Experience with the Implementation of a Concurrent Graph Reduction System on an nCUBE/2 Platform

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

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

Abstract. This paper reports on some experiments with the implementation of a concurrent version of a graph 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

Running complex application programs non-sequentially in multiprocessor systems is known to be a formidable organizational problem. It relates to a programming paradigm suitable for exposing problem-inherent concurrency, to the orderly cooperation of all processes participating in the computation, and to a process management discipline which ensures a stable overall system behavior and an efficient utilization of resources.

The outcome of decisions in non-trivial programs and, hence, the workload they generate generally depends on actual parameter values and can therefore not be anticipated. This rules out efficient static schedules for non-sequential processing worked out by the compiler or by the programmer. Instead, workload partitioning and scheduling should be dynamically controlled by the system in order to achieve a reasonably balanced workload distribution over the available processing sites. The course of actions to be taken in concrete situations (states of program execution) must be inferred by the program code itself (usually by control constructs which identify opportunities to spark off new processes), in compliance with the actual states of load distribution.

The functional programming paradigm is known to be perfectly suited for this purpose. Programs are pure algorithms which are liberated from all procedural elements, feature simple recursive control structures which lend themselves elegantly to divide_and_conquer techniques, and - most importantly - they are free of side-effects, i.e., the determinacy of results is guaranteed by the Church-Rosser property. The task structures which dynamically evolve when executing functional programs non-sequentially are strictly hierarchical, communications are locally confined and governed by tight synchronization margins. Hierarchical structures also are inherently free of deadlocks.

The simplicity of this concept led to many proposals and system implementations for non-sequential program execution [AJ89, GH86, HB93, HR86, Klu83, PvE93, JCSH87, Sch92, SGH$^+$86]. This paper reports on experiences with the implementation of a concurrent version of the reduction system π-RED$^+$ [SBK92, GKK92] on an nCUBE/2 system. π-RED$^+$ is an interactively controlled applicative order graph reducer developed at the University of Kiel which truthfully realizes the reduction semantics of an applied λ-calculus. It accepts as input programs of the high-level functional language KIR[Klu93] (which is 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 which serves as an intermediate level of code generation. The current implementation on the nCUBE/2 uses an ASM code interpreter written in C. Work on a compiler-backend which converts ASM code into nCUBE/2 machine code is currently in progress.

Our nCUBE/2 configuration comprises 32 processing sites, each equipped with an nCUBE/2 processor, 16 MBytes of local memory and an autonomous network communication unit which, in the particular setting, serves up to 5 bit-serial communication channels per site for data transfers from and to as many physically adjacent sites. Each channel transmits data bi-directionally at a rate of roughly 4 Mbits/sec. Each site is controlled by a UNIX-like operating system kernel nCX.

In the sequel we will outline the basic concept of performing non-sequential computations with π-RED$^+$. We will also briefly describe the implementation of the system, and 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$^+$

The purpose of implementing π-RED$^+$ on the nCUBE/2 is to provide a versatile testbed for a systematic investigation of several ways of exploiting concurrency wrt

- the organizational measures that are necessary to enforce a stable system behavior which satisfies essential invariance (safety and liveness) properties;
- workload partitioning (granularity), distribution and balancing, scheduling disciplines and fairness regulations (if necessary);
- the influence of system configurations (interconnection topologies), task granularities, algorithmic and communication complexities on the ratios of useful computations vs process management and communication overhead, and thus on net performance gains.

As a first step, we investigated divide_and_conquer computations based on a system concept which was proposed as early as 1983 [Klu83] and first implemented as a distributed string reduction simulator on a network of four PDP 11/20 processing nodes by 1984 [SGH+86]. The load distribution and performance figures obtained from this implementation were rather deceptive due to the small number of processing sites and, even more so, due to the n^2-complexity problem inherent in string reductions, against which the overhead of communicating program terms among processing sites became almost totally negligible, yielding nearly ideal performance gains.

With a more advanced hardware platform (up to 32 processing sites, far more memory capacity, faster communication links) and efficient graph reduction techniques at hand, which allow for more complex application programs, this picture is bound to look quite differently.

