Centro Svizzero di Calcolo Scientifico

CSCS

Swiss Scientific Computing Center

CSCS-TR-94-01

ETH Eidgenössische Technische Hochschule Zürich

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

Application-Driven Development of an Integrated Tool Environment for Distributed Memory Parallel Processors

C. Clémençon, K.M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler and B.J.N. Wylie

April 6, 1994

Technical Report

Application-Driven Development of an Integrated Tool Environment for Distributed Memory Parallel Processors

CSCS-TR-94-01

April 6, 1994

Abstract: The *Joint CSCS-ETH/NEC Collaboration in Parallel Processing* comprises the development of an integrated tool environment together with applications and algorithms for distributed memory parallel processors (DMPPs). Tool and application developers interact closely: the requirements of the tools are defined by the needs of the application developers, and once an application requirement becomes an integral part of the tool environment, the tools ease parallelization of similar applications and whole application classes. Additional features of the project are the use of a standardized DMPP high-level programming language (HPF) and low-level message passing interface (MPI). The tool environment integrates parallelization support, a parallel debugger, and a performance monitor and analyzer. Applications already investigated include some of those currently considered difficult to parallelize on DMPPs.

In this paper we summarize the tool and application development efforts and show preliminary performance results of three applications effectively parallelized on two DMPP platforms with the assistance of our tool environment.

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
CSCS–ETH, Section of Research and Development (SeRD),
Swiss Scientific Computing Center,
La Galleria, CH-6928 Manno, Switzerland
decker@cscs.ch

† NEC SX-Center, Switzerland

Technical Report

(MPI) [MPI93]. The applications can either be developed using MPI, or at a higher level using our parallelization support tool, i.e., a compiler for High Performance Fortran (HPF) [HPF93] and a few extensions defined by us. Although our first major development platform is a NEC Cenju-3 DMPP, we prove portability of tools and applications by also supporting networks of workstations for development. An Intel Paragon has also been available for validation of applications developed using our compilation system. First performance measurements presented in this paper were collected on the NEC Cenju-2 (an experimental prototype of the Cenju-3), and on the Intel Paragon.

This paper summarizes our current tool and application development efforts, with emphasis on the interaction between tool and application developers. We first outline the integrated tool environment and its components. Then the work on applications and algorithms is summarized by describing how several example applications were parallelized both by using low-level message passing and by using high-level data parallel programming. Finally, performance measurements on the above two DMPP platforms are shown to demonstrate the practical portability of our software, and how application developers influence the functionality of the tool environment.

2 The Tool Environment

2.1 Design Objectives

Our major objectives in designing the tool environment can be summarized as follows:

- Application-oriented tool design. That is, the tools are developed in a sequence of prototypes, and a team of application developers continuously provide feedback when using and testing the prototypes.

- Designing and implementing an integrated tool environment which supports parallel program development in a high-level data-parallel MIMD language and/or using explicit low-level message passing.

- Use of standardized programming languages and machine interfaces. At the high level, HPF and possible future HPF extensions are supported. At the low level, MPI serves as the machine interface.

- Proposing and implementing HPF language extensions and tool features to also provide parallelization, debugging, performance monitoring, and analysis support for scientific applications considered today difficult to parallelize on DMPPs (e.g., unstructured sparse matrix computations).

Some of the above goals are also met by tool environments offered by some DMPP vendors and other research groups. Meiko and Intel, for instance provide instrumented versions of the proprietary message passing libraries, CSN and NX respectively. Instrumented programs drive modified versions of the ParaGraph [HF93] trace visualization tool. Both vendors also provide parallel debuggers with their machines — Intel `ipd`, and Meiko the `pdb` debugger. Thinking Machines offer an integrated environment called Prism [ABJ+91] for program debugging and visualization on their CM series of DMPPs in their proprietary Fortran and C language variants. Distributed data can be visualized during program execution, or performance data analyzed after execution completes. On their CM-5 series, individual nodes can open separate debugger windows. Applied Parallel Research market an interactive Fortran program browser and

analyzer, called Forge [APR93], which can also assist parallelizing programs for a number of distributed- and shared-memory platforms. This tool has been recently enhanced to support HPF, and can assist conversion of Fortran programs to (and from) HPF. We can only summarize functionality for current DMPP environments which we consider most important. Many other such environments exist and for a more complete overview of existing tools, the reader is referred for instance to [Che93].

However we believe that our main goals, i.e., portable programming using both standardized message passing and data-parallel programming interfaces, support of irregular computations, and most important application-driven tool design, together are not targeted by vendors and other research groups.

2.2 Overview

There are three component tools within our tool environment, sharing a common User Interface (UI): a Parallelization Support Tool (PST), a Performance Monitor and Analyzer (PMA), and a Parallel Debugging Tool (PDT).

The integrated environment accepts high-level extended HPF programs and low-level message passing code. PST acts mainly as a compiler for both paradigms. PMA and PDT are designed with the same philosophy, i.e., it will be possible for the user to obtain information at different levels of abstraction. The lowest level of abstraction, providing the most detailed information, is as close as possible to the DMPP's hardware. Since we are also interested in porting the environment to several DMPPs, this lowest level of abstraction is the common communication platform (MPI) installed on all machines considered. For instance, a detailed break-down of parallelization overhead in communication, computation, and idle times on all processors will be provided at any time of program execution. A higher level of abstraction is provided by considering features of the high-level language, such as global name space and data distribution, or data parallel execution mode which appears to the user as a single program thread. The inter-relationship of the components within the tool environment is shown in Fig. 1.

PST aims to provide extensive run-time support for scientific applications today considered difficult to parallelize on massively parallel distributed systems. Working as a preprocessor on source programs, the application sources will be instrumented to generate both performance information for PMA and debugging information for PDT.

