Preliminary Experience with a π-RED$^+$ Implementation on an nCUBE/2 System

Torsten Bülck, Achim Held, Werner Kluge, Stefan Pantke, Carsten Rathsack, Sven-Bodo Scholz, Raimund Schröder *

March 27, 2021

Abstract

This paper reports on some preliminary experiments with the implementation of a concurrent version of the reduction system π-RED$^+$ on an nCUBE/2 system of up to 32 processing sites. They primarily concern basic concepts of workload partitioning and balancing, the relationship between relative performance gains and the computational complexities of the investigated programs, resource management and suitable system topologies. All programs used for these experiments realize divide_and_conquer algorithms and have been run with varying (sizes of) data sets and system parameters (configurations).

1 Introduction

This paper reports on some preliminary experience with a very recent implementation of a concurrent version of the reduction system π-RED$^+$ [SBK92, GKK92] on an nCUBE/2 system. We favored a distributed over a shared memory system primarily for reasons of scalability, but also for the more challenging organizational problems which make it attractive from a research point of view. The choice of an nCUBE/2 system was primarily motivated by the problem-free portability (and adaptability) of existing code, the ease with which different system topologies may be configured, and by quality of the software (tool) support and cost/hardware-machinery considerations, but less so by peak performance figures.

π-RED$^+$ is an applicative order graph reducer which realizes the reduction semantics of an applied λ-calculus. It accepts as input programs of either of the high-level functional languages KIR or OREL/2[Klu93, SP90] (both of which are dynamically typed, statically scoped and strict) and returns as output partially or completely reduced programs in high-level notation.

The run-time system of π-RED$^+$ is based on an abstract stack machine ASM[Gär91] which serves as an intermediate level of code generation. The current implementation on the nCUBE/2 uses an ASM code interpreter written in

*Christian-Albrechts-Universitaet Kiel, Institut fuer Informatik, D–24105 Kiel, Germany, E–mail: base@informatik.uni–kiel.d400.de

C. Work on a compiler-backend which converts ASM code into nCUBE/2 machine code is currently in progress.

The nCUBE/2 system comprises 32 processing sites, each equipped with an nCUBE/2 processor and 16 MBytes of memory. The processors include autonomous network communication units which may serve 14 bit-serial communication channels per site. The actual system configuration uses up to five of these channels in each site for data transfers from and to as many adjacent sites, and one channel for input/output. Each channel transmits data bi-directionally at a rate of roughly 5 Mbits/sec.

The nCUBE/2 processor features a CISC architecture similar to the MC 68000 wrt instruction set and addressing modes. 16 registers (not including program counter and stack pointer registers) are available for general purposes. Each site runs an operating system kernel nCX.

The implementation of π-RED$^+$ on the nCUBE/2 requires on each site a single nCX master process, under which run the ASM interpreter, a tailor-made micro-kernel which manages several reduction processes as subprocesses, and a process monitoring subsystem.

In the sequel we will outline the basic concept of performing non-sequential computations with π-RED$^+$[Klu83, SGHKW86]. We will also briefly describe the implementation of the micro-kernel and of the event monitoring subsystem which produces run-time data that allow us to extract performance figures and to re-construct in slow motion on a graphical display the creation, allocation and termination of processes within the system. In the main part we will discuss in detail performance figures obtained from running several representative example programs with varying system parameters.

2 Concurrent Computations in π-RED$^+$

Our plan is to systematically investigate, by means of the π-RED$^+$ implementation on the nCUBE/2 systems, various forms of exploiting concurrency in functional systems, among them

- the classical demand-controlled divide_and_conquer approach which directly derives from the recursive nature of functional programs;
- cascading recursive function calls via streams;
- speculative evaluations in alternative program terms;
- systems of cooperating functional processes that communicate via streams.

As a first step, we implemented divide_and_conquer computations, which will be the subject of this paper.

They may be made explicit in high-level functional programs by a construct of the form:

```
letcon
     x_1 = e_1, ... ,x_n = e_n
in e
```

which translates into a λ-term $f\ e\_1 \ldots e\_n$, where f denotes an abstraction $\lambda\ x\_1 \ldots \lambda\ x\_n\ e$ [1]. Its evaluation under an applicative order reduction regime is recursively defined as

$$\textsc{eval}(f\ e\_1 \ldots e\_n) = \textsc{eval}(f\ \textsc{eval}(e\_1) \ldots \textsc{eval}(e\_n)) \ .$$

The recursive nesting of EVALs defines a hierarchy of (or a parent-child relationship between) evaluator instances, of which those that apply to the argument terms $e\_1, \ldots, e\_n$ can be executed concurrently, or in any order. We are free to associate with every evaluator instance a process. A parent process that evaluates the application $f\ e\_1 \ldots e\_n$ may therefore create concurrently executable child processes for any subset of its argument terms, and evaluate the remaining arguments under its own control. The creation of further child processes may recursively continue until some upper bound which saturates the processing capacity of the system is reached.

