Arndt Bode Mike Reeve Gottfried Wolf (Eds.)

PARLE '93 Parallel Architectures and Languages Europe

5th International PARLE Conference
Munich, Germany, June 14-17, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, FRG

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Arndt Bode
Institut für Informatik, TU München
Arcisstr. 21, D-80333 München, Germany

Mike Reeve
European Computer-Industry Research Centre (ECRC)
Arabellastraße 17, D-81925 München, Germany

Gottfried Wolf
Deutsche Forschungsanstalt für Luft- und Raumfahrt (DFLR)
Rudower Chaussee 5, D-12489 Berlin Adlershof, Germany

CR Subject Classification (1991): C.1-4, D.1, D.3-4, F.1.3

[illegible]

ISBN 3-540-56891-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-56891-3 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

PARLE is an international, European based conference which focuses on the parallel processing subdomain of informatics and information technology. Parallel processing is today recognized as an area of strategic significance throughout the world. As a result, many national, pan-European and world-wide initiatives are being planned or already exist to further research and development in this area.

Ever increasing demands are being made on computer technology to provide the processing power necessary to help understand and master the complexity of natural phenomena and engineering structures. Within human organizations ever more processing power is needed to master the increasing information flow. Many so-called "Grand Challenges" have been identified as being orders of magnitude beyond even the most powerful computers available today.

Although the microelectronics industry has made vast, impressive strides both in improving the processing power available from individual components and in dramatically reducing the cost of basic processing power, it is not in itself enough to satisfy even today's requirements.

Parallel processing technology offers a solution to this problem. By taking several basic processing devices and connecting them together the potential exists of achieving a performance of many times that of an individual device. However, it is still an important topic of research to discover how to do this optimally and then to be able to effectively exploit the potential power through real applications solving real-world problems. Some progress has been made, particularly in isolated applications, but building parallel application programs is today recognized as a highly complex activity requiring specialist skills and in-depth knowledge of both the application domain and the particular parallel computer to be used.

Many international conferences in the area of parallel processing focus on the now well-established technical areas broadly described as number-crunching. Although this area is also within PARLE's scope, it has tended to put more emphasis in its technical program on other areas, such as novel architectures, symbolic processing, parallel database technology,and functional and logic technology. These represent some of the most difficult challenges in making general purpose parallel computing a reality.

The PARLE Conference came into existence in 1987. It started its life as an initiative coming from the ESPRIT I programme and was financially supported by the Commission of the European Communities through that programme. Between 1987 and 1991 the conference was held biannually around Eindhoven in the Netherlands with Philips taking the responsibility for its organization.

In 1991 Philips decided that they no longer wished to continue organizing PARLE and so the future of the conference was reviewed. The conference Steering Committee members felt that PARLE had an important role to fulfil and so decided to continue, but with a revised format. PARLE is now focused on be-coming *the* European conference with an international reputation in the domain of parallel architectures and languages.

A new conference organizational format has been adopted to emphasize this new commitment in a number of ways:

- the conference will be held annually,
- PARLE will be held exclusively within Europe,
- the conference venue will change country each year,
- the Steering Committee will represent various European countries,
- the Steering Committee should contain some of the most eminent European workers,
- the Programme Committee will represent most European countries,
- the Programme Committee will contain some important non-European experts,
- the Programme Committee will include specialists from industry as well as academia.

This format was first used for the 1992 conference held in June in a suburb of Paris and organized through the French Informatics Society, AFCET. It was judged to be a success in terms of the number of paper submissions received, the increased level of attendance, and the improved technical quality.

PARLE intends to become the European forum for interchange between experts in the parallel processing domain. It is intended to attract both industrial and academic participants with a technical programme designed to provide a balance between theory and practice. This role is an important function of PARLE and a consequence of its history.

The ESPRIT programme was partly conceived as an umbrella for collaborations involving industrial and academic participants. By working together and exchanging ideas, an important synergy can occur which profits both communities and can lead to extremely fast exploitation of innovative solutions in the market place. This promotes mutual understanding of important issues and prevents technology transfer barriers.

PARLE reflects this by promoting exchange between industry and academia, between practitioners and theoreticians, especially within the European context but also within the rest of the international community involved in the field of parallel processing systems. These different roles represent a key component of the strength and importance of PARLE.

Within Europe, the ESPRIT programme represents a significant research effort involving industrial and academic workers in, amongst other topics, the design and implementation of new computer architectures, theoretical work, parallel language design and development, tools to support parallel application construction, and, of course, the construction of prototype parallel applications. Considering the history and roles of PARLE, it is natural that the conference is supported by the Commission of the European Communities through the ESPRIT Programme and through representation at the level of the Steering

Committee. The European nature has also been emphasised through the support of CEPIS, the Council of European Informatics Societies, which represents over 200 000 Information Technology professionals in Europe.