PMA utilizes trace information generated during the execution of a parallel program, and assists the performance tuning and interpretation of program execution through visualization and analysis of this information. Different levels of abstraction are supported, from the analysis of communications and memory utilization of individual processes through to global views of the data-parallel execution.

PDT similarly supports debugging at different levels of abstraction, often with support from PST and PMA. Debugging of single node programs at the source level through to high-level support of the parallel distributed program (mostly) presented as a single-thread is planned.

The user interface provides a single interface between the component tools and the user. It primarily consists of a source code browser, which can also be directed (and possibly annotated) by PMA and PDT to show features or source regions of interest. Output from a running program, sent to both standard output and error streams by processor nodes, is also displayed under the control of the UI in a separate window.

The UI also directs the operation of the other tools and controls parallel program construction and execution. This latter role is performed in conjunction with the *Tool Services Agent*

2.5 The Performance Monitor and Analyzer (PMA)

PMA provides facilities to assist with the instrumentation and analysis of program execution. Initially, PMA is used to configure parallel program instrumentation. Subsequently it interprets trace information generated during execution, assisting with performance tuning and interpreting program execution through visualization and analysis of this information. Different levels of abstraction are supported, from the analysis of communications and memory utilization of individual processes through to global views of the "data-parallel" execution.

In the first prototype, PMA drives a collection of ParaGraph displays after the execution of a parallel program has completed and execution trace information from the instrumented communications library has been collated. Later prototypes include additional performance statistics generated (as a first step) by post-processing the trace files. Information provided by PST, concerning memory consumption and dynamic data distributions, is also included as additional statistics. Customized displays are being designed, to provide increased insight into the program execution. These displays can be interactively rescaled, to allow the resolution of finer details, or to provide a more general impression of program behavior. One such display can show each thread of the parallel program represented by its own utilization time-line, with communication highlighted between threads. Other metrics can also be selectively combined in the display, sharing the same annotated time-axis to facilitate the correlation of associated properties.

PMA sets instrumentation "checkpoints," i.e., source lines defining regions where communication tracing is switched, possibly specified in conjunction with the source browser of the UI. These instrumentation regions, with an appropriate instrumentation level (from a simple execution path to a detailed performance profile), are configured by PMA in liaison with the TSA. It is the TSA which modifies the program instrumentation state and maintains the instrumentation of the parallel program during execution.

It should be noted that our final target is *interactive* (rather than post-mortem) profiling of parallel programs. To minimize PMA's impact on program execution, and reduce the amount of trace information which might otherwise burden the system I/O and disk capacity, efforts will be made to process the communication traces in parallel on the distributed memory platform. This will also remove some sequential bottlenecks in trace data transfer and processing, providing scalable support for larger systems.

2.6 The Parallel Debugging Tool (PDT)

PDT is as a source-level, interactive parallel debugger. The first prototype consisted of a wrapper to TSA. It offered the same functionality and commands as a standard sequential source-level debugger, plus a few extensions required for debugging parallel programs on DMPPs. Commands are provided to attach to and detach from the target platform, to open a partition, to load and run a parallel program, to deal with global break-points and exceptions, and to switch from one processor to another when the program is stopped for stack, registers and data examination. This functionality is similar to that supported by Meiko's **pdb** for the CS-1 and Thinking Machines' Prism debugger for the CM-5.

Later prototypes address more specifically each programming level supported by the tool environment. At the high-level we plan to fully support source-level debugging of PST programs presented as single-threaded programs, and we will facilitate interpretation of large distributed arrays by providing graphical views of the data layout (such as views supported by Prism and Forge) and how it is redistributed as the program executes. At the message-passing level, we are

adding facilities for run-time deadlock detection, conditional break-pointing, and race condition detection.

2.7 The User Interface (UI)

The UI currently supports browsing through program source files, selecting break-points and checkpoints, possibly invoking different compilers (i.e., C, Fortran 77, and PST, with automatic selection of appropriate compilation flags and libraries). It allows PDT and PMA to be invoked and managed from the one unifying interface.

Future enhancements to UI will provide feedback between PMA and UI to allow, for instance, direct display of the most important global performance statistics in the source browser beside related lines of source code, or annotations of an interactive program call-graph structure display.

3 Parallelization of Applications and Algorithms

3.1 Overview

Application developers in the project are currently concerned with parallelizing a wide range of applications on a number of platforms. In order to better evaluate the environment provided by the tool developers and the underlying machine architectures, we have set up an application suite, which consists of a wide range of existing sequential research codes as potential parallelization candidates.

The research emphasis of the group lies in parallelizing unstructured problems, typically those which are arise from the solution of partial differential equations solved with irregular grids. This is reflected in our choice of codes discussed in this section. Similar research on this topic is taking place at other institutions, see [SG93, Sim91, SS86, Saa90], et al.

As a first step, we have parallelized several programs in order to gain experience in both low-level message passing techniques and data parallel programming on the available platforms. Strong emphasis has been placed on producing portable message passing and data parallel code on several architectures, as will be described.

The long term objectives are the following:

- Parallelization of further algorithms using PST and MPI, as described in our application suite document [JMSd93].
- The design and implementation of a library of parallelized routines, solvers, etc, in order to have plug-in building blocks for applications.
- Development of software for dynamic mesh partitioning. This is strongly linked to the introduction of directives in PST by the tool developers to supplement those in HPF.

3.2 Application Suite

The application suite consists of roughly 20 different existing applications to be ported to parallel platforms using the tool environment. These include several finite element packages, an electronic structure code, a ray-tracing application, and several molecular dynamics codes.

The application suite document specifies further the approach taken to parallelizing applications, which we summarize in six points:

1. The application is ported in a *bottom-up* approach by rewriting it in HPF/PST. The underlying algorithms are not changed.

