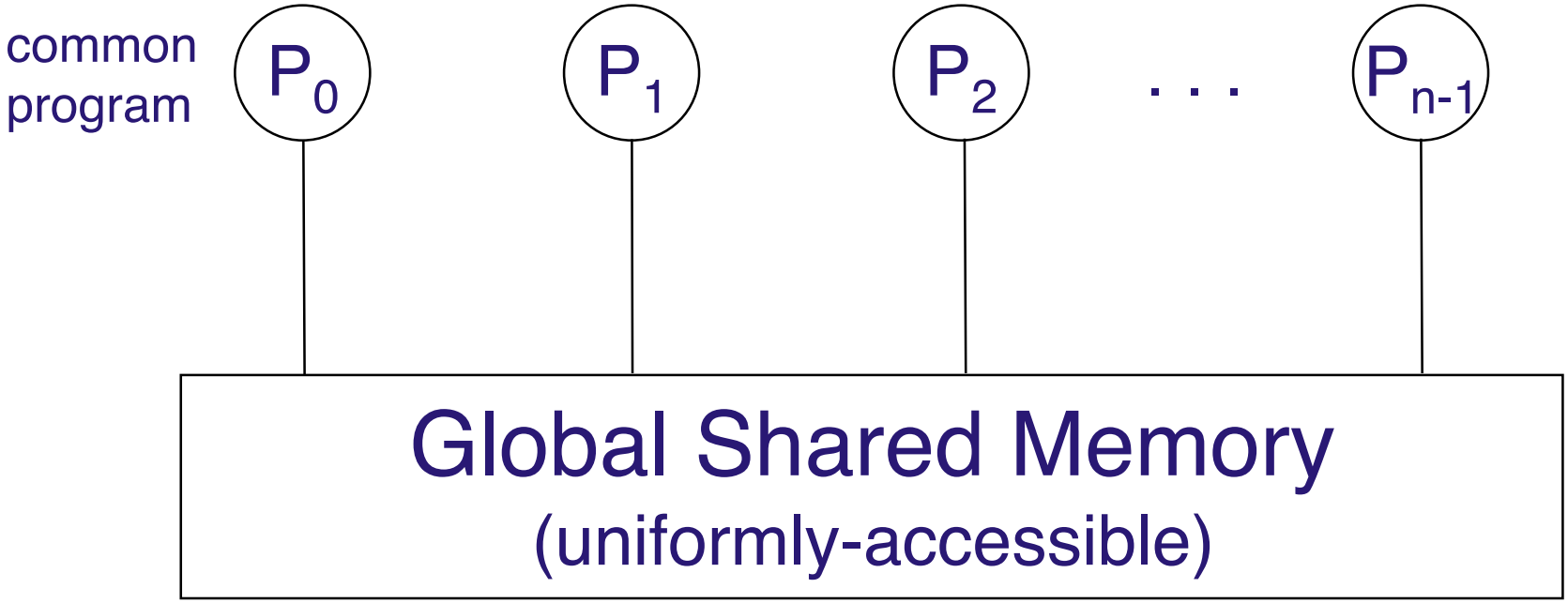
PRAM Model Introduction

PRAM Model

- PRAM = Parallel, Random-Access Machine
- **Idealized model** introduced in 1978 by R.Cole,
  based on theoretical RAM model
- Unbounded number of processors, to fit problem
- Shared common memory
  + local memories per processor
- Processors operate synchronously,
  could be loosened to SPMD with synchronization routines
- Writing to common memory is **synchronous**