A typical interaction scheme between parent and child processes, as it is realized in π-RED$^+$, is depicted in fig. 1. Here a process is completely specified by a triple $< p, t, [e, k] >$, of which $[\ldots]$ contains its changeable parts [Klu83, SGHKW86]. In particular

e denotes the program term executed under the control of the process;

k denotes a count value which gives the actual number of reduction steps the process is still allowed to perform on e (and must be decremented upon each instance of a reduction);

p identifies the parent process by which the process under consideration was created and to which it must eventually return the partially or completely evaluated term e;

t denotes a unique place-holder token which identifies, in the term executed by the parent process p, the syntactical position into which the (partially) evaluated term e must be inserted.

The figure shows a parent process pm which originates from yet another process pp. In the course of evaluating the application $f\ e\_1 \ldots e\_n$, it creates child processes $p1, \ldots pi$ for the argument terms $e\_1, \ldots, e\_i$ in that it abstracts them out and replaces them with the place–holder tokens $t1, \ldots ti$, respectively. These tokens are also inserted into the respective child process triples. All child processes are also initialized with the actual reduction count value k of the parent process pm.

The processes pm and $p1, \ldots pi$ can now proceed concurrently to reduce the argument terms. Synchronization between parent and child processes occurs upon termination of the latter, either after having decremented their count values to zero or after having reached the (weak) normal forms of their terms, denoted as $e\_1^N, \ldots, e\_i^N$ (which is the situation shown in the figure). Synchronization includes the substitution of the place-holders in the term reduced under the control of the parent process by the evaluated argument terms that return from the child processes. Thereupon, the parent process pm may continue to

[1] Alternatively, any other program term e with subterms $e\_1 \ldots e\_n$ that are to be set up for concurrent evaluation can be brought into this form by pre-processing.

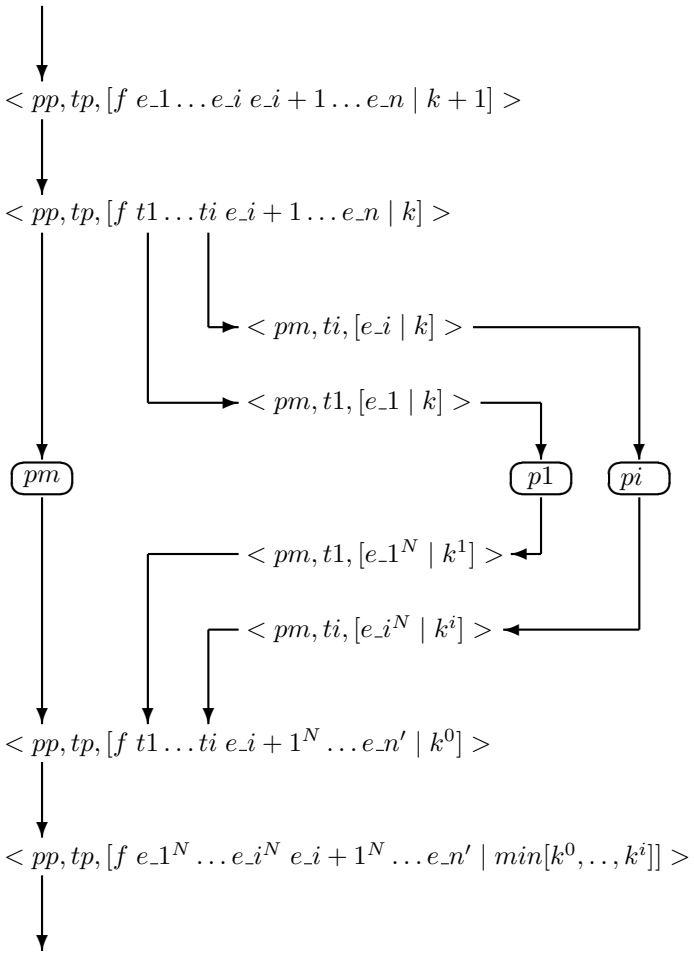

Figure 1: Process interaction under demand-controlled divide_and_conquer computations

reduce the remaining application, with the count value k set to the minimum of the count values left over by all processes at the point of synchronization.

Concurrent processes within the evolving hierarchy can be scheduled non-preemptively and truly in any order. Neither different priorities nor fairness regulations need therefore be taken into consideration.

Stability of the entire computation can be guaranteed by two simple measures which in fact realize system-specific invariants. The creation of new processes is made dependent on the availability of place-holder tokens in a system-supported finite reservoir. Potential instances of spawning new processes can only succeed if the appropriate number of tokens can be allocated (and thereby removed) from the reservoir, otherwise the parent processes simply continue by evaluating the respective terms under their own regimes. Terminating processes de–allocate the tokens they hold in possession and recycle them to the reservoir. Tokens are allocated dynamically on a first come/first serve basis and

under complete system control. The total number of processes that at any time participate in a computation can never exceed the number of tokens with which the reservoir was initialized [Klu83].

The primary purpose of counting reduction steps is to terminate in an orderly form potentially non-terminating recursions. An initial count value which the user is asked to specify prior to every program run is monotonically decremented upon each reduction step. Even though actual count values are replicated whenever new processes are being created, they are bound to decrement to zero eventually. The entire computation comes to a halt if this value is reached in any of the branches of the process hierarchy since points of process synchronization pass on the minimal count value of any of the synchronized processes. Irrespective of the way a computation terminates, the system always assembles a complete program (either fully or partially reduced) which is returned to the user in high-level notation.