A typical example for exploiting concurrency in functional programs is the towers_of_hanoi problem. It consists in computing the sequence of moves necessary to transfer a stack of disks with different diameters from a location `A` to another location `B` using a third location `C` where disks can be put away temporarily, so that in all three locations the disks are always stacked up in the order of monotonically decreasing diameters. In KIR-notation, the program looks like this:

```
def
  hanoi[ n, x, y, z ] = if ( n eq 1)
                        then << x, y >>
                        else let k = ( n - 1 )
                             in ( hanoi [ k, x, z, y ] ++
                                  ( << x, y >> ++ hanoi [ k, z, y, x ] ))
in hanoi [ h, A, B, C ]
```

where "<" and ">" are delimiters of n-ary sequences (lists) of elements, "++" denotes list catenation, and `h` denotes the number of disks that are initially on stack `A`.

Though the concurrency inherent in this program in the form of the two recursive calls of `hanoi` can be easily detected by a compiler, it is in more complex programs often helpful to make it explicit, using a construct of the form:

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

With it the function `hanoi` would have to be redefined as:

```
  hanoi [ n, x, y, z ] = if ( n eq 1)
                         then << x, y >>
                         else let k = ( n - 1 )
                               in letpar
                                       p = hanoi [ k, x, z, y ],
                                       q = hanoi [ k, z, y, x ]
                                  in ( p ++ ( << x, y >> ++ q ))     .
```

π-RED$^+$ translates `letpar`-constructs as above into a function application f `e_1` ... `e_n`, where f denotes a function with the formal parameters `x_1` ... `x_n` and body `e`[1]. Its evaluation under an applicative order reduction regime is recursively defined as

$$\text{EVAL}(f\ \texttt{e\_1} \ldots \texttt{e\_n}) = \text{EVAL}(f\ \text{EVAL}(\texttt{e\_1}) \ldots \text{EVAL}(\texttt{e\_n}))\ .$$

The recursive nesting of EVALs in fact defines a hierarchy of (or a parent-child relationship between) evaluator instances, of which those that apply to the argument terms `e_1` ... `e_n` can be executed concurrently, or in any order. We are free to associate with each evaluator instance a process (or a thread). A parent process that evaluates the application f `e_1` ... `e_n` may therefore create concurrently executable child processes for any subset of its argument terms `e_1` ... `e_n`, and evaluate the remaining arguments under its own control. The creation of further child processes may recursively continue until some upper bound is reached which saturates the processing capacity of the system.

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

Stability of the entire computation can be guaranteed by two simple measures which in fact realize system-specific invariance properties.

The creation of new processes is made dependent on the availability of place-holder tokens (tickets) in a system-supported finite reservoir. Potential instances of spawning new processes can only succeed if the appropriate number of tickets 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 tickets they hold in possession and recycle them to the reservoir. Tickets 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 tickets with which the reservoir was initialized [Klu83].

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

When performing divide_and_conquer computations in a distributed system, identical copies of the complete program code are first downloaded into all the local memories. Thereupon, one designated processing site starts with some initial parent process, from where the computation spreads out, by recursive creation of child processes, over all the other processing sites. In order to avoid idling processing sites, the process hierarchy ought to unfold several times over the entire system so that each site holds a pool of processes, of which generally some are executable and others are temporarily suspended.

Ideal for a simple workload distribution and balancing scheme is a symmetric system topology in which each processing site

[1] Alternatively, any other program term `e` with subterms `e_1` ... `e_n` that are to be set up for concurrent evaluation can be brought into this form by pre-processing.

– 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.

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 and also supports the ASM interpreter.

The process scheduling scheme realized by the OS kernel is depicted in fig. 1. In addition to the usual queues and tables, it includes input/output buffer areas and the local ticket pool.

