provides key assistance during data-parallel program development, e.g., with synchronizing breakpoints and distributed data visualization. Profiling with the Performance Monitor and Analyzer identifies the regions of the computation, where parallelization and communication overheads are most critical, and targetted execution tracing is also provided for detailed investigation of underlying communication bottlenecks. Since initial program versions may lack both performance and scalability, tuning typically starts with refining data and loop distributions for improved load balance or more efficient execution. Meanwhile the tools are used collectively for debugging, performance analysis and verification. Further optimization may include the insertion of MPI calls into the code, replacing automatically-generated communication primitives with explicit message-passing. Such hybrid HPF/MPI codes also benefit from deterministic program replay and deadlock detection facilities developed to support explicit message-passing programming.

We have evaluated this step-wise approach using *Annai*, and have compared the NAS kernels written in HPF/PST against equivalent MPI implementations of the codes (including recently published NPB 2.0, which were re-written from scratch). The resulting performance and scalability compare favorably, in view of the comparative ease of data-parallel programming with respect to the explicit message-passing paradigm.

Acknowledgments: The Multigrid kernel was parallelized with the help of U. Kühn (Universität Münster, Germany), and the Integer Sort kernel with the help of F. Sukup (Technische Universität Wien, Austria).

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, March 1994.

[2] High Performance Fortran Forum. High Performance Fortran language specification: Version 1.0. *Scientific Programming*, 2(1&2), 1993.

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

[4] Andreas Müller and Roland Rühl. Extending High Performance Fortran for the support of unstructured computations. In *Proceedings of the 9th International Conference on Supercomputing (ICS'95, Barcelona, Spain)*, pages 127–136. ACM, July 1995. ISBN: 0-89791-728-6.

[5] H. P. Zima, P. Brezany, B. M. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification: Version 1.1. Technical Report ACPC/TR92-4, Austrian Center for Parallel Computation, Vienna, Austria, March 1992.

[6] Barbara Chapman, Hans Zima, and Piyush Mehrotra. Extending HPF for advanced data-parallel applications. *IEEE Parallel & Distributed Technology*, pages 59–70, October 1994.

[7] Christian Clémençon, Karsten M. Decker, Vaibhav R. Deshpande, Akiyoshi Endo, Josef Fritscher, Paulo A. R. Lorenzo, Norio Masuda, Andreas Müller, Roland Rühl, William Sawyer, Brian J. N. Wylie, and Frank Zimmermann. Tool-supported parallel application development. In *Proceedings of the 15th International Phoenix Conference on Computers and Communications (Phoenix, AZ, USA)*, pages 294–302. IEEE Computer Society Press, March 1996. ISBN: 0-7803-3255-5.

[8] Christian Clémençon, Akiyoshi Endo, Josef Fritscher, Andreas Müller, Roland Rühl, and Brian J. N. Wylie. The 'Annai' environment for portable distributed parallel programming. In Hesham El-Rewini and Bruce D. Shriver, editors, *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS-28, Maui, Hawaii, USA), Volume II*, pages 242–251. IEEE Computer Society Press, January 1995. ISBN 0-8186-6935-7.

[9] Yoshiki Seo, Tsunehiko Kamachi, Kenji Suehiro, Masanori Tamura, Andreas Müller, and Roland Rühl. Kemari: a portable HPF system for distributed memory parallel machines. Technical Report CSCS-TR-95-04, Centro Svizzero di Calcolo Scientifico, CH-6928 Manno, Switzerland, May 1995.

[10] Christian Clémençon, Josef Fritscher, and Roland Rühl. Visualization, execution control and replay of massively parallel programs within Annai's debugging tool. In Vincent Van Dongen, editor, *Proceedings of the High Performance Computing*

Symposium, (HPCS'95, Montréal, Canada), pages 393–404. Centre de recherche informatique de Montréal, July 1995. ISBN: 2-921316-12-9.

[11] Brian J. N. Wylie and Akiyoshi Endo. The Annai/PMA Performance Monitor and Analyzer. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'96, San Jose, CA, USA)*, pages 186–191. IEEE Computer Society Press, February 1996. ISBN: 0-8186-7235-8.

[12] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report RNR-95-020, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, December 1995.