PARLE'93 was organized in Munich by the European Computer-Industry Research Centre, ECRC, in cooperation with the Technical University of Munich and SIEMENS Central Research Laboratories. The conference was sponsored by the ESPRIT Programme of the Commission of the European Communities, ECRC, the Dresdner Bank, the city of Munich, AFCET, CEPIS, GI and ITG.

More than 200 papers were submitted and the best 52 were accepted as full papers. Additionally, the proceedings include short summaries of the papers accepted for presentation at the poster session and brief overviews of some of the CEC ESPRIT projects that provided support for PARLE'93.

An industrial exhibition was organized as part of the PARLE'93 conference. This provided an excellent opportunity for all attendees to gain first-hand experience of the newest products available. Considering this was the first time such an event has been held as part of a PARLE conference, it is gratifying that so many major international companies chose to participate. PARLE'93 also featured tutorials covering advanced parallel processing techniques. These two things have undoubtedly added an important new dimension to PARLE and we hope that next year's organizers will continue with them.

The programme chairmen are grateful to the authors, the members of the programme and steering committees, the referees, the supporting societies, and the organizing committee for their help in preparing PARLE'93 and the proceedings. We would also like to thank the following for their efforts in ensuring the smooth running of the conference: Uli Fuetterer, Christiane and Susanne Hollmayer, Isabelle Syre, and Ulrich Koschkar and family.

April 1993 Arndt Bode, Mike Reeve, Gottfried Wolf

PARLE 93 Organization

PARLE 93 Steering Committee

Werner Damm (U. of Oldenburg, D)	Jean Claude Syre (Bull SA, F)
Jose Delgado (INESC, P)	Jorgen Staunstrup (TU Denmark, DK)
Lucio Grandinetti (U. of Calabria, I)	Mateo Valero (U. of Catalunya, E)
Constantin Halatsis (U. of Athens, GR)	Thierry Van der Pyl (DGXIII, CEC)
Ron Perrot (U. of Belfast, UK)	Pierre Wolper (U. of Liege, B)
Martin Rem (TU Eindhoven, NL)	

PARLE 93 Organizing Committee

Arndt Bode (T. U. Munich, D)	Joint Programme Chair
Werner Damm (U. of Oldenburg, D)	Steering Committee Liaison
Doug DeGroot (Texas Instruments / CSC, USA)	N. & S. American Co-ordinator
Ulrike Jendis (ECRC, D)	Treasurer
Peter Kacsuk (KFKI, H)	East European Co-ordinator
Masaru Kitsuregawa (U. of Tokyo, J)	Japan & Asian Co-ordinator
Rudi Kober (Siemens/ZFE, D)	Exhibition Chair
Michael Ratcliffe (ECRC, D)	Organizing Committee Chair
Mike Reeve (ECRC, D)	Joint Programme Chair
Gottfried Wolf (DLR, D)	Joint Programme Chair

PARLE 93 Programme Committee

Emile Aarts (Philips/Research, N)
Gul Agha (U. of Illinois, USA)
Khayri Ali (SICS, S)
Makoto Amamiya (Kyushu U., J)
Francoise Andre (IRISA, F)
Gianfranco Balbo (U. of Torino, I)
Bjorn Bergsten (Bull/RADO, F)
Maurelio Boari (U. of Bologna, I)
Kiril Boyanov (Bulgarian A. Sci., BU)
Andrzej Ciepielewski (Carlstedt, S
Michel Cosnard (ENS, F)
Felix Costa (INESC, P)
Jose Cunha (U. Nova de Lisboa, P)
Bill Dally (MIT, USA)
John Darlington (Imperial College, UK)
Doug DeGroot (TI/CSC, USA)
Josep Diaz (U. of Catalunya, E)
Daniel Etiemble (U. of Paris-Sud, F)
Geoffrey Fox (Syracuse U., USA)
Ivan Futo (Multilogic, H)
Guang Gao (McGill U., CDN)
Jean Luc Gaudiot (USC, USA)
David Gelernter (Yale, USA)
Wolfgang Gentzsch (GENIAS Soft., D)
Pascal Gribomont (I. Montefiore, B)
Jozef Gruska (U. of Hamburg, D)
Anoop Gupta (Stanford, USA)
John Gurd (U. of Manchester, UK)
Chris Hankin (Imperial College, UK)
Michalis Hatzopoulos (U. of Athens, GR)
Hiromu Hayashi (Fujitsu Labs., J)
Manuel Hermenegildo (U. of Madrid, E)
Tony Hey (U. of Southampton, UK)
Peter Hilbers (Shell/Research, NL)
Ladislav Hluchy (Slovak A. Sci., CS)
Chris Jesshope (U. of Surrey, UK)
Peter Kacsuk (KFKI, H)
Martin Kersten (CWI, NL)
Masaru Kitsuregawa (U. of Tokyo, J)
Rao Kotagiri (U. of Melbourne, AUS)
Simon Lavington (U. of Essex, UK)
Bernard Lecussan (U. of Toulouse, F)
Burkhard Monien (U. of Paderborn, D)
Peter Muller-Stoy (Siemens/ZFE, D)
Lee Naish (U. of Melbourne, AUS)
Flemming Nielson (U. of Aarhus, DK)
Wolfgang Paul (U. of Saarbrucken, D)
Emile Restivo (U. of Porto, P)
Leonardo Roncarlo (Elsag, I)
Dirk Roose (K. U. Leuven, B)
Paul Spirakis (U. of Patras, GR)
Kazuo Taki (Kobe U., J)
Hiroaki Terada (Osaka U., J)
Mario Tokoro (Keio U., J)
Roman Trobec (Inst. "J. Stefan", Slovenia)
Ulrich Trottenberg (GMD, D)
Marek Tudruj (Polish A. Sci., PL)
Emile Zapata (U. of Malaga, E)