3 The Implementation of π-RED$^+$ on the nCUBE/2

When performing divide_and_conquer computations in a distributed memory system, one processing site (to which we will henceforth refer as site 0) usually starts with some initial parent process, from where the computation spreads out over all the other processing sites. A process that executes in a particular site may create child processes in topologically adjacent sites until all sites eventually become active, provided the particular application problem yields as much concurrency. In order to avoid idling processing sites, the process hierarchy ought to unfold several times over the entire system so that each processing site holds a pool of processes, of which some are executable and others are temporarily suspended.

Balancing the workload irrespective of sizes and structures of the application problems requires full system control over the partitioning of programs into concurrently executable pieces and their allocation to processing sites. This must be done in compliance with specific program properties on the one hand and the availability of system resources (processing and memory capacity) on the other hand. Ideal for a simple workload distribution and balancing scheme is a symmetric system topology in which each processing site

- is either physically or at least logically interconnected with the same number of adjacent sites;
- has concessions, in the form of tickets held in a local pool, to distribute to each of its adjacent sites the same number of (child) processes; the concessions (tickets) may be claimed by any of the processes executing in the site.

In an nCUBE/2 (or subcube) of some 2^n processing sites there are n immediate neighbors to each site. With k tickets per neighbor available in each site, we may have up to $2^n * n * k + 1$ processes in the system at anyone time [2].

[2] In addition to the $n * k$ processes it receives from its adjacent sites, there may be one more process in site 0.

However, since all site-to-site data transfers are transparent to the user, any other system topology (e.g., all sites are fully connected or interconnected as a tree) may be logically realized on the nCUBE/2 as well.

Installing concurrent π-RED$^+$ on the nCUBE/2 requires a single nCX master process in each site which runs as a subsystem a tailor-made operating system kernel which manages reduction processes. The subsystem also includes the ASM interpreter (or processor) and a process monitor.

The process scheduling scheme realized by the OS kernel is depicted in fig. 2. In addition to the usual queues and tables, it includes input/output buffer areas and the local ticket pool.

The buffer areas serve as interfaces between the nCX master process and the network communication unit. Every program term arriving at a site through one of the interconnection lines is written into the input buffer, from where the nCX process copies it into its own address space. Conversely, every program term that must be transferred to some other site is by the nCX process written into the output buffer, from where the network communication unit takes over.

The local ticket pools are initialized with some $n * k$ tokens, of which k each are designated for process creation in one of the n adjacent sites.

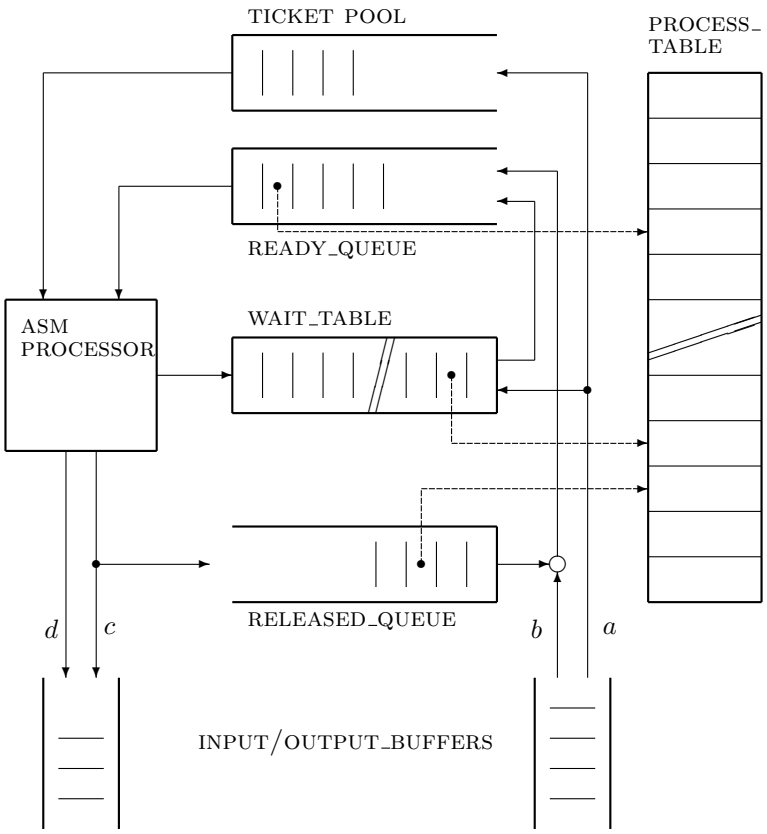

Figure 2: Process scheduling

The process table is implemented as an array of $n * k$ context block frames

since each site may receive as many program terms for processing as it has concessions (tickets) to send terms off to other sites. Pointers to unused context blocks are held in the *released_queue*. The *ready_queue* and the *wait_table* respectively hold pointers to context blocks of executable processes and of parent processes waiting for the synchronization with child processes. These data structures too must provide for $n * k$ pointer entries [3].