2. The parallelized code is debugged with PDT.
3. The performance of the code is analyzed with PMA.
4. Commonly used routines are marked to be optimized later in a library, possibly using low-level MPI routines.
5. The tool environment is evaluated with respect to the parallelization effort required, the program performance and the software portability.
6. Weak points of the underlying architecture are identified.

We elaborate this process here in some detail. The parallelization depends mainly on the structure of the program and the data structures used. General issues are clear: to match data and work distribution in order to preserve (or enforce) locality as much as possible. How to do this is still up to the creativity and skill of the programmer. The approach to the parallelization will be determined by the most time consuming parts of the code. One should keep in mind that these may differ from sequential to parallel computers.

The code is first analyzed with respect to the most time consuming parts of the algorithm, the interaction between algorithms and data structures, and distribution or replication of data structures. Algorithms are often determined by the data structures used — this may require changes for a parallel implementation. Apart from the efficiency of the algorithms using these data structures, consideration should also be given to memory usage and scalability. Replicating large data objects can result in very efficient code, but it limits the problem size to that which can be stored on a single processor (irrespective of the number of processors used), and therefore restricts scalability.

After this analysis, the parallelization tool (PST) is used for an initial parallelization of the code. The resulting parallel code may be investigated and debugged with the parallel debugger (PDT), and the performance monitor (PMA) used to find inefficiencies: communication bottlenecks, load imbalances, or excessive overhead introduced by the compiler (e.g., in computing a communication pattern). This information can be used to make modifications to the data and work distribution, and to help the compiler with further optimizations by adding directives to specify independencies and invariants in the code.

3.3 Feedback to Tool Developers and System Designers

Along with regular releases by the tool developers, there is timely feedback from the application developers. Feedback not only consists of bug reports and requests for optimizations, but especially identifies desired tool functionality enhancements. This has already resulted in performance optimizations in PST, higher usability in PDT, and the provision of additional information by PMA.

The inclusion of new functionality is necessary for our research in unstructured problems. Currently support for such problems is provided mainly by external libraries, e.g., [BSS91]. We feel that it is imperative for the parallelization tool environment itself to provide functionality to support irregularly distributed code and data structures, to debug parallel code efficiently, and to visualize sparse matrices and their distribution. Response from the tool developers has been positive and such functionality is planned for incorporation into future versions of the tool environment.

The feedback to the system designers is the desire for low message latencies and high communication bandwidths, hardware support for global communications primitives such as broadcasts, and a high resolution timer to properly monitor program execution. Proper treatment of

these issues is necessary for the development and efficient execution of a wide range of applications.

3.4 Overview of Applications Tested

From a range of benchmark codes available to us, for this paper we chose to concentrate on two commonly addressed benchmarks from a standard suite, and a third commercial benchmark application (in two problem sizes). This was primarily done in order to investigate and demonstrate some of the interesting aspects of PST, namely user-defined mappings and multiple communication patterns which can be saved and reused independently.

PST accepts a user-defined function for the mapping of global array indices to local array indices and the processor number on which the data resides. This feature is very important in unstructured problems for which the standard HPF data distribution directives are insufficient, and existing codes which utilize packed data representations. In addition, routines working on distributed data require a different communication pattern instance at every grid level. Saving these patterns, as discussed in the PST description, can give huge performance savings when the entire procedure is repeated many times.

NAS Kernels

The Numerical Aerodynamics Simulation kernels (hereafter NAS kernels) are widely accepted and implemented benchmarks for a variety of computers — not only the current generation supercomputers, but also the shared and distributed memory multiprocessing supercomputers. Therefore these benchmarks are suitable as one means to evaluate our effort in the ongoing project.

The NAS benchmark kernels consist of eight codes, known as Embarrassingly Parallel (EP), Multigrid (MG), Conjugate Gradient (CG), 3DFFT PDE (FT), Integer Sorting (IS), LU-solver (LU), Pentadiagonal Solver (SP), and Block Tridiagonal Solver (BT) [BBDS92].

Here, we consider two codes among them:

MG: calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on a $n \times n \times n$ grid with periodic boundary conditions. The NAS MG benchmark deals with a maximum grid size of $n = 256$, though for our benchmarks we used $n = 64$, which was more appropriate to the size of the machines which we had available at the time.

CG: CG finds an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros using the inverse power method. The sparse matrix used in our benchmarks has 14,000 columns and 1,853,104 non-zeros ("Class A").

A Finite Element Code for Unstructured Problems

This application, made available to us by Electricité de France (EDF), solves the two-dimensional heat equation with mixed Dirichlet and von Neumann boundary conditions on a rectangular plate using a finite element method (see [Esc93]). The resulting sparse linear system is solved by the conjugate gradient (CG) method [HS52, GL89], and requires indirect addressing because of the sparse matrix storage. Both a stationary and a time-dependent problem are solved, and the average performance is shown. We consider a small problem (FE1) with approximately 7,000

unknowns and a larger problem (FE2) with approximately 110,000 unknowns. The problem differs from the NAS CG kernel in that the structure results from a finite element grid.

3.5 Benchmark Parallelization

Multigrid NAS Kernel (MG)

The V-cycle multigrid algorithm is applied to a $n \times n \times n$ grid in order to approximate the discrete Poisson problem $\Delta u = v$ over the unit cube with periodic boundary conditions, and an irregular grid-point initialization. The description of the algorithm can be found in [Hac85].

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. The most efficient blocks have been found empirically to be "matchsticks": all grid points in one dimension are on one processor and distributed blockwise in the other two dimensions.

A first version of MG was parallelized using explicit MPI routines (in a complementary project [Küh93]) and where the code was rewritten in a highly optimized fashion.

The second approach used PST to parallelize the original Fortran code supplied by NAS. 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 the PST code, each grid level in this long array is distributed regularly in the matchstick fashion. A user-defined mapping was used to map each global index to the index of the local data element in the appropriate grid. 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 table lookups.