PARLE 93 Sponsors

ESPRIT Programme, Commission of the European Communities
European Computer-Industry Research Centre (ECRC)
Dresdner Bank
Stadt München
AFCET
CEPIS
GI
ITG

Contents

ESPRIT Project Overvies

Simulation–based Comparison of Hash Functions for Emulated Shared Memory*

Curd Engelmann[1] and Jörg Keller[2]

[1] Universität des Saarlandes, Computer Science Department
Im Stadtwald, 6600 Saarbrücken, Germany
[2] Centrum voor Wiskunde en Informatica
Postbus 4079, 1009 AB Amsterdam, The Netherlands

Abstract. The influence of several hash functions on the distribution of a shared address space onto p distributed memory modules is compared by simulations. Both synthetic workloads and address traces of applications are investigated. It turns out that on all workloads linear hash functions, although proven to be asymptotically worse, perform better than theoretically optimal polynomials of degree $O(\log p)$. The latter are also worse than hash functions that use boolean matrices. The performance measurements are done by an expected worst case analysis. Thus linear hash functions provide an efficient and easy to implement way to emulate shared memory.

1 Introduction

Users of parallel machines more and more tend to program with the view of a global shared memory. Commercial machines (with more than 16 processors) however usually have distributed memory modules. Therefore the address space has to be mapped onto memory modules, memory access is simulated by packet routing on a network connecting processors and memory modules. This has to be done in a way that for (almost) all access patterns the requests are distributed almost evenly among the memory modules. The reason to demand this is obvious: if cases happen where the number of requests per module (the so called *module congestion*) is too high, then performance gets very poor.

Several kinds of hash functions have been proposed. But their theoretically provable properties are asymptotical results. As currently available machines are quite small (the number p of processors and memory modules usually is less than 1000) the actual behaviour of the chosen hash function can differ quite a lot from these theoretical properties. The lack of experimental data makes the selection of a particular hashing scheme difficult in practice. We are not aware of comparisons of hash functions based on simulated behaviour.

* This work was supported by the German Science Foundation (DFG) in SFB 124, TP D4, and by the Dutch Science Foundation (NWO) through NFI Project ALADDIN under Contract number NF 62-376. Part of this work was done while the second author was working at Universität des Saarlandes, Computer Science Department, Saarbrücken, Germany.

The goal of this investigation is to provide these data by comparing four kinds of hash functions by simulations. In Sect. 2 the most common kinds of hash functions are introduced. Section 3 describes the types of synthetic and real access patterns that were used as workloads. Section 4 sketches the experiments made and Sect. 5 presents and discusses the results.

2 Hash Functions

As already mentioned, a hash function serves to map a global address space onto distributed memory modules. More formally, for an address space M of size $m = 2^{v}$ and a set N of $p = 2^{n}$ memory modules, the mapping is a function $h : M \to M$ that maps addresses to memory cells. The function $mod : M \to N, mod(x) = x \text{ div } m/p$ specifies the module of a memory cell x, the function $loc : M \to M', loc(x) = x \bmod m/p$ specifies the local address of cell x.