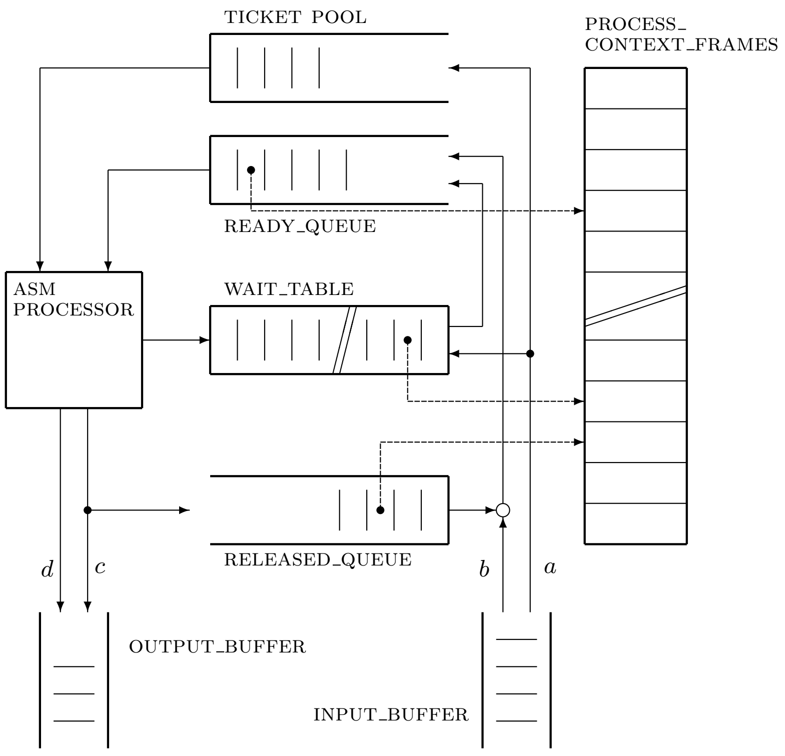

Fig. 1. Process scheduling

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 transmits 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 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 (which includes reference counting and dynamic type checking).

The programs that were investigated include the following:

`hanoi` - which computes, by recursive induction, the sequence of disk moves for the towers_of_hanoi problem;
`det` - which computes the determinant of a square-shaped matrix by recursive expansion along the first row;
`mandel` - which computes graphical representations of Mandelbrodt sets by recursive division into subsets of rows;
`fractal` - which computes graphical representations of fractals by recursive composition of basic structures (with a recursion depth of 10).

They were all run with 2^n ($n \in \{1, 2, \ldots, 5\}$) processing sites and with the following system configurations:

- with all sites of a subsystem logically fully connected, i.e., each site may create subprocesses on each other site;
- with the subsystems operated as hypercubes, i.e., each site may create subprocesses on its n physically adjacent sites;
- and the subsystems configured as binary trees.

In both the fully connected and the hypercube configurations, all nodes were initialized with some $k \in \{1, 2, 4, 8\}$ tickets per interconnection. The tree configuration was run with only one ticket per interconnection from an inner node to a successor node, i.e., the topmost root node could not receive child processes from, and the leaf nodes were not permitted to create child processes in other nodes.

Moreover, workload was distributed so that a process executing an application f `e_1 ... e_n` that is earmarked for concurrent evaluation either

- creates processes in adjacent sites for all its subterms `e_1 ... 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 was applied to tree configurations, as otherwise only about half the number of processing sites would be involved in the computation.

Figures 2, 3, 4 and 5 show, as representative examples, the results of some systematic performance measurements with varying system parameters and configurations for the programs `hanoi`, `det`, `mandel` and `fractal`. They show relative performance gains versus system configurations, with `hyp<n>` denoting hypercubes and `sym<n>` denoting fully interconnected systems, in each case with `n` specifying the number of tickets 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).

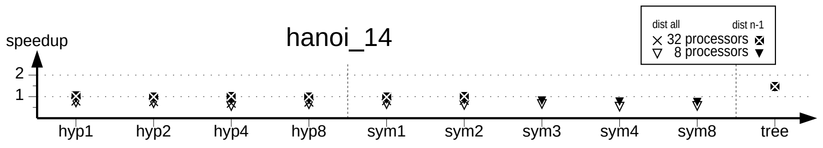

Fig. 2. Solving the towers_of_hanoi problem for 14 disks

Based on these diagrams, the following general observations can be made.

The overriding factor in achieving performance gains that grow nearly linearly with the number of processing sites involved is the ratio between the complexity of the algorithms, measured, say, in numbers of recursion steps or in numbers of data elements to be processed, and the complexities of communicating program terms (in most cases essentially data structures) among processing sites. 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 `hanoi` program for which both the complexity of the algorithm and the length of the sequences to be moved among processing sites are roughly $O(2^n)$.