A program term transmitted to a site for execution enters through the $input\_buffer$, picks up the pointer of a free context frame from the *released_queue* to create a new process which immediately lines up in the *ready_queue* (arrow labeled *b* in the figure). A terminating process returns its context frame pointer to the *released_queue* and the normal form of the program term, via the $output\_buffer$, to the site from which it was received (arrow *c*).

An active process may create a child process by consuming a ticket from the local pool, if one is actually available, and by sending the term off, via the $output\_buffer$, to the processing site for which the ticket is designated (arrow *d*). Master processes that cannot continue until synchronization with child processes are temporarily suspended by putting their context frame pointers into the wait table. An evaluated term that returns, via the $input\_buffer$, to a site synchronizes with its (suspended) parent process and returns its ticket to the pool (arrow *a*).

4 The Process Monitoring System

In order to obtain some accumulative performance figures and to gain insight into the actual course of program execution, our nCUBE/2 implementation has been augmented by a process monitoring systems. It records in each processing site individually relevant events such as the creation, termination and other status changes of processes, the allocation and de-allocation of heap space, the transmission of messages, etc. These events are in their order of occurrences over time written into site-specific monitor files, as the computation proceeds. After termination, all monitor files are merged into one, again with all events ordered in time, and the merged file may then be passed on to some evaluation and display tool which produces graphical output, either in the form of performance plots or in the form of a graphical animation which shows what was going on in the system during program execution.

A major design objective for the monitoring system was to provide some degree of freedom in choosing the events that are to be recorded for subsequent evaluation, and to have the recording interfere with, and thus distort, as little as possible the reduction processes proper. This can largely be achieved by generating, as an integral part of compiling application programs to ASM code, dedicated monitor instructions which produce event-specific entries in the monitor files. Inserting these instructions into the ASM code is controlled by parameters that specify the chosen (sub)set of events. However, the selection of events that relate to process scheduling and memory (heap and stack) management must be handled differently since they are effected by the micro-kernel code which cannot be changed between program runs. Instead, the monitoring

[3] The data structures in processing site 0 must have one more entry to accommodate the initial process.

system provides a set of library monitor functions from which the ones that are to be effective during a program run are accessed via a function table.

The evaluation and display tool provides a rich variety of features which can be easily extended by the user. To customize the analysis and to create graphical displays, the merged monitor file is piped through a script that produces graphics control sequences which drive the front-end of the display tool. These control sequences either produce plots of accumulative performance figures or re-play in slow motion the computation as it proceeds in time and spreads out over the processing sites, e.g., in terms of actual process slot or memory space allocation, or in terms of processor/memory utilization.

The main analysis module generates plots which simply relate two quantities to each other. These quantities may be data that are either directly read from the monitor file or obtained after some complex computations. The respective program modules are contained in scripts that are either provided by the monitoring system or specified by the user in a superset of C.

A special display module is available for animation purposes. For instance, to display process allocation over time it creates a plot as depicted in fig. 3. Along the vertical and horizontal axes it respectively displays processing sites and available process slots in each site. The slots are filled with different colors to indicate whether or not a process is allocated at all, nd if so, in which state it is (executing, ready for execution, or suspended). The small square on the right depicts on a percentage basis the accumulated processor utilization for reductions, communication handling, and idle times (from left to right). The path through the process slots in different sites connects processes that are in a parent-child relationship with each other, with the topmost parent process in the upper left corner. The situation shown in this figure is a snapshot taken at some instant in time.

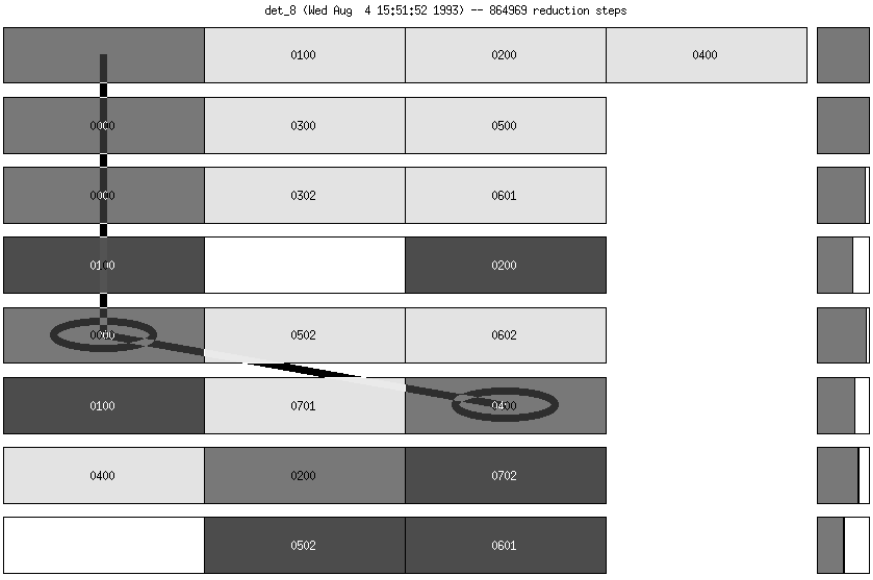

Figure 3: Process layout and workload balancing

5 Performance Measurements on Selected Example Programs

In this section we will discuss some performance figures measured on selected example programs with varying system parameters. All figures are given in terms of speedups relative to the execution times of the same programs on a single nCUBE/2 processing site. They are based on the interpretation of ASM code rather than on the execution of compiled nCUBE/2 machine code. Since the latter can be expected to be faster than the former by up to an order of magnitude and more, depending on particularities of the application programs, the respective figures for compiled nCUBE/2 code are bound to look decidedly less favorable. Nevertheless, the data obtained from interpretation at least indicate to which extent system configuration (network topology), system parameters and program properties influence workload distribution, balancing and performance gains.

