Architecture and Programmability of the NEC Cenju-3

Christian Clémenson\textsuperscript{a}, Karsten M. Decker\textsuperscript{a}, Akiyoshi Endo\textsuperscript{b}, Josef Fritscher\textsuperscript{b}, T. Maruyama\textsuperscript{c}, Norio Masuda\textsuperscript{b}, Andreas Müller\textsuperscript{a}, Roland Rühl\textsuperscript{a}, William Sawyer\textsuperscript{a}, Eric de Sturier\textsuperscript{a}, Brian J. N. Wylie\textsuperscript{a} and Frank Zimmermann\textsuperscript{b}

\textsuperscript{a}Section of Research and Development (SeRD)
Swiss Scientific Computing Center (CSCS-ETH)
Manno, Switzerland

\textsuperscript{b}NEC SX-Center
Via Cantonale
Manno, Switzerland

\textsuperscript{c}C&C System Laboratories
NEC Corporation
Tokyo, Japan

The Joint CSCS-ETH/NEC Collaboration in Parallel Processing is one of the major research projects currently conducted at CSCS. This project comprises the development of the portable integrated tool environment Annai together with applications and algorithms for distributed-memory parallel processors. As part of the collaboration, NEC agreed to provide CSCS with a parallel computer to be used as the main development platform. The 16 node NEC Cenju-2 supplied initially was recently replaced with a far more powerful machine, namely a 128 node NEC Cenju-3. This article describes the architecture of this new machine and gives an introduction to Annai. In addition, first Cenju-3 benchmark results are presented.

1. Introduction

Since April 1993, ten researchers have been assembled to work within the Joint CSCS-ETH/NEC Collaboration in Parallel Processing [4] at the Swiss Scientific Computing Center (Centro Svizzero di Calcolo Scientifico, CSCS), part of ETH Zürich. Six researchers participate in the development of an integrated tool environment for distributed-memory parallel processors (DMPPs), and four are working on applications and the development of new parallel algorithms. The overall goal of the project is to make DMPPs as general purpose as the platforms mostly used for high-performance scientific computing today, namely shared-memory vector computers.

The integrated tool environment, called Annai, supports both low-level and high-level language parallelization, parallel debugging and performance analysis [5]. Annai is a prototype system currently not available on other Cenju-3 systems. It is developed as a sequence of prototypes, which are used and evaluated by the application developers. This prototype evaluation allows rapid feedback and requests of the application developers for functionality enhancements are immediately considered for inclusion in future tool prototypes. The application and algorithm developers do not only target applications known to parallelize easily. To have a fair representation of the typical workload of today’s high-performance supercomputers, an application suite was compiled [9] which includes many applications known to be difficult to parallelize on DMPPs, such as unstructured sparse matrix computations.

Both tools and applications are developed in a portable way; specific tuning to a particular system is not among our goals. Therefore the tools use the standard Message Passing Interface (MPI) [11] as low-level machine interface. The applications can either be programmed using MPI, or at a higher level using our parallelization support tool, i.e., a compiler for High Performance Fortran (HPF) [8] with extensions defined by us. Our major development platform is provided by NEC as part of the collaboration. The initial platform, the 16 node Cenju-2 (with 1 Gbyte of memory total), was recently replaced with a 128 node Cenju-3 that features 8 Gbytes of memory.

2. Architecture of the NEC Cenju-3

NEC’s Cenju-3 is a state-of-the-art DMPP. It consists of up to 256 powerful processing elements (PEs), which communicate via a packet-switched multi-stage interconnection network (MIN). Several hosts can be attached to the Cenju-3 to serve as compilation engines and provide access to file systems and local networks as well as other devices. NEC uses high-performance 64-bit EWS4800 series workstations to avoid bottlenecks.

2.1 The Processing Elements

Figure 2 gives a more detailed view of the PE architecture. Each PE comprises a VR4400SC RISC processor, 1 MByte of secondary cache, 32 or 64 Mbytes of main
memory, and an intelligent interface to the interconnection network, the so-called interprocessor connection mechanism.