In addition, loop distribution directives were applied to all of the loops over the entire cube on each grid level. These loops do not need to be synchronized due to the absence of data dependencies — this fact can be specified in PST for further efficiency. The streamlining of the code in this fashion is illustrated in the trace of Fig. 5, where the lack of synchronization between processors during the last three V-cycles is shown by the fact that each can immediately continue to subsequent sections (i.e., the horizontal bars don't line up vertically).

Conjugate Gradient NAS Kernel (CG)

The conjugate gradients method is well described in the literature [HS52, GL89]. Our approach to parallelization of the NAS CG kernel was the same in both the MPI and PST versions: starting with the example Fortran source code, the sparse matrix was generated by sorting elements by their column index. Matrix columns are then distributed over all the processors, either in a cyclic fashion (round-robin) or blockwise.

The principal difference between the MPI and PST versions is in the referencing of the data structure: the former has local indexing written into the code, while the latter accesses global indices, as in the original, but has a user-defined global-to-local mapping to determine local indices.

In the most efficient implementation all processors keep a full copy of all n-vectors (five of which are necessary for the conjugate gradient iteration) local to all processors. The only communication is in the matrix-vector multiplication: scalar products of matrix rows and the incoming vector are constructed; portions of the resulting vector are gathered on all processors, so that each has its own copy.

The MPI version takes advantage of the `MPI_Allgather()` global communication routine to efficiently redistribute the resultant vector. The PST split-by-column version has a very simple user-defined mapping which requires almost no additional memory. Function calls for the global-to-local mapping cannot be used, because the matrix data structure is referenced frequently.

Since the CG "Class A" problem is also fairly dense, the cost of having local n-vectors (i.e., sequentially executed BLAS1 routines [LHKK79]) is not appreciable when compared to the matrix-vector multiplication. The final MPI and PST versions are both surprisingly simple, since all n-vectors are local and the only communication occurs in the matrix-vector multiplication. Unfortunately, the lack of data locality requires this communication to be global.

Finite Element Application

We developed two parallel versions of the code. One was obtained using the PST directives, the other has the communication hand-coded using MPI routines. The most time-consuming part is the matrix-vector multiplication in CG which uses indirect addressing, because the sparse matrix is stored in compressed form (ITPACK format [KRYG82]).

For the MPI version, we used the so-called replicated data approach. That means that all processors hold all data. The computation itself is decomposed in the sense that each processor updates only a certain subdomain. In our case, the replicated data approach has the advantage that the program can be parallelized with minimal alterations to the code. A drawback of the approach is obvious: we are limited to problems that fit in the local memory of one processor. The time-dependent case FE2 could not be solved on the Paragon for this reason.

Each processor is responsible for the update of vector elements corresponding to a block of rows of the matrix. There are two routines which require communication. The first is the calculation of dot-products, which requires a global sum, done using an MPI reduction routine. The second is the matrix-vector multiplication which requires indirect addressing: a processor might access an element which has not been locally updated. To implement an efficient communication scheme, we take advantage of the sparsity of the matrix. Before the iteration we determine the pattern of the necessary communication, which remains the same during the iterations, and then we perform only the necessary point-to-point communication within each iteration.

In the parallelization with PST a domain decomposition approach is used. The unknowns are distributed over the processors, and the rows of the matrix are distributed correspondingly. This leads to an implementation with (in principle) a minimal overhead in memory usage and computation.

Efficient memory utilization was important because PST can have considerable memory overhead (beyond that of an explicit parallelization). It turns out, however, that this overhead was not too large; it ranges from 40% on 4 processors to 70% on 64 processors. Communication is only required for the inner products and the matrix-vector multiplication. The communication in the inner products does not reduce the performance significantly on a small number of processors. The matrix-vector multiplication only needs communication between neighboring sub-domains/processors. Therefore, communication costs can be kept relatively low. Most of the overhead is nevertheless generated in the matrix-vector multiplication in the form of integer arithmetic in address computations. Using PST this overhead can be considerably reduced by indicating with an annotation that a given assignment only operates on local data.

4 Results

Two distinct platforms were chosen to evaluate our work: to demonstrate the portability, and to compare the performance and effectiveness, of parallelization in a high-level language (using PST as opposed to manual-coding using explicit function calls). There was no intention to compare the different hardware architectures themselves, nor were any specific application optimizations undertaken for either machine — the same code was executed on both.

4.1 Basic Platforms

The NEC Cenju-2

The current Cenju-2 configuration used for the performance measurements described in this section consists of 16 computational nodes, featuring a MIPS R3000 chip-set and 64 Mbytes of memory. Each CPU runs at 25 MHz and has two 64 Kbyte caches for data and instructions, respectively. The nodes are connected with a multi-stage shuffle-exchange network built of 4 x 4 switching units. A processor interfaces to the network with two Direct Memory Access Controllers (DMACs). Although each switching unit has a bandwidth of 32 Mbytes/s for each of the four channels, the two DMACs provide a bi-directional hardware bandwidth of 12.5 Mbytes/s. Our Cenju-2 is operating in multi-user space-shared mode.

The Fortran compiler we used was the native compiler on the Cenju-2 host, an NEC Engineering Workstation (EWS4800 series), running OS version Release 6.1 Revision 07. As C compiler and back-end to PST the GNU C compiler `gcc` version 2.5.7 was used.

Intel Paragon

The Intel Paragon XP/S 5+ used features 96 Intel i860 XP processors. The processing nodes are connected in a rectangular mesh pattern, unlike the hypercube inter-connection pattern used in the earlier Intel iPSC/860. The node processor chips operate at 50 MHz, contain 16 Kbyte data and instruction caches, and can issue a multiply and add instruction in one cycle. The maximum bandwidth from cache to floating point unit is 800 Mbytes/s. Hardware communication bandwidth between any two nodes is 200 Mbytes/s full duplex. Each node of the machine used also had 32 Mbytes of memory.