In contrast, the `det` program has a computational complexity of $O(n!)$, largely due to the considerable redundancy of the algorithm, and must move

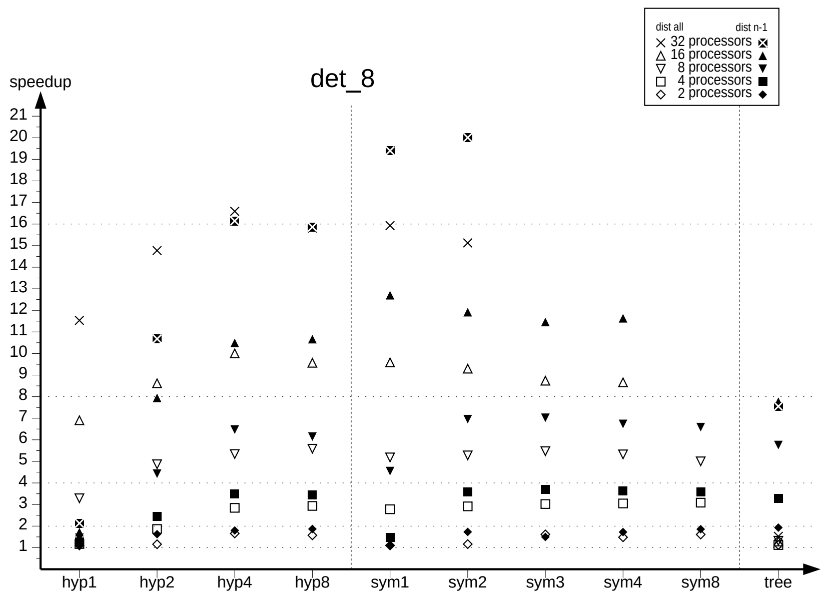

Fig. 3. Computing the determinant of an 8×8 matrix

matrices of sizes $O(n^2)$. 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.

The computational complexities of the two other example programs 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.

Another general observation concerns job granularity, which becomes finer with increasing numbers of tickets. When running the example programs on the hypercube configurations, the performance figures improve with increasing numbers of tickets (some slightly drop again when using 8 tickets). 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 more management overhead at the expense of useful computations. However, this negative effect seems to be more than offset, at least up to 4 tickets, 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. To some lesser extent, the same observations can be made for some of the example programs with the fully interconnected configuration. However, with the `fractal` program, we clearly have a performance degradation with increasing numbers of tickets, as expected.