An optimal mapping function h should guarantee low module congestion for almost all possible access patterns (if all addresses of one pattern are distinct). This is achieved by using classes of functions in which each function has low module congestion for almost all patterns. A particular function is randomly chosen. This guarantees with very high probability that the current application does not exhibit the patterns on which the chosen function produces hot spots.

An additional problem consists in patterns with several processors concurrently accessing one cell. This problem cannot be solved by hashing. However there exist routing algorithms that perform *combining*. Requests that access the same cell are merged during routing, answers are duplicated. Ranade's emulation algorithm [10] is a good example. Therefore, concurrent access does not increase module congestion.

A class that restricts module congestion to $O(\log p)$ is

$$\mathcal{H} = \left\{ p(x) = \left(\sum_{i=0}^{\xi} a_i \cdot x^i \right) \bmod P \bmod m : 0 \leq a_i < P \right\} .$$

P is a prime larger than m, $\xi = O(\log p)$. A function of $\mathcal{H}$ is obtained by randomly choosing the values for a_i. This class was used in several theoretical investigations [6, 8, 10] to emulate shared memory on a processor network. The module congestion of $O(\log p)$ is sufficient because access from processors to memory modules across a constant-degree interconnection network needs time $\Omega(\log p)$ anyway.

However the functions in $\mathcal{H}$ are not bijective. This means that several addresses of the shared memory could be mapped onto the same cell. This requires secondary hashing on each memory module. Ranade [10] describes a method that performs secondary hashing in constant time and increases the size of the memory module only by a constant factor.

In practice however one should avoid secondary hashing and waste of memory because a constant factor of performance loss can destroy an asymptotically good result. Furthermore, the time to evaluate the hash function should be short. The functions in $\mathcal{H}$ require $\xi = O(\log p)$ multiplications and additions and a modulo division by a prime which needs a lengthy computation.

Therefore some alternatives were proposed:

1. For $\xi = 1$ one obtains a linear function. This reduces evaluation time to one multiplication, one addition and one modulo division. The function is still not bijective.
2. Furthermore if the modulo division by a prime is skipped and the coefficient a_0 is set to zero, the evaluation time is reduced to one multiplication. The operation modulo m is not counted because m is a power of two. If only odd values are chosen for a_1 the function also is bijective.
3. If the binary representation of an address is seen as a boolean vector, the hash function consists of multiplying this vector with an invertible boolean matrix. The time to evaluate this function is shorter than one multiplication.

Dietzfelbinger et. al. prove that the first alternative is asymptotically equivalent to the second [5]. Furthermore he proves that linear functions can result in a module congestion of $\Theta(\sqrt{p})$ for patterns with addresses of the form $b + s \cdot i$ where $i = 0, \ldots, n-1$ [4]. The constants b and s are called *base* and *stride*. This means that linear functions modulo a power of two are asymptotically worse than polynomials.

The third alternative was used in the design of the IBM RP3. Norton and Melton [9] introduce a class of boolean matrices where all matrices are invertible (which means bijectivity). Optimal distribution can be guaranteed for patterns with strides where s is a power of two and where in the binary representation of base b bits s to $s + \log n - 1$ are zero. For other bases the module congestion is at most 2. No theoretical results are given for other patterns, but their simulations hint that distribution is acceptable for other patterns, too. One particular matrix is obtained by randomly choosing several bits of the matrix and then computing all the other bits with respect to the above properties.

3 Workloads

The workloads are chosen to compare the hash functions with respect to known differences, especially behaviour on access patterns with strides, and with respect to patterns taken from applications. Therefore both synthetically generated patterns and application traces were taken.

The synthetic traces consist of randomly chosen patterns as a reference and strides with $s = 1, 13, 32$. The strides were chosen to compare matrix hashing and the other hash functions and to check whether linear functions get worse on these patterns. For $s = 32$ and $s = 1$ matrix hashing is optimal [9]. Theoretical results about the performance on the others are not known.

The traces were taken from three application programs: list ranking, matrix multiplication and connected components. The reasons for taking traces from applications are the variety of produced patterns and the structure of single patterns that often is more complex and less regular than in synthetical traces.

The three applications are chosen to represent a large variety of algorithms. Matrix multiplication is an example of a class of algorithms where the access patterns are regular and do not depend on the particular input values. Many other numerical algorithms behave that way, especially as many of them are originally designed to work on a processor network with a fixed interconnection structure (see e.g. [3]).

List ranking represents combinatorial algorithms where access patterns depend on the actual data. An example technique is pointer doubling. Processor i loads or stores $F[F[i]]$, where F is an array in the shared memory. Part of the accesses to shared memory still are regular. If processor i loads or stores $F[i]$, the access pattern is a stride with $s = 1$. Many PRAM algorithms working on lists and graphs are of this type (see e.g. [7]).