As C compiler and back-end to PST, the PGI compiler version 4.1.1 is used under operation system release OSF AD version 1.1.4.

Message Passing Interface

On the Cenju-2, an efficient MPI subset has been implemented in three layers. The lowest layer consists of three hardware-dependent basic functions, i.e., non-tagged send, receive and poll operations which are partially supported by the operating system. The second layer consists of 12 operations for tagged point-to-point communication of contiguous data streams, i.e., blocking receives combined with non-blocking sends, blocking receives combined with blocking sends, and higher level (tagged) poll operations. On top of these, the third layer consists of 14 global communication operations implementing barrier, one-to-all and all-to-all broadcast, gather and scatter, all-to-one and all-to-all reduction, and multicast operations.

On the Intel Paragon the whole MPI subset was implemented on top of the native NX communication library. Note that most of our MPI functions had an NX equivalent, and could therefore be implemented with pre-processor macros. Note also that the achieved performance

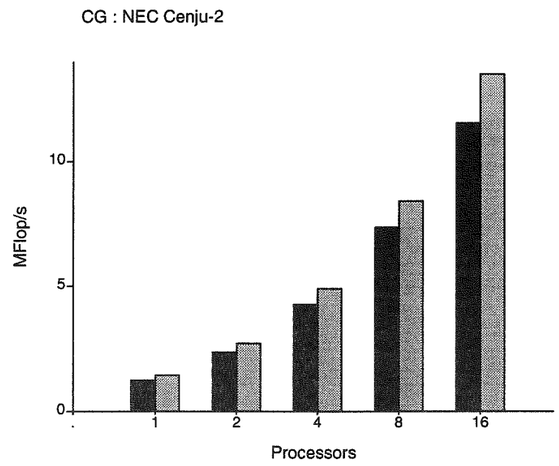


Figure 2: *The MPI CG version compiled with the Cenju's native Fortran compiler (light), and also with PST (dark), i.e., without using the global name space supported by PST, but explicit message passing calls.*

from this approach is quite likely to be inferior to a more direct use of the machine and vendor-provided functions.

4.2 Comparative Evaluation of Different Benchmark Versions

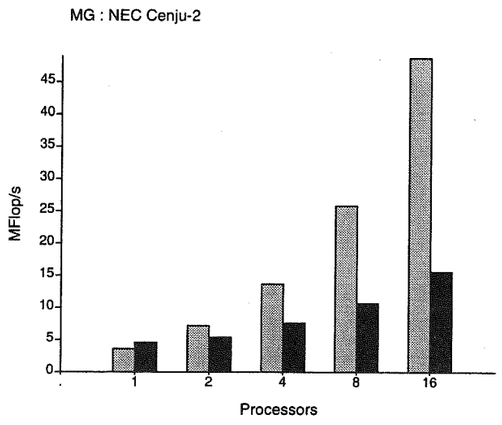
In the parallelization of our three benchmarks we are concerned with several issues. The principal interest is a comparison of the codes parallelized with the high-level data-parallel programming tool PST and low-level message-passing techniques (MPI). The criteria for this comparison are the ease and comfort of parallelization, the scalability of the two versions, especially for large problems, and, finally, absolute performance. We feel that the last is a significant, but not the only, consideration in such an evaluation, particularly when the development time for parallelization is considered.

The MPI versions of CG, FE1, and FE2 are written in Fortran 77 extended with calls to the MPI library. MG is written in C with MPI. The PST versions are all written in Fortran 77 with parallelization directives: PST translates this annotated Fortran code into C with MPI communication routine calls. PST can also be used to translate the MPI Fortran codes into MPI C codes, just as it can compile sequential Fortran programs.

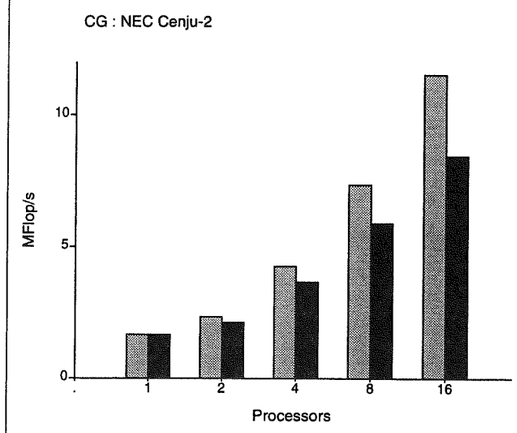
In order to put the MPI–PST comparison on an equal basis, we compare the MPI version translated by the PST front-end to the C compiler with the pure PST version. The quality of the PST generated C code is illustrated well in Fig. 2, where the performance of the MPI version of the CG benchmark compiled on the Cenju-2 with PST is compared to the same benchmark compiled with the Cenju's native Fortran compiler. Compilation with PST yields 10–20% slower code for the range of machine sizes considered.

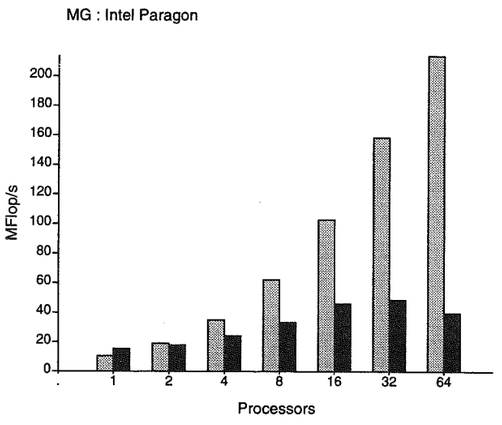
4.3 Message Passing vs. PST Global Name Space