The programs that were investigated are the following:

`fib` - which computes the Fibonacci numbers by two-fold recursive induction;

`towers` - which computes the sequence of disc moves for the towers_of_hanoi problem, again by recursive induction;

`quick` - which quicksorts a sequence of unsorted numbers (the sequences are chosen so that they recursively break up into about equally sized subsequences);

`queens` - which solves the n-queens problem in that it computes all possible placements by recursively separating a first placement from the remaining placements;

`det` - which computes the determinant of a square-shaped matrix by recursive expansion along the first row (multiplication of the leftmost uppermost element with the matrix from which the first row and the first column are dropped);

`pat_mat` - which searches, by recursive partitioning, through a square-shaped matrix of numbers for occurrences of a specific submatrix;

`red_add` - which recursively partitions a matrix along rows into equally sized submatrices and performs reductions by addition along the columns;

`ma_mult` - which recursively partitions matrix multiplications into multiplications of equally sized submatrices;

`mandel` - which computes graphical representations of Mandelbrodt sets by recursive division into subsets.

They were all run with the following system configurations:

- with (sub)systems of 2^n processing sites, and $n \in \{1, 2, \ldots, 5\}$;
- with all sites of a subsystem logically fully connected, i.e., each site may directly communicate program terms to each other site;

- with the subsystems operated as hypercubes, i.e., each site may directly communicate with the n adjacent sites whose addresses differ in just one binary position;
- and, as somewhat exotic configurations, with the subsystems configured as trees.

In both the fully connected and the hypercube configurations, all nodes were initialized with some $k \in \{1, 2, \ldots, 8\}$ tickets per interconnection. In the tree only one concession per interconnection from an inner node to a successor node made sense, i.e., the topmost root node could not receive child processes and the leaf nodes were not permitted to create child processes in other nodes.

Moreover, workload was distributed so that a process which executes an application $f\ e\_1\ \ldots\ e\_n$ that is earmarked for concurrent evaluation

- either creates processes in adjacent sites for all its subterms $e\_1, \ldots, e\_n$ as long as tickets are actually available in the pool and reduces the remaining terms, if any are left, under its own regime;
- or it always reduces at least one of the subterms itself (i.e., with a two-fold expansion of the application problem as the standard case, one subterm is transferred to another site, the other one is evaluated by the parent process).

The latter distribution scheme is the only one that has been applied to tree configurations, as otherwise only about half the number of processing sites would be involved in the computation.

In some of the programs we also used thresholds on the recursion depth beyond which the creation of new processes is surpressed irrespective of the availability of tickets. This measure is intended to prevent the job granularity from becoming too fine in relation to the overhead involved in creating new processes.

Figures 4, 5, 6 and 7 show, as representative examples, the results of some systematic performance measurements with varying system parameters and configurations for the programs `quick`, `det`, `mat_mult` and `mandel`. They show relative performance gains versus system configurations, with `hyp<n>` denoting hypercubes and `sym<n>` denoting fully inter connected systems, in each case with n denoting the number of tokens that is available per interconnection. The actual number of processing sites is represented by dots of different shapes, which also distinguish distribution of all (or of as many as possible) subterms, denoted as `dist all`, from distribution of all but one subterm, denoted as `dist n-1` (the shapes of the dots are defined in the boxes in the upper right corners). Based on these diagrams, the following general observations can be made which also apply to the programs for which no performance figures are shown.

The overriding factor in achieving a performance gain that grows nearly linearly with the number of processing sites involved is the ratio between the complexity of the algorithm, measured, say, in numbers of recursions or in numbers of data elements to be processed, and the complexity of communicating a program term (in most cases a data structure) from one processing site to another. If this ratio is nearly one (or some other nearly constant value) no significant performance gain can be expected. This may be exemplified by the `quick` program on the one hand, and the `det` program on the other hand. In