The connected components algorithm represents algorithms where access patterns depend on the actual data, but not all processors may participate in the access. This together with concurrent accesses to some cells, which get combined, makes module congestion smaller. Thus, connected components and similar algorithms are remarkable exceptions compared to list ranking type algorithms.

The list ranking algorithm is taken from a survey [7]. For a given linked list of n elements, the distance (or *rank*) to the end of the list is computed for each element. The algorithm needs n processors and $O(\log n)$ time. The list is represented as an array F, where $F[i]$ means successor of i in the list. For the last element of the list, $F[i] = i$. The rank is contained in array R. The PARDO code is shown in Fig. 1(a).The access patterns of this algorithm partly depend on the structure of the list and partly are strides with $s = 1$.

In the matrix multiplication algorithm $C = A \cdot B$, each processor computes one element of the destination matrix C. In order to avoid concurrent accesses, all processors start at different rows and columns of the matrices A and B. The PARDO code is shown in Fig. 1(b).Matrices A and C consist of $n = w2^{2z}$ elements and have dimension $2^z \times w2^z$, matrix B has dimension $w2^z \times w2^z$. The algorithm needs n processors and takes time $O(n^{1/2})$. The access patterns of this algorithm only depend on the dimensions of the matrices.

The connected components algorithm was adapted from Shiloach and Vishkin [11]. For a given undirected graph $G = (V, E)$, the connected components are computed. The algorithm needs $n = \max(|V|, 2|E|)$ processors and takes time $O(\log n)$. The graph is represented by two arrays HEAD and TAIL. For a given edge e, HEAD[e] and TAIL[e] contain the nodes to which e is adjacent. The components are represented by an array F. Two nodes u, v are in the same component if and only if $F[u] = F[v]$ after running the program. The PARDO code is shown in Fig. 1(c).The access patterns partly depend on the structure of the input graph and partly are strides with $s = 1$. Not all processors participate in every access.

```
(* Init rank $R$ *)
for $i := 1$ to $n$ pardo
  if $F[i] = i$ then $R[i] := 0$ else $R[i] := 1$
od ;
(* Compute rank $R$ *)
for $t := 1$ to $\lceil \log n \rceil$ do
  for $i := 1$ to $n$ pardo
    $R[i] := R[i] + R[F[i]]$ ;
    $F[i] := F[F[i]]$ (* Pointer doubling *)
  od
od ;
```

(a) list ranking

```
(* $n = w2^{2z}$ *)
$k := 2^z$; $m := w2^z$; $l := w2^z$;
for $(i, j) := (1, 1)$ to $(k, m)$ pardo
  $C[i, j] := 0$ (* Init $C$ *)
od ;
for $r := 1$ to $l$ do
  for $(i, j) := (1, 1)$ to $(k, m)$ pardo
    $t := (i + j + r) \bmod l$ ;
    $C[i, j] := C[i, j] + A[i, t] \cdot B[t, j]$
  od
od ;
```

(b) matrix multiplication

```
for $u \in V$ pardo $F[u] := u$ od;
for $t := 1$ to $2 \log |V|$ do
  for $u \in V$ pardo $change[u] := 0$ od;
  starcheck;
  for all $(u, w)$ with $\{u, w\} \in E$ pardo
    if $star[u]$ and $F[w] < F[u]$ then
      $F[F[u]] := F[w]$;
      $change[F[u]] := 1$;
      $change[F[w]] := 1$
    fi
  od;
  starcheck;
  for all $(u, w)$ with $\{u, w\} \in E$ pardo
    if $star[u]$ and not $change[F[u]]$
                and $F[w] \neq F[u]$ then
      $F[F[u]] := F[w]$
    fi;
    $F[u] := F[F[u]]$
  od
od.

proc starcheck ;
begin
  for $i \in V$ pardo
    $star[i]$ := true;
    if $F[F[i]] \neq F[i]$ then
      $star[F[F[i]]]$ := false
    fi;
    $star[i] := star[F[F[i]]]$
  od
end;
```

(c) connected components

Fig. 1. Code of applications

4 Experiments

To obtain the input data for the experiments, all applications are simulated by sequential programs, only the address traces are extracted. This frees us from considering a particular microprocessor instruction set and compiler. The address traces of the synthetic workloads are generated by a program, that simulates 4 steps of the machine. In the workloads with strides, the base b is increased each step by ns.

We are only interested in the resulting module congestion and not in the time to route the requesting packets from processors to memory modules. Therefore we can neglect the structure of the interconnection network. We only model it by a latency term because the processors perform latency hiding (see below).