(a) NAS Multigrid



(b) NAS Conjugate Gradient





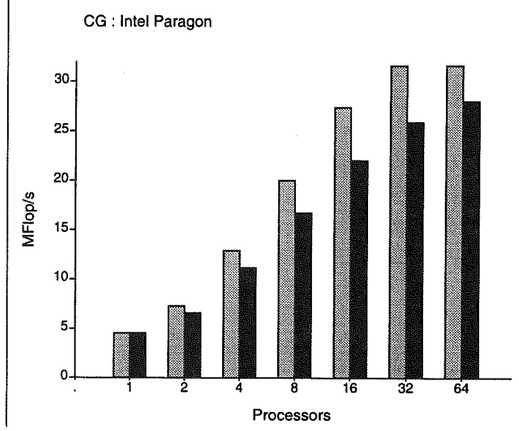
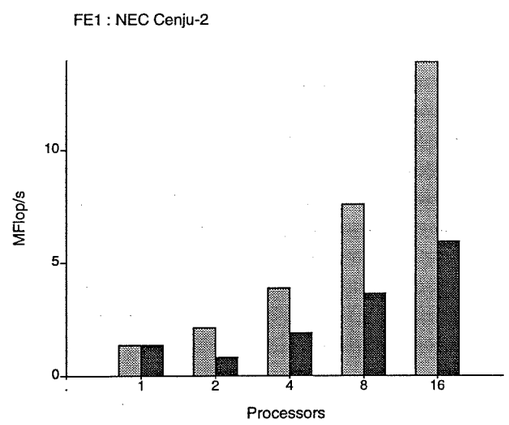
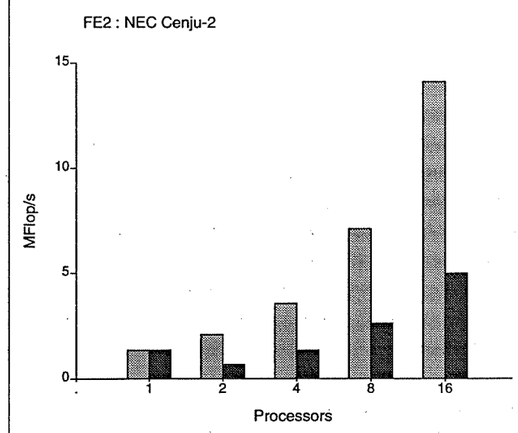


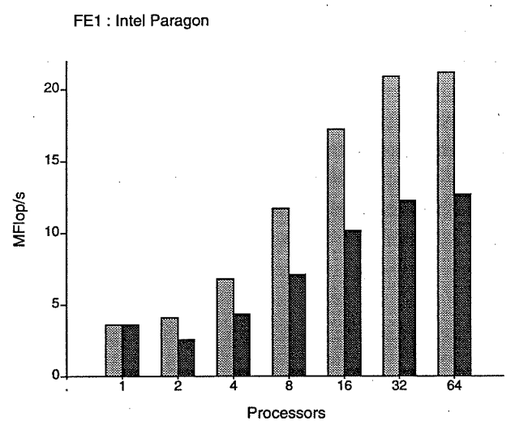
Figure 3: *MG and CG were parallelized with MPI calls, and also using PST's global name space support. Performance of both versions is measured on both the Cenju-2 and on the Paragon with several machine sizes. MPI performance is depicted in light grey, and PST global name space performance in dark grey. [Note the different scaling of the axes.]*

(a) EDF Benchmark FE1



(b) EDF Benchmark FE2





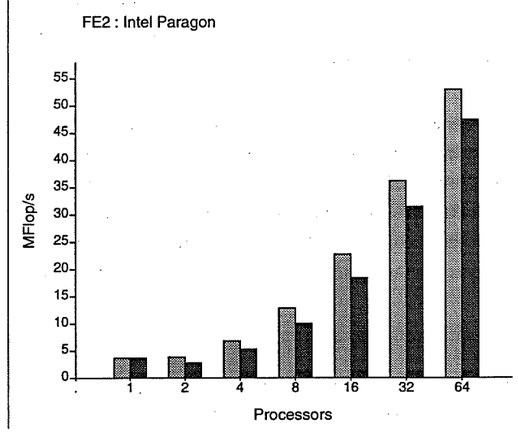


Figure 4: *The FE benchmark was parallelized with MPI calls, and also using PST's global name space support. Two different sizes of problem were considered, referred to as FE1 and FE2. Performance of both versions is measured on both the Cenju-2 and on the Paragon and several machine sizes. MPI performance is depicted in light grey, and PST global name space performance in dark grey. The FE2 MPI Paragon graph does not include the results for the time-dependent case. [Note the different scaling of the axes.]*