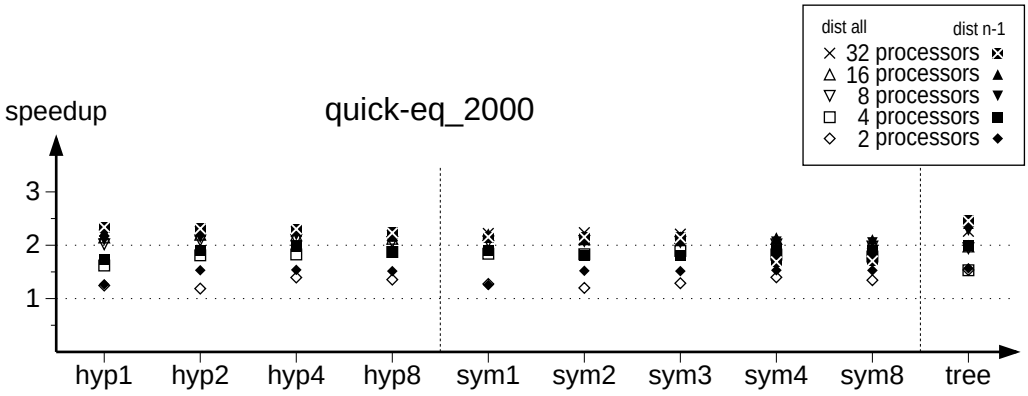

Figure 4: Sorting a list of 2000 elements

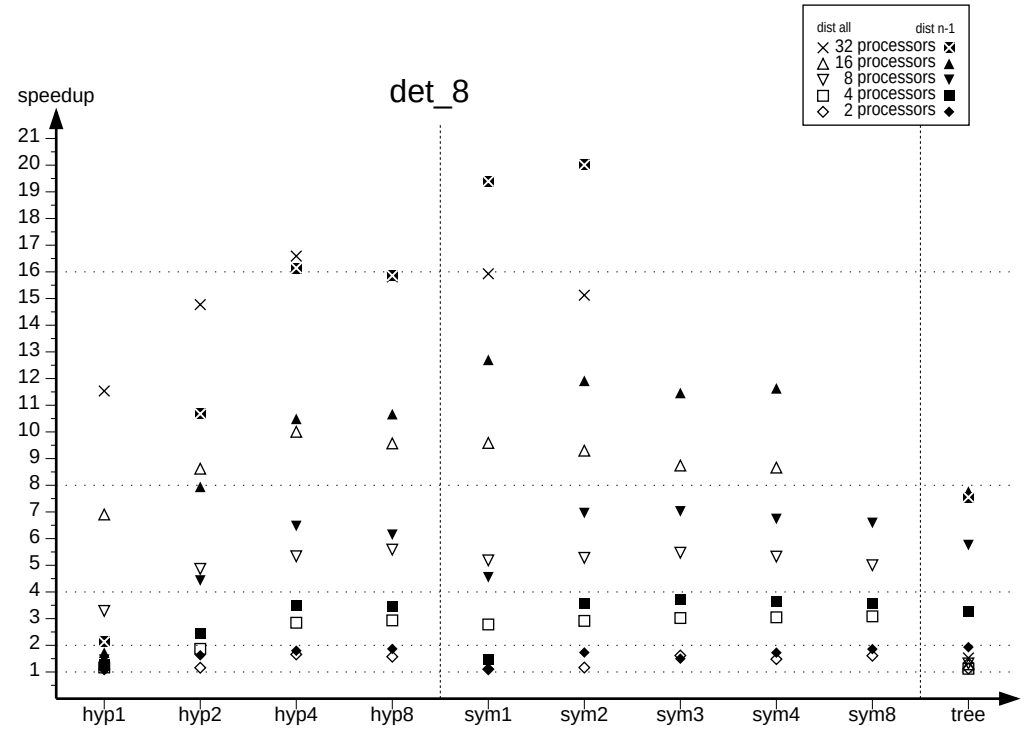

Figure 5: Computing the determinant of an 8×8 matrice

the former case, the complexity of the algorithm is $O(n * lg\ n)$ and the length of the sequences to be moved among processing sites is n , hence the performance gain can only be marginal. In the latter case, we have a computational complexity of $O(n!)$ and must move matrices of sizes (n^2). Since $n!$ grows much faster than n^2 with increasing n, performance gains are primarily limited by the number of processing sites and thus grow linearly with them.