All experiments are carried out for $m = 2^{22}$, the prime P is chosen closest to m. We simulate machines with $p = 2^u$, $u = 5, \ldots, 10$ processors. We run multiple processes per processor to hide the network latency from processors. The processes are executed in a round-robin manner, one instruction per turn. The exact number c of necessary processes per processor is depending on p, e.g. $O(\log p)$ in a butterfly network. We choose a fixed c to obtain comparable results and take $c = 5$ as an average from a machine size of $p = 128$ [2]. Therefore in each step $5p$ requests are made. Step in this context means synchronous execution of one instruction on each of the $5p$ processes.

As polynomials we used functions of degree $\xi = 2, 10, 20$. Each of the experiments was done 5 times with randomly chosen hash functions. More exactly, for each class five functions were randomly chosen and then used for all workloads and machine sizes.

As input for list ranking a list of length $n = 10p$ was randomly chosen. As input for connected components, a graph with $n = 10p$ nodes and $5p$ edges was randomly chosen. The problem size n is twice as large as the number of processes in these applications. Each process simulates two program processors step by step. A problem size larger than $5p$ is needed to obtain access patterns depending on the list or graph.

In matrix multiplication, the dimensions of the matrices are as follows: if $p = 2^{2z}$ then $w = c = 5$, if $p = 2^{2z+1}$ then $w = 2c = 10$.

In each experiment we measured for each step of the trace the maximum module congestion c_{max} and then computed the expected value of all c_{max} averaged over all steps. The analysis is a kind of (expected) worst case analysis. Each expected value was checked for significance by looking at the variance. The five values obtained by using five functions of one class for each experiment were checked against significant differences. In case there were none, the average was taken. In case there were some, ten additional hash functions were chosen and the average was taken from these 15 values. Significant differences appeared only for stride $s = 13$, $p = 2^7, \ldots, 2^9$ in both linear functions and for stride $s = 32$, $p = 2^9, 2^{10}$ in the linear function modulo power of two.

Because of mapping $5p$ requests per step onto p memory modules, $E(c_{max}) \geq 5$. The only exception is connected components, because not necessarily all processors make accesses in IF statements (see Sect. 3).

5 Results

The results of the experiments are presented in two ways. First we show the performance of the hash functions sorted by benchmarks. In Fig. 2 the performance on random patterns is given as a reference. The legend of the hash functions is shown in Fig. 3, which shows all other benchmarks. Second we show the performance sorted by hash functions in Fig. 4.

All figures are built as follows: the x-axis shows $\log p$ in range $5 \ldots 10$, the y-axis shows the expected value of the maximum module congestions in range $4 \ldots 14$.

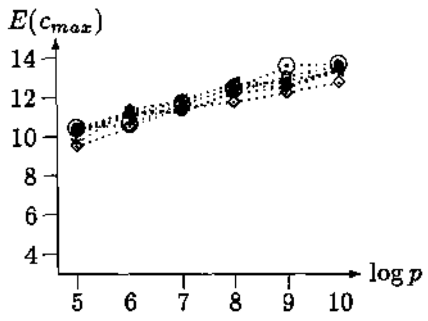

Fig. 2. Performance on random patterns

The performance on random patterns (see Fig. 2) is similar for all hash functions. Thus none of the hash functions is bad in an obvious way. The maximum module congestion rises from 10 for $p = 32$ to 12 for $p = 1024$. This will serve as a reference to analyse the performance on the other benchmarks.

5.1 Analysis of Benchmarks

The curves of Fig. 3 show similar shapes for all benchmarks: the polynomials of different degrees behave in a similar way and so do the three other hash functions. The behaviour of the polynomials furthermore is on all workloads worse than the behaviour of the simpler hash functions. Among the linear functions, the one modulo a prime always behaves a little bit worse than the linear function modulo a power of two. Thus the most interesting part is the comparison of our simple linear function with the boolean matrix hashing.

For strides that are a power of two, the boolean matrix hashes values optimally (see (a) and (c)) and reaches a module congestion of 6. The module congestion reached by the linear function lies between 6.5 and 7.5, so it is not far away.

A similar behaviour of linear function and boolean matrix can be seen in (d) and (f). This results from the fact that part of the accesses in these workloads are strides 1, when processors load or store values in arrays in the manner that processor i reads or writes $F[i]$.

However, as soon as we obtain other patterns, the boolean matrix hashing gets worse than the linear function (see (b) and (e)). Even for the matrix multiplication workload, where accesses always consist of $5 \cdot p^{1/2}$ strides with $s = 1$ and $p^{1/2}$ processors involved in each stride, the linear function is better.