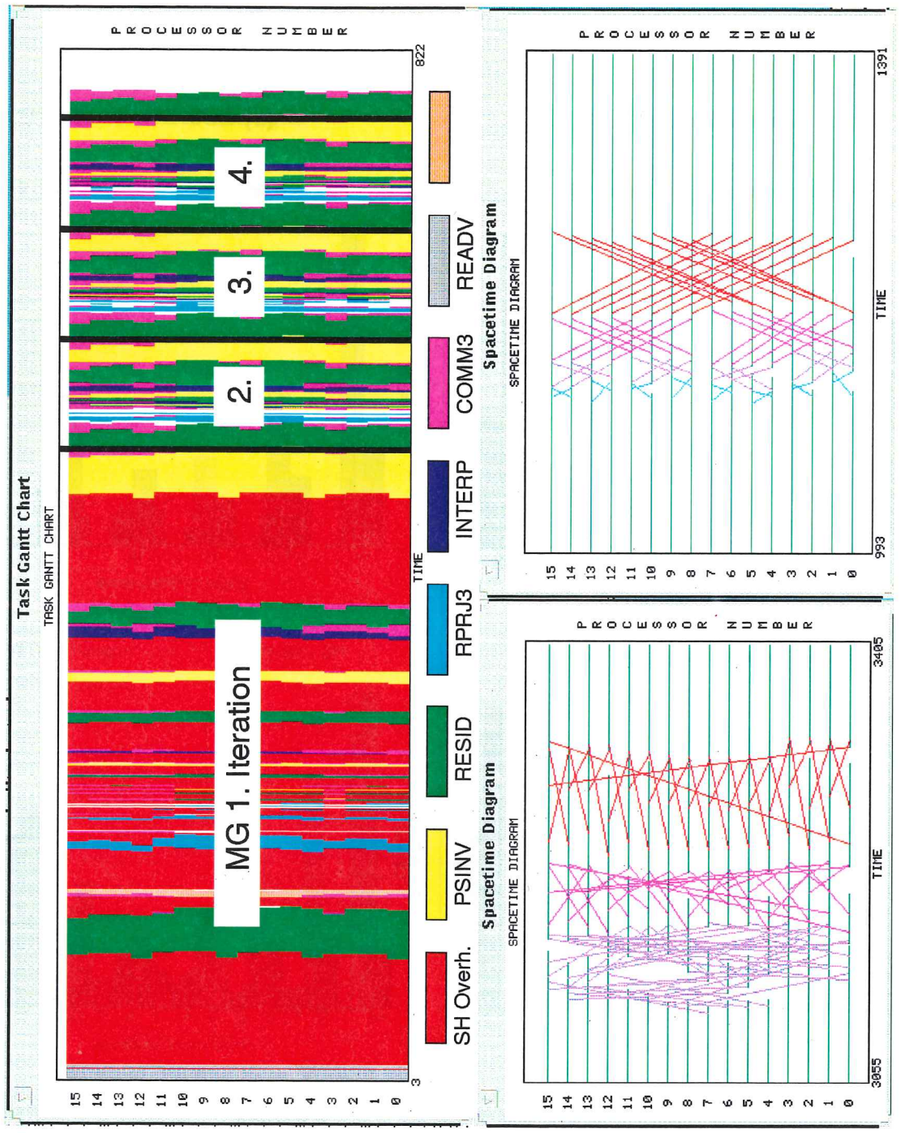


Figure 5: *The upper part of the figure shows a task Gantt chart of the first four V-cycles of MG. Each of the 6 main routines is depicted in a different color. Symbol-handler overhead is depicted in red. It can be seen that this overhead only occurs during the first V-cycle, although the main routines are called with different parameters throughout the V-cycles. The lower two figures show a communication space-time diagram of a matrix-vector multiplication during CG: on the left, the communication pattern generated by PST is shown, while on the right, the pattern underlying the MPI routine* `MPI_Allgather` *is shown.*

MG (Multigrid)

The current PST version supports a global name space only through the previously described run-time mechanism. In contrast to the other benchmark programs, for MG the global name space could be supported through compile-time analysis only. PST will be enhanced with such static analysis as soon as it is incorporated into the NEC HPF system.

As shown in Fig. 3(a), the performance achieved with the data-parallel program compiled by PST is much less than that of the MPI benchmark version, but as already mentioned, we do not consider it indicative for performance achieved with MG compiled by future PST versions. However, the benchmark uses multiple communication patterns for all its critical code segments: the main subroutines are called with exponentially decreasing or increasing problem size as the program steps through one V-cycle. Since a larger 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. This is shown in Fig. 5, where a task Gantt chart of the first four V-cycles of a program run is depicted. Performance measurements of Fig. 3(a) refer to the steady state, i.e., the overhead of the symbol-handler is not included in the measurements.

Note that the problem size considered ($n = 64$) is small. For larger problems, however, we expect the code to run more efficiently, since the block surface (communication) to volume (computation) ratio decreases as $1/n$.

On the basis of analysis such as the one depicted in Fig. 5, requests for optimizations could be quickly incorporated into the tool environment. The performance thus improved to the point where it is comparable to the highly optimized MPI code.

CG (Conjugate Gradient)

As already mentioned in section 3.5, the MPI version takes advantage of the `MPI_Allgather()` routine to efficiently redistribute the resultant vector. As can be seen from the trace displays in Fig. 5, the communication pattern generated by PST to gather the resultant vector on all processors is almost as efficient as the butterfly pattern underlying our implementation of the `MPI_Allgather()` global communication routine.

The results in Fig. 3(b) for the Cenju-2 and Paragon show strong similarity over the range of problems on 1 to 64 processors, and indicate that PST has managed to create near optimal communication patterns for this problem. The performance curve on the Paragon appears to flatten off faster than that of the Cenju-2: this can be explained through differences in the communication/computation ratio [APR89, AR92].

FE1 and FE2 (Finite Element)

Fig. 4 shows the performance measured for the finite-element application. We see that on the Cenju-2 the performance of the MPI versions are about a factor of two better than that of the PST versions. On the Paragon the differences are slightly smaller. Note that the PST benchmark version has much smaller memory requirements than the MPI version. Depending on the problem size this can lead to problems on DMPPs featuring small local memories. A more storage-efficient MPI version could be written, but this would lead to a much more complicated program. Moreover, this program would be more expensive in address computations as well. The PST versions have a large initial parallel overhead in address computations, however, this

overhead involves only local computation and therefore becomes increasingly less significant for larger numbers of processors. Very good relative speedup figures are observed, especially the run-times for FE2 on the Cenju-2 which indicate almost linear speed-up.

5 Conclusions

The Joint CSCS-ETH/NEC Collaboration in Parallel Processing has produced first prototypes of tools within an integrated environment, which have successfully been used in the effective parallelization of a number of applications on various DMPPs. As the project continues, the tools are becoming more powerful and refined, and more useful to the application developers, based partly on suggestions which arise from their experiences with the tools and new applications. The hardware and system software has also benefitted from the experiences of those who made first extensive use of them.