Other bad candidates for performance gains are the towers_of_hanoi program

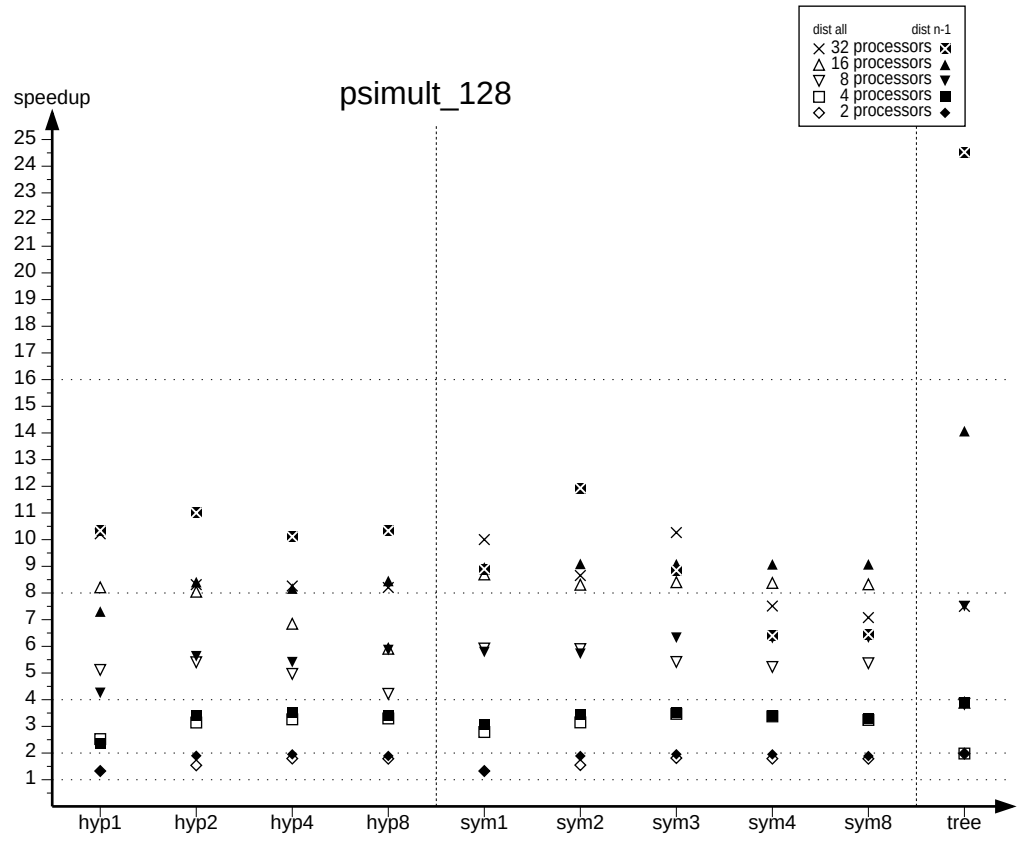

Figure 6: Multiplying two 128×128 matrices

and reduction along rows or columns of a matrix. The computational complexities of all other programs that were investigated grow much faster than the complexities of moving the data structures and thus yield performance gains that are nearly linear wrt the number of processing sites.

The cause of this phenomenon is predominantly of a purely technical nature. Since the nCX master processes which in each site run the reduction system proper communicate with the network communication units through dedicated buffer areas, the nCUBE/2 processors are always involved in site-to-site data transfers in that they have to load and unload these buffers. The time that must be devoted to these transfers is taken away from useful computations. Thus, with computational complexities about the same as transfer complexities, there is little difference, as far as total program run-time is concerned, between distributing the computation over several processing sites versus doing everything sequentially in one site.

Somewhat surprising is the observation that for all example programs run-time performance significantly improves with increasing numbers of tickets and thus with increasing granularities of the computations performed by the processes. As in shared memory systems, one would expect the opposite effect since creating processes far in excess of the available processing sites (which definitely is the case with 2 or more tickets per interconnection and site) means