[13] Subhash Saini and David H. Bailey. NAS parallel benchmarks results 12-95. Technical Report RNR-95-021, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, December 1995.

[14] Applied Parallel Research. APR releases xHPF 2.1, world's first HPF to turn in NAS benchmark results. HPC Select News article, July 1995. Further information available from APR, Placerville, CA, USA.

[15] Larry F. Meadows, Douglas Miles, and Mark Young. Performance results of several High Performance Fortran benchmarks. In *Proceedings of the 9th International Parallel Processing Symposium (Santa Barbara, CA, USA)*, pages 516–517. IEEE Computer Society Press, April 1995. ISBN 0-8186-7074-6.

[16] W. Hackbusch. *Multi-Grid Methods and Applications.* Springer Verlag, 1985.

[17] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[18] P. N. Swarztrauber. FFTPACK (version 4). NETLIB, April 1985. National Center for Atmospheric Research, Boulder, CO 80307, USA.

[19] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 1. Addison-Wesley, second edition, 1973.

1994

TR-94-04 W. SAWYER, G. KREISS, D. SORENSEN AND J. LAMBERS: Arnoldi Method Applied to Burgers' Equation. (May 1994)

TR-94-05 E. DE STURLER: A Performance Model for Krylov Subspace Methods on Mesh-based Parallel Computers. (May 1994)

TR-94-06 R. GRUBER: PE2AR: Program Environments for Engineering Applications and Research. (August 1994)

TR-94-07 B. J. N. WYLIE AND A. ENDO: Design and Realization of the Annai Integrated Parallel Programming Environment Performance Monitor and Analyzer. (August 1994)

TR-94-08 A. MÜLLER AND R. RÜHL: Extending High Performance Fortran for the Support of Unstructured Computations. (November 1994)

TR-94-09 C. CLÉMENÇON, J. FRITSCHER AND R. RÜHL: Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. (November 1994)

TR-94-10 E. GERTEISEN: Implementation of Finite Volume Fluid Solvers into the PE2AR Database Environment. (December 1994)

TR-94-11 E. GERTEISEN: A Generic Data Structure for the Communication of Arbitrary Domain Splitted Mesh Topologies. (December 1994)

1995

TR-95-01 C. CLÉMENÇON, J. FRITSCHER, M. MEEHAN, AND R. RÜHL: An Implementation of Race Detection and Deterministic Replay with MPI. (January 1995)

TR-95-02 K. DECKER AND S. FOCARDI: Technology Overview: A Report on Data Mining. (February 1994)

TR-95-03 C. CLÉMENÇON, K. DECKER, V. DESHPANDE, A. ENDO, J. FRITSCHER, N. MASUDA, A. MÜLLER, R. RÜHL, W. SAWYER, B. J. N. WYLIE, AND F. ZIMMERMANN: Tool-Supported Development of Parallel Application Kernels. (April 1995)

TR-95-04 Y. SEO, T. KAMACHI, K. SUEHIRO, M. TAMURA (NEC CENTRAL RESEARCH LAB., KAWASAKI, TOKYO) AND A. MÜLLER, R. RÜHL (CSCS): Kemari: a Portable HPF System for Distributed Memory Parallel Machines. (June 1995)

TR-95-05 A. ENDO AND B. J. N. WYLIE: Annai/PMA Instrumentation Intrusion Management of Parallel Program Profiling. (November 1995)

TR-95-06 P. ACKERMANN AND U. MEYER: Prototypes for Audio and Video Processing in a Scientific Visualization Environment based on the MET++ Multimedia Application Framework. (June 1995)

TR-95-07 M. GUGGISBERG, I. PONTIGGIA AND U. MEYER: Parallel Fractal Image Compression Using Iterated Function Systems. (May 1995)

1996

TR-96-01 W. P. PETERSEN: A General Implicit Splitting for Stabilizing Numerical Simulations of Langevin Equations. (February 1996)

CSCS/SCSC — Via Cantonale — CH-6928 Manno — Switzerland
Tel: +41 (91) 610 8211 — Fax: +41 (91) 610 8282

CSCS/SCSC — ETH Zentrum, RZ — CH-8092 Zürich — Switzerland
Tel: +41 (1) 632 5574 — Fax: +41 (1) 632 1104

CSCS/SCSC WWW Server: http://www.cscs.ch/