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

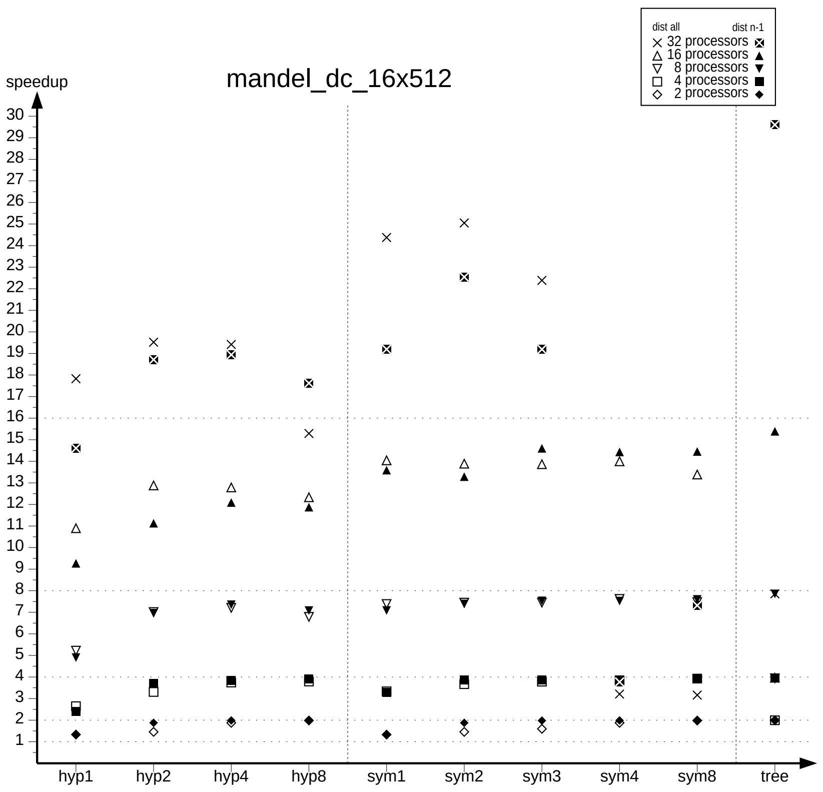

Fig. 4. Computing a 16 × 512 points wide subset of the Mandelbrodt set

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 seem to indicate that it inflicts slightly less processing time spent on management overhead and slightly less idle time caused by 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. This explains why some of the dots are missing from the diagrams, e.g., those for the `determinant` program executed on 32 processors configured as `sym3`, `sym4`, or `sym8`.

A tree configuration seems to have a decisive performance edge for all ap-

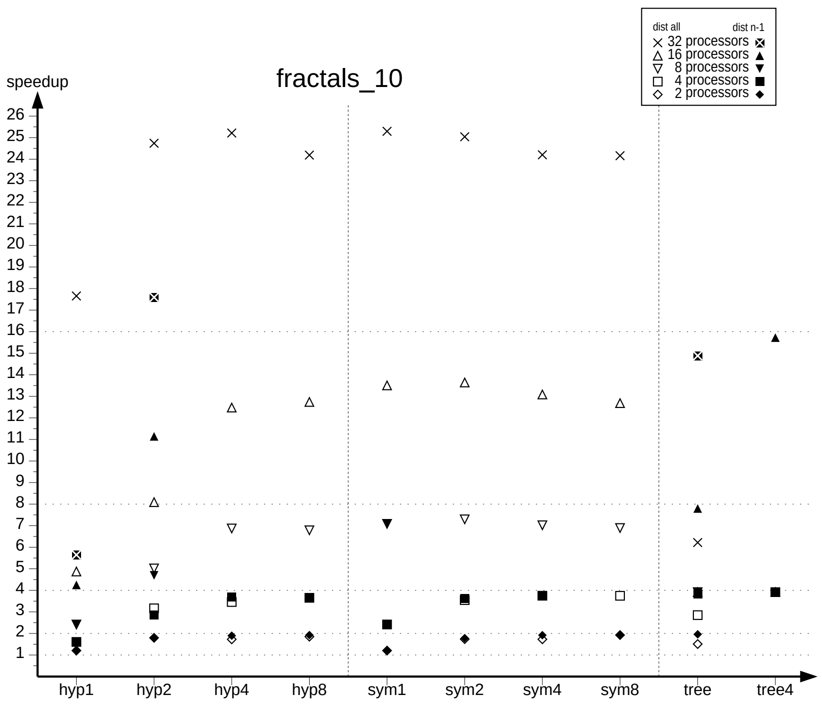

Fig. 5. Computing fractals with recursion depth 10

plication problems which unfold reasonably balanced trees, as for instance 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 `det` 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 runtimes that are too short in relation to the overhead of managing them. However, the same ends can be more easily achieved by adapting 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.

5 Conclusion

Implementing divide_and_conquer computations is only a first step towards a system concept which supports other forms of concurrent computations based on functional program specifications as well. Absolutely essential for this model are systems of cooperating functional processes which communicate via classical message passing mechanisms. Typical examples are (multigrid) relaxations for numerical solutions of PDEs which may require parameter-controlled recursive refinements of mesh sizes to compute critical parts, say of a fluid dynamics application, with higher resolution. However, there are also many ‘stand-alone’ applications for the divide_and_conquer scheme, e.g., in rule-based computations which typically unfold large search trees.