5.2 Analysis of Hash Functions

Figure 4 shows the performance of the different hash functions. Because the connected components benchmark is not comparable to the others as explained in Sect. 3, it is not shown here. The first observation is, that all hash functions behave on all workloads not worse than on random patterns. The second observation is that the polynomials show roughly the same behaviour on all workloads as they do on random patterns (see (d) to (f)). We conclude that their performance is independent of the application. That is what we expected. But this performance is bad in comparison to what is reached by the other functions that behave better than on random patterns on all workloads.

The linear function (see (a)) shows almost uniform behaviour on all workloads, too, but it varies between 6.5 and 8, which is significantly better than the behaviour on random patterns.

The behaviour of the linear function modulo a prime is not uniform and varies between 6.3 and 10.

The behaviour of the boolean matrix hashing function can be divided in an expected optimal behaviour for strides with s a power of two and a significantly higher module congestion for other patterns, which is however still below the one produced by random patterns.

6 Conclusions

The above experiments show surprisingly that linear functions modulo a power of two and boolean matrix functions show best performance for practical use. Both have the additional properties of bijectivity and short evaluation time. The choice between these two depends on the expected user profile (if such exists) and the surrounding machine architecture. For machines that already contain a hardware multiplier this could be used to perform hashing in the case of the linear functions. Moreover, the use of matrix hashing is restricted by the fact that an implementation needs $(\log m)^2$ bit register hardware to store the boolean matrix. Therefore, if no user profile is known and chip area is restricted (or a multiplier already available), the use of the linear function is preferable.

The observations presented here lead to the use of linear hash functions in the prototype design of the SB–PRAM [1, 2] which emulates a synchronous shared memory machine with $p = 128$ physical processors and provides hardware support for hashing and packet routing including combining.

Unfortunately, some open questions remain. First, there is no theoretical framework to explain why simple hash functions work better than complex ones. Also, the exact relationship between linear functions with and without "modulo prime" is still unknown.

Acknowledgements

The authors would like to thank Martin Dietzfelbinger for helpful discussions.