The VR4400SC CPU is a MIPS [10] compatible 64 bit processor. In addition to an instruction pipeline and an integer unit (IU), a floating-point unit (FPU), a memory management unit (MMU), and a high-speed primary cache of 32 Kbytes capacity also are mounted on-chip. To achieve single-cycle execution, the instruction pipeline is subdivided into eight stages. Its heartbeat is twice as fast as the external clock speed. Therefore, the peak performance of the 75 MHz version of the CPU used in larger Cenju-3 configurations, amounts to 150 MIPS and the FPU achieves a peak performance of 50 MFlops at this clock rate.

The redesigned interprocessor connection mechanism of the Cenju-3 is one of the major improvements over its predecessor. Whereas in the Cenju-2 a general-purpose DMA chip was used to transfer data between processors, the interprocessor connection mechanism of a Cenju-3 PE is made up of dedicated custom-design hardware components. They send and receive messages largely independently from the CPU at a transfer rate of 40 Mbytes/second; long messages are automatically divided into packets. Although the Cenju-3 is primarily meant as a message passing computer, the hardware is flexible enough to also emulate a single address space.

2.2 The Interconnection Network
The processing nodes of a Cenju-3 communicate via a multi-stage interconnection network (MIN). MINS are composed of (usually small) crossbar switches, which are arranged in several stages. The Cenju-3 features a packet-switched network and the switches on the way from a sender to a receiver form a pipeline. The network of the Cenju-3 is based on 4 × 4 crossbar switches working at a clock speed of 20 MHz. They achieve a throughput of 40 Mbytes/second since all data paths are 16 bits wide. The network of a 256 PE Cenju-3 consists of four stages, each of which comprises 64 switches. The interconnection scheme of the stages follows a so-called baseline pattern. The latency is 5 clock cycles per stage for a single-cast and 6 cycles per stage for a multi-cast operation.

3. The Tool Environment

3.1 Design Objectives
NEC Tokyo will use Anmai as a prototype programming environment to be developed for NEC Cenju-3 users. For a more detailed description of Anmai, the interested reader is referred to [5]. To summarize its major design objectives:
- Application-driven tool design.
- Support of a high-level data-parallel language and/or explicit low-level message passing.
- Use of standardized programming languages (HPF and future extensions at the high level) and machine interfaces (MPI at the low level).
- Support for scientific applications considered today difficult to parallelize on DMPPs (e.g., unstructured sparse matrix computations).

3.2 Components of Anmai
There are three component tools within our tool environment that share a common User Interface (UI): a Parallel-
ization Support Tool (PST), a Parallel Debugging Tool (PDT) and a Performance Monitor and Analyzer (PMA).

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.

Figure 4 shows a typical Anmai user session during performance monitoring and debugging of an HPF program. For the remainder of this section we will explain each of Anmai's components used in this session in more detail.

3.3 Parallelization Support Tool (PST)

PST first consisted of the Oxygen compiler [13]. Oxygen is a parallelizing Fortran compiler providing a global name space on DMPPs. It accepts Fortran 77 annotated with compiler directives for data distribution and loop parallelization, and generates parallel C code with message-passing primitives. For the current PST prototype, we have extended an existing HPF compiler from NEC to allow the same programming paradigm as built into the Oxygen Fortran annotations.

For unstructured computations, such as linear algebra on sparse matrices, a global name space should be supported by run-time analysis, as has been also shown recently by other research groups [3]. In PST we use a mechanism called run-time data consistency analysis which proved to be very effective when parallelizing a solver package used for finite-element based semiconductor device simulation [12].

PST data and loop annotations allow data and control-flow distribution using either the well known HPF static block or cyclic distributions or dynamic user-defined distributions. Critical code segments are compiled into a pre-processing phase (known as the "symbol handler") and executor. The symbol handler can consist of additional iterations when required by non-local data accesses in the critical code segment.