The major characteristics of the project are:

- Joint development of tools and applications to provide mutual interaction between the two parts of the project. The requirements of the tool environment are determined by abstraction from applications of scientific interest, an approach which ensures that the tools really give support for problems which are important in practice. On the other hand, once an application requirement has been built into a tool environment, it can likely be re-used to efficiently parallelize other similarly structured applications, and the general programmability of the underlying DMPP is significantly enhanced.

- Use of standardized programming languages and machine interfaces for portability. At the high level HPF and at the low level MPI are used.

- Parallelization of applications known today to be difficult to parallelize on DMPPs. In our opinion, part of these difficulties stem from missing or inappropriate tool and high-level language features. In our project, both application developers and tool designers interact to define and include support for such new features in all tool environment components.

To demonstrate the importance of these characteristics of our project, we have shown preliminary performance measurements of three application programs on two DMPP systems, a NEC Cenju-2 and an Intel Paragon. The programs were parallelized both using low-level message-passing primitives and high-level data parallel language constructs. All but one of the programs are based on irregular sparse matrix computations. Compiler-generated parallel programs were not expected to perform as well as the hand-parallelized and optimized benchmark versions, but this is more than compensated by the additional comfort of a high-level programming interface as provided by our integrated tool environment.

Acknowledgments As part of the 1993 CSCS Summer Student Internship Program, an initial port of `gdb` to the Cenju-2 EWS front-end was done by J. Blandy, functions for generating PICL-format traces were incorporated in our MPI implementation by M. T. Nyeu, and the MPI NAS multigrid kernel was parallelized by U. Kühn. We would also like to thank Electricité de France, and R. Gruber and M. Tomassini from CSCS, for their consultation as part of a wider benchmark project.

References

[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, November 1991.

[APR89] M. Annaratone, C. Pommerell, and R. Rühl. Interprocessor communication speed and performance in distributed-memory parallel processors. In *Proc. 16th Symp. on Computer Architecture*, pages 315–324, Jerusalem, Israel, June 1989. IEEE-ACM.

[APR93] APR (Applied Parallel Research). HPF parallelization tools for clustered workstations and distributed memory multi-processor systems. HPC Select News, article 914, August 1993. Further information available from APR, Placerville, CA.

[AR92] M. Annaratone and R. Rühl. Balancing interprocessor communication and computation on torus-connected multicomputers running compiler-parallelized code. In *Proc. SHPCC 92*, Williamsburg VA, March 1992. IEEE.

[BBDS92] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmarks Results. Technical Report RNR-93-016, NASA Ames Research Center, CA, Dec 1992. With supplementary NAS parallel benchmark update tables of Feb. 1994.

[BSS91] H. Berryman, J. Saltz, and J. Schroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3), June 1991.

[CEF+94] C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie. An environment for portable distributed memory parallel programming. *Proc. IFIP WG 10.3 Working Conf. on Programming Environments for Massively Parallel Distributed Systems (Ascona, Switzerland, April 1994)*, 1994. To be published by Birkhäuser (Basel).

[Che93] D. Y. Cheng. A Survey of Parallel Programming Languages and Tools. Technical Report RND-93-005, NASA Ames Research Center, Moffet Field, CA, March 1993.

[Dec93] K. M. Decker. Methods and Tools for Programming Massively Parallel Distributed Systems. *SPEEDUP Journal*, 7(2), November 1993.

[Esc93] P. Esclangon. Support theorique d'un benchmark supercalculateur. Technical Report 93NJ00009, Service Informatique et Mathématiques Appliquée, Electricité de France, April 1993. ISSN 1161-059X.

[GL89] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1989.

[Hac85] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer Verlag, 1985.

[HF93] M. T. Heath and J. A. E. Finger. ParaGraph: A Tool for Visualizing Performance of Parallel Programs. Technical report, UIUC/ORNL, 1993.

[HKT91] S. Hirandani, K. Kennedy, and C. W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing 91*, Albuquerque, NM, November 1991. ACM-IEEE.

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

[HS52] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[JMSd93] G. Jost, N. Masuda, W. Sawyer, and E. de Sturler. Application Suite Document. Technical Note SeRD-CSCS-TN-93-15, CSCS, December 1993.

[KMv90] C. Koelbel, P. Mehrotra, and J. van Rosendale. Supporting shared memory data structures on distributed memory architectures. In *2nd Symp. on Principles and Practice of Parallel Programming*, page 177, Seattle, WA, March 1990. ACM SIGPLAN.

[KRYG82] D. R. Kincaid, J. R. Respess, D. M. Yound, and R. G. Grimes. ITPACK 2C: A Fortran package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Trans. Math. Soft.*, 8:302–322, 1982.

[Küh93] U. Kühn. Analysis of Selected NAS Benchmark Kernels. Technical note, CSCS, October 1993.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[MPI93] MPIF (Message Passing Interface Forum). Document for a Standard Message-Passing Interface. Technical report, Oak Ridge National Laboratory, TN, 1993.

[PR94] C. Pommerell and R. Rühl. Migration of Vectorized Iterative Solvers to Distributed Memory Architectures. In *Colorado Conference on Iterative Methods (Breckenridge, Colorado, April 1994)*, 1994. Preliminary proceedings.

[Rüh92a] R. Rühl. Evaluation of compiler generated parallel programs on three multicomputers. In *Proc. ACM Int. Conf. on Supercomputing*, Washington, July 1992.

[Rüh92b] 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. CSRD Technical Report 1029, University of Illinois, IL, August 1990.

[SG93] J. R. Shewchuk and O. Ghattas. A compiler for parallel finite element methods with domain-decomposed unstructured meshes. In *Proc. 7th Int. Conf. on Domain Decomposition Methods*, Pennsylvania State University, PA, October 1993.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[SS86] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 10:856–869, 1986.

[ZBC+92] 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.