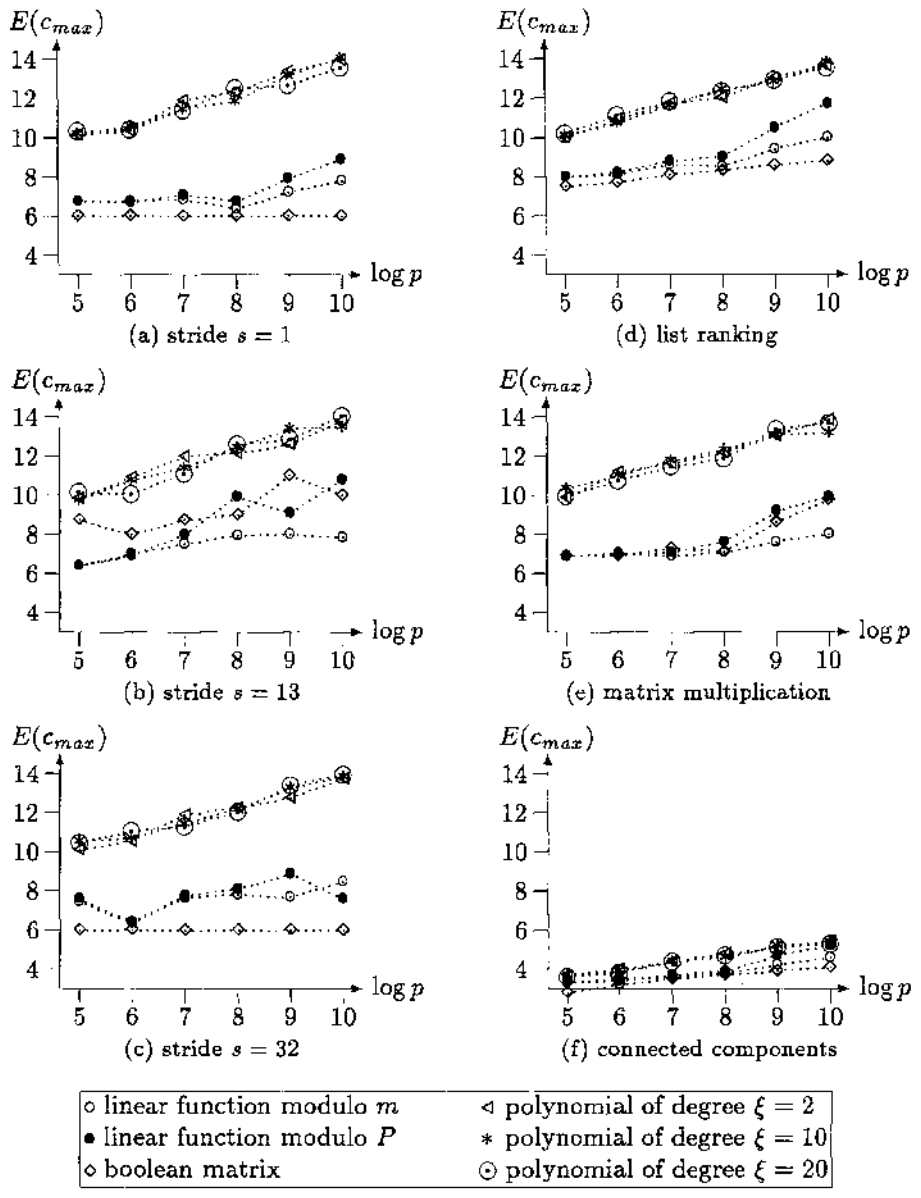

Fig. 3. Performance on benchmarks

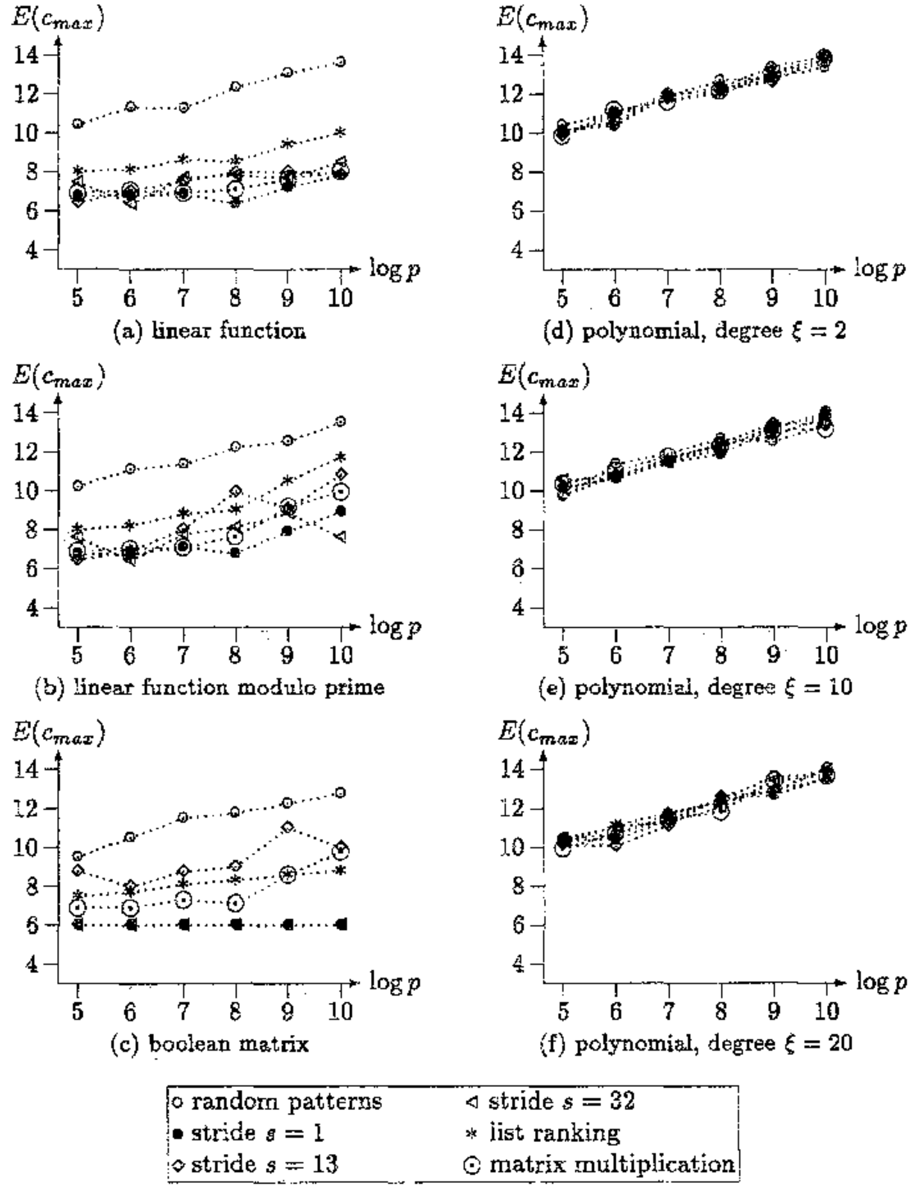

Fig. 4. Performance of hash functions