PST allows re-use of communication patterns, once generated by inspector or symbol handler. This is often the case in practice when irregular computation is required, for

![Figure 4](image-url)

Figure 4. A typical Anmai user session with PDT and PMA for debugging and instrumentation configuration of a data parallel program compiled by PST. Program execution, as monitored by output in the separate window, has halted at a previously defined breakpoint (marked with a 'B' sign). Additional instrumentation is similarly being specified prior to resuming execution by selecting points in the UI source browser display.
instance for the iterative solution of linear systems of equations. Another example is the MG code described in Section 4.2.1.

3.4 Parallel Debugging Tool (PDT)

PDT is an interactive, source-level debugger whose aim is to assist programmers in debugging high-level HPF/PST programs, as well as low-level MPI programs based on explicit message-passing. As with any classical debugger, PDT provides means for stopping the target program at points of interest in the execution, and then inspecting its state. Debugging large-scale parallel programs, which typically feature huge amounts of distributed data, can only be efficient if the programmer can manipulate global entities of the program as a whole, i.e., entities involving all of the participating processes. In this respect, PDT supports control and data breakpoints with global break conditions, and provides easily-interpreted, interactive graphical representations of large distributed data objects. At the message-passing level, PDT assists programmers with deadlock detection, race detection, and deterministic replay. Figure 4 shows a typical PDT session. In conjunction with the UI program source browser, the user can set control breakpoints with a mouse-based point and click interface. Figure 5 shows examples of interactive distributed array displays, which may be textual or graphical.

Figure 5. Examples of PDT interactive distributed array displays. Each array element is represented by a rectangular cell, colored to specify on which processor that data item is located. Alternatively, a three-dimensional representation of the actual data values provides a better understanding of their underlying significance, and can help in tracking program evolution.

3.5 Performance Monitor and Analyzer (PMA)

PMA provides facilities to assist with the monitoring and tuning of parallel applications, supporting high-level HPF/PST programs and low-level message-passing programs based on MPI. Initially, PMA is used to configure program instrumentation, ensuring that the desired level of information is collected with minimal intrusiveness. A typical PMA instrumentation session is shown in Figure 4, where the integrated facilities of the UI program source browser are used to select and mark regions which have instrumentation dynamically configured in the loaded (and possibly running) executable. Subsequently, the information collected during program execution is interpreted by PMA for performance analysis and visualization. Targeted understanding and tuning is supported via directed user interaction, selecting critical routines from overall performance profiles for localized investigation right down to the detail of underlying message-passing events and additional memory utilization overheads. Examples of prototype displays covering this hierarchical analysis, and incorporating reference back to the original program source structure, are shown in Figure 6. Program performance can be monitored throughout its execution by requesting updates at intermediate stages, when the collected information may be processed using the parallel computing system itself. Alternatively, after execution has completed, performance profiles can be analyzed off-line and compared with previously stored versions.

Figure 6. Examples of PMA interactive performance displays: high-level performance profiles and execution summaries, and low-level processor utilization and message-passing event visualizations. From the high-level summaries, key routines and source blocks can be targeted for additional instrumentation, which (on re-execution or continued execution) generate detailed low-level traces for in-depth analysis of communication and utilization.

3.6 Annai User Interface (UI)

The UI currently supports browsing through program source files, selecting break-points and checkpoints, and 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. UI is currently being enhanced to provide feedback between PMA and UI. 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 are supported.

4. Parallelization of Applications and Algorithms

4.1 Overview

Application developers in the project are currently concerned with parallelizing a wide range of applications on a number of platforms. 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.

Emphasis has been placed on producing portable message-passing and data-parallel code on different architectures: the Cenju-3 described previously and the Intel Paragon XP/S 5+ featuring 96 Intel i860 XP processors. For a more complete discussion of the Paragon architecture, see [1].

We wished to demonstrate the tool environment’s 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 source code was compiled and executed on both.

4.2 Comparative Evaluation of Applications

From the many benchmark codes available, 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). We investigated some of the interesting aspects of PST, namely user-defined mappings and multiple communication patterns which can be saved and re-used independently. These features are not supported by HPF.

The three benchmarks described below compare the codes parallelized with the high-level data-parallel programming tool PST and low-level message-passing techniques (MPI), in terms of the scalability of the two versions, especially for large problems and 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.