The performance figures presented in this paper are not yet too conclusive for two reasons. Firstly, we have so far investigated only small programs with fairly predictable behavior, balancing the dynamically evolving workload reasonably well (which is essential for good performance gains). More serious and complex application programs suitable for divide_and_conquer computations are currently under investigation. Secondly, all measurements are based on the interpretation of abstract machine code. The interpretation includes dynamic type checking (which typically accounts for 20 % of the total run-time) and reference counting for earliest possible release of unused heap space (which accounts for another 20 to 30 %). The speed of executing compiled nCUBE/2 machine code vs interpreting ASM code can be expected to improve by about an order of magnitude, while the overhead for process management and data communication remains about the same. Hence, the performance gains measured for our example programs may decrease considerably unless they are offset by decidedly larger problem sizes.

To put the absolute performance of the (sequential) ASM-interpreter into perspective, we did some comparative run-time measurements with implementations of the same algorithms in HASKELL[HJW+92], CLEAN[PvE93], and SISAL [Can93], using standard compilers[2,3,4] for these languages and a Sun SPARC 10/20 as the common system platform. Fairly representative for all examples are the run-time figures for the `fractal` program. The sequential ASM interpreter takes 675.1 seconds, the HASKELL code executes in 123.3 seconds, and the fastest implementations, CLEAN and SISAL , take 67.8 seconds and 52.1 seconds respectively. The SISAL code runs about as fast as an equivalent C program (51.8 seconds). Thus, ASM code interpretation is slower by roughly a factor of 10 to 12 than the code produced by the thoroughly optimized CLEAN and SISAL compilers.

Work on a compiler which translates ASM-code into C is currently in progress. A first non-optimized version which still includes dynamic type-checking and reference counting improves runtime performance by about a factor

[2] HASKELL Version 0.999.5 of Chalmers University
[3] CLEAN Version 0.84 of the University of Nijmegen
[4] SISAL Version 1.8 of Lawrence Livermore National Laboratory

of 5 relative to interpretation. A concurrent version will be implemented after all optimizations are done.

References

[AJ89] L. Augustsson and T. Johnsson: *Parallel Graphreduction with the $\langle \nu, G \rangle$-Machine.* In FPCA '89, London, 1989.

[Can93] D.C. Cann: *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, LLNL, Livermore California, 1993. part of the SISAL distribution.

[GH86] B. Goldberg and P. Hudak: *Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor.* In J.H. Fasel and R.M. Keller (Eds.): Graph Reduction, Sante Fé, LNCS, Vol. 279. Springer, 1986, pp. 94–113.

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

[HB93] M. Haines and W. Böhm: *Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of SISAL.* In A. Bode et al. (Eds.): PARLE '93, LNCS, Vol. 694. Springer, 1993, pp. 12–23.

[HJW$^+$92] P. Hudak, S. Peyton Jones, P. Wadler, et al.: *Report on the Programming Language Haskell.* Yale University, 1992.

[HR86] P.G. Harrison and M.J. Reeve: *The Parallel Graph Reduction Machine Alice.* In J.H. Fasel and R.M. Keller (Eds.): Graph Reduction, Santa Fé, LNCS, Vol. 279. Springer, 1986, pp. 94–113.

[JCSH87] S.L. Peyton Jones, C. Clack, J. Salkid, and M. Hardie: *GRIP - a High Performance Architecture for Parallel Graph Reduction.* In G. Kahn (Ed.): FPCA '87, Portland, Oregon, LNCS, Vol. 274. Springer, 1987, pp. 98–112.

[Klu83] W.E. Kluge: *Cooperating Reduction Machines.* IEEE Transactions on Computers, Vol. C-32, 1983, pp. 1002–1012.

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

[PvE93] R. Plasmeijer and M. van Eekelen: *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[SBK92] C. Schmittgen, H. Blödorn, and W.E. Kluge: π-RED* - *a Graph Reducer for Full-Fledged λ-Calculus.* New Generation Computing, Vol. 10(2), 1992, pp. 173–195.

[Sch92] J. Schepers: *Invariance Properties in Distributed Systems.* In L. Bougé (Ed.): CONPAR '92, LNCS, Vol. 634. Springer, 1992, pp. 145–156.

[SGH$^+$86] C. Schmittgen, A. Gerdts, J. Haumann, W. Kluge, and M. Woitass: *A System Supported Workload Balancing Scheme for Cooperating Reduction Machines.* In 19th Hawaii International Conference on System Sciences, Vol. I, 1986, pp. 67–77.

This article was processed using the LaTeX macro package with LLNCS style