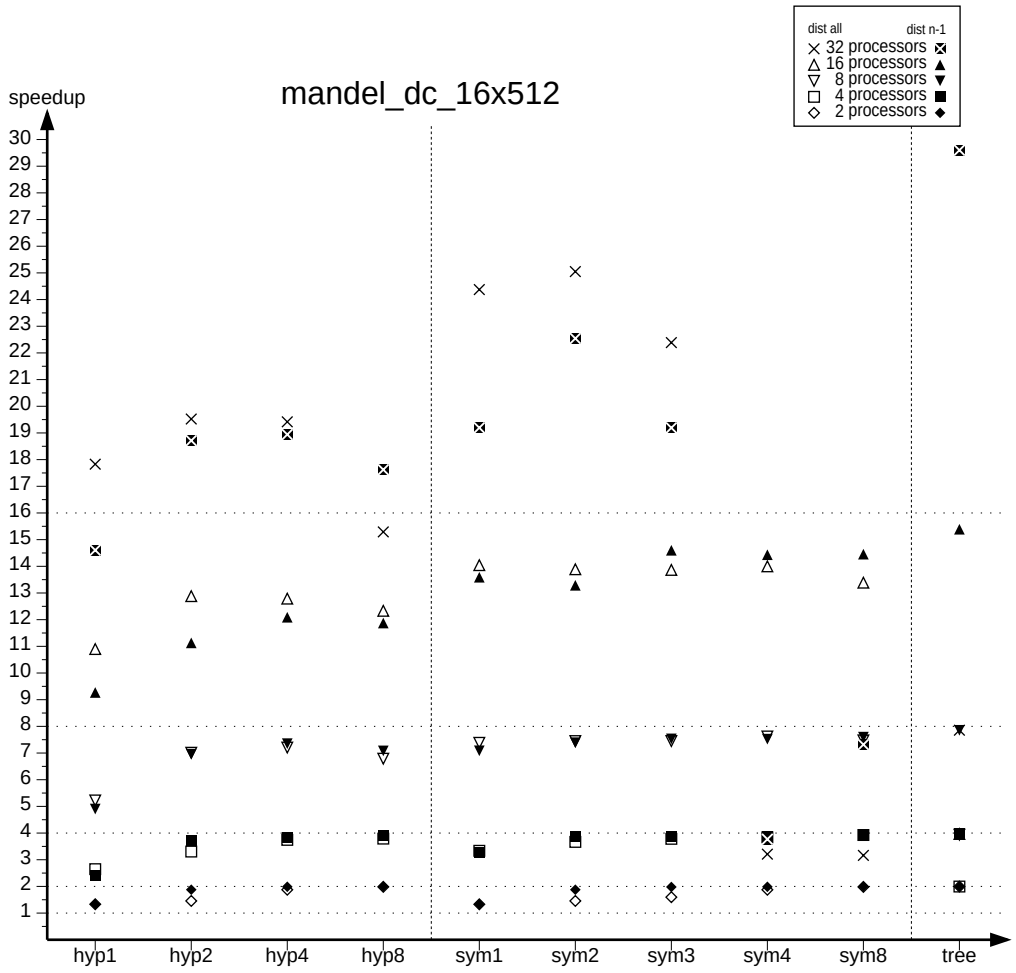

Figure 7: computing a 16×512 points wide subset of the Mandelbrodt set

more management overhead at the expense of useful computations. However, this negative effect is more than offset by a more balanced overall workload distribution. With more processes allocated to each node there are generally more opportunities to replace terminating processes immediately with executable processes lined up in the local *ready_queues*, as watching the respective animation movies confirms.

On average, performance is slightly better if at least one of the concurrently executable terms is reduced under the control of the parent process, as opposed to creating new processes in adjacent sites for all terms. This phenomenon is somewhat difficult and speculative to explain. When spawning fewer processes per instance one would expect that the computation takes more time to spread out over the available processing sites and thus utilize them less efficiently. However, the performance data and the animation movies seem to indicate that it inflicts slightly less processing time spent on management overhead and slightly less idle time due to parent processes waiting for synchronization with their

child processes. Also, on average there are slightly more executable processes per site.

The choice between hypercubes and fully interconnected configurations has only a marginal effect on performance. However, full interconnections cause another problem with increasing numbers of processing sites: since too many processes must be accommodated per site, many programs run out of memory space since the partitions that can be allocated per process become too small. A tree configuration seems to have a decisive performance edge for all application problems which unfold reasonably balanced trees, as for instance matrix multiplication or the recursive partitioning of the Mandelbrodt problem. The symmetric network configurations do significantly better with application problems that develop unbalanced structures (as, for instance, the queens problem).

Limiting the recursion depth up to which an application problem may be split up into concurrently executable pieces generally improves the performance by some 10 to 20 percent since it prevents fine granularities and thus process run-times that are too short in relation to the overhead of managing them. However, the same ends can be more easily achieved by adapting the the size of the subcube (or the number of processors participating in the computation) to the size of the application problem. In order to maintain a high ratio of useful computations vs communication and process management overhead, it is generally important that after exhaustion of all ticket pools the system engages in lengthy periods of sequential computations which are only infrequently interrupted by process switches and data communications. For this to be the case the application problem must primarily be prevented from spreading out too thinly over too many processing sites rather than sustaining a fairly large number of processes per site.

6 Conclusion

The current nCUBE/2 implementation of π-RED$^+$ is operational since May '93 and reasonably stable since July '93. The performance measurements discussed in this paper are only of a preliminary nature for two reasons. Firstly, we have so far investigated only the usual set of toy programs with very predictable behavior. More serious and complex application programs suitable for divide_and_conquer computations are currently being developed as part of a graduate programming course. Secondly, all measurements are based on the interpretation of abstract machine code and thus are not very conclusive wrt to performance figures that can be obtained from executing compiled nCUBE/2 code. While the overhead for process management and data communication remains about the same, the speed of executing compiled nCUBE/2 code vs interpreting ASM code can be expected to improve by at least an order of magnitude, i.e., the performance gains of non-sequential processing may depreciate considerably. It is not yet clear whether they can be offset in all cases by decidedly larger problem sizes as they also require more processing time for data transfers.

References

[Gär91] D.Gärtner: *π-RED$^+$: Ein Interaktives Codeausführendes Reduktionssystem zur Vollständigen Realisierung eines Angewandten λ-Kalküls.* PhD Thesis (in German), Internal report, University of Kiel, 1991

[GKK92] D.Gärtner, A.Kimms, W.E.Kluge: *π-RED$^+$—A Compiling Graph Reduction System for a Full Fledged λ-Calculus.* Proc. of the 4th International Workshop on Parallel Implementation of Functional Languages, eds H.Kuchen, R.Loogen, University of Aachen, 1992

[Klu93] W.E. Kluge: *A User's Guide for the Reduction System π-RED.* Internal report, University of Kiel, 1993

[Klu83] W.E.Kluge: *Cooperating Reduction Machines.* IEEE Transactions on Computers, Vol. C-32, No 11, 1983, pp. 1002-1012

[SBK92] C.Schmittgen, H.Blödorn, W.E.Kluge: *π-RED*- a Graph Reducer for Full-Fledged λ-Calculus.* NGC, Vol. 10 No 2, Ohmsha and Springer Verlag, 1992, pp. 173-195

[SGHKW86] C.Schmittgen, A.Gerdts, J.Haumann, W.E.Kluge, M.Woitass: *A System-Supported Workload Balancing Scheme for Cooperating Reduction machines.* 19th Hawaii International Conference on System Sciences, Vol. I, 1986, pp. 67-77

[SP90] H.Schlütter, E.Pless: *Die Reduktionssprache* OREL/2. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, 1990