4.2.1 Multigrid NAS Kernel (MG)

The NAS MG kernel calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on an $n \times n \times n$ grid with periodic boundary conditions (see [2]). For our benchmark we used grid size $n = 64$.

The parallelization of this benchmark arose directly from the nature of the problem: the cubic grid was partitioned into blocks. The communication required was then only the exchange of elements between adjacent block faces. The most efficient blocks were found empirically to be "matchsticks": all grid points in one dimension are on one processor and distributed block-wise in the other two dimensions.

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

Results

The streamlining of the code is illustrated in the trace of Figure 8: it is apparent that synchronization between processors during the last three V-cycles is not necessary (i.e., the horizontal bars don’t line up vertically).

The benchmark uses multiple communication patterns for all its critical code segments: the main subroutines are called with decreasing or increasing problem size as the program proceeds through one V-cycle. Since a 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 Figure 8, where a task Gantt chart of the first four V-cycles of a program run is depicted. Performance measurements of Figure 7 refer to the steady state, i.e., the overhead of the symbol-handler is not included in the measurements.

As shown in Figure 7, the performance achieved with the data-parallel program compiled by PST is less than that of the MPI benchmark version, although we do not consider this indicative for performance achieved with MG compiled by future PST versions.

4.2.2 Conjugate Gradient NAS Kernel (CG)

The CG kernel finds an estimate of the smallest 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 had 14,000 columns and 1,853,104 non-zeros ("Class A").

Our approach to parallelization of this 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 were then distributed over all the processors, e.g., block-wise. For maximum efficiency, all $n$-vectors were replicated on all processors. The only communication was in the matrix-vector multiplication: scalar products of matrix rows and the incoming vector were constructed; portions of the resulting vector were gathered on all processors, so that each had its own copy.

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.

Since the CG "Class A" problem is also fairly dense, the cost of having local $n$-vectors is not appreciable when compared to the matrix-vector multiplication.

**Results**

As apparent from the trace displays in Figure 8, 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 Figure 7 for the Cenju-3 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.

### 4.2.3 Finite Element Application

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 [7]. The resulting sparse linear system is solved by the conjugate gradient (CG) method, 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 most time-consuming part is the matrix-vector multiplication in CG which uses indirect addressing, because the sparse matrix is stored in compressed form.

For the MPI version, a replicated data approach was employed. The computation itself was decomposed in the sense that each processor updated only a certain subdo-
main. The replicated data approach had the advantage that the program could be parallelized with minimal alterations to the code. On the other hand, only problems that fit in the local memory of one processor can be solved. The time-dependent case FE2 can not be solved on the Paragon for this reason.

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

The communication in the inner products did not reduce the performance significantly on a small number of processors. The matrix-vector multiplication only needed communication between neighboring sub-domains/processors. Therefore, communication costs could be kept relatively low.

**Results**

Figure 7 shows the performance measured for the finite-element application. On the Cenju-3 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, as explained above. The PST versions have a large initial parallel overhead in address computations, but very good relative speedup figures are observed for larger numbers of processors (e.g. for FE2).
5. Summary

In April 1993, the Joint CSCS-ETH/NEC Collaboration in Parallel Processing was officially started and CSCS received an NEC Cenju-2 with 16 processors as the first development platform in July 1993. On that machine we have developed first prototypes of tools within the integrated environment Annai. Annai has successfully been used in the effective parallelization of a number of applications on various DMPPs.

In July 1994, a 128 processor Cenju-3 was installed at CSCS to replace the Cenju-2. After porting our message-passing platform and the tool environment, we collected first performance measurements, some of which show impressive results although the tools and system software used can still be considered prototypes and performance is likely to improve as the collaboration continues.

Acknowledgments

As part of the 1993 CSCS Summer Student Internship Program, J. Blandy and M. T. Nyeu were involved in first prototype development of PDT and PMA, respectively, and the MPI NAS multigrid kernel was parallelized by U. Kühn, J. Cahouhet from Électricité de France, and G. Jost (NIC) and M. Tomassini (CSCS) have been involved in benchmarking FE1 and FE2.

References