PRAM Diagram

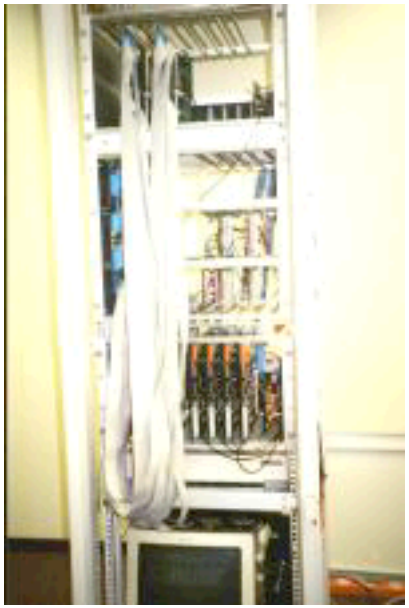
Processors (with local dedicated memory)
n adjusted to problem size

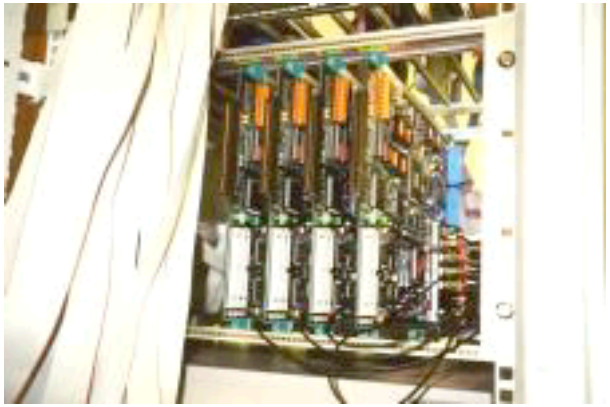


Use of PRAM Model

- Simple and elegant for some problems
- Can tell us certain things about structuring, especially for synchronous computation
- Can be simulated on parallel machines (e.g. by rescheduling, Brent's lemma, etc.)
- At least one was being constructed

SB-PRAM constructed at Universität des Saarlandes Inst. of Parallel Computing





The project goal was to achieve a 64 physical (2048 virtual) processor machine with 2 GByte of global memory and 256 hard disks.

Current Status of SB_PRAM

Project overview

The SB-PRAM is a MIMD parallel computer with shared address space and uniform memory access time (CRCW-PRAM-Model). Processors and memory modules are connected by a butterfly network. Each SB-PRAM processor module consists of a custom ASIC processor with extended Berkeley-RISC instruction set, a local program memory and SCSI interface. Network nodes and memory modules provide hardware support for concurrent read and concurrent write memory access and parallel prefix operations. Network latency is hidden by pipelining several virtual processors (hardware threads with zero switching overhead) on one physical processor. Network congestion is reduced by hashed addresses. Hot spots are avoided by combining.

Project status

We have succeeded in building a 64 physical (2048 virtual) processor machine with 4 GByte of global memory. The machine is currently switched off.

Project members

Prof. Dr. Wolfgang J. Paul

Memory-Conflicts

- All processors can read or write to distinct shared memory locations in one time step.
- What if two processors try to **read** from the same memory location in the same time step?
- What if two processors try to **write** to the same memory location in the same time step?

PRAM Varieties Based on Memory-Conflict Models

- Generally concurrent reading and writing to a single location is disallowed.
- **EREW** (**E**xclusive-**R**ead, **E**xclusive-**W**rite) Concurrent reading or writing to a location is disallowed.
- **CREW** (**C**oncurrent-**R**ead, **E**xclusive-**W**rite) Concurrent writing to a location is disallowed.
- **CRCW** (**C**oncurrent-**R**ead, **C**oncurrent-**W**rite) Concurrent writing to a location is allowed.

Sub-varieties of CRCW (1) indicate how conflict is resolved

- CRCW-**Common**: Concurrent writing is allowed only if it is known that all processors will be writing the same value (writing **no** value is always an option).

- CRCW-**Arbitrary**: If multiple processors attempt to write, one will be chosen arbitrarily as the winner and the others ignored.

Sub-varieties of CRCW (2) indicate how conflict is resolved

- CRCW-**Priority**: If multiple processors attempt to write, the highest-priority will be chosen as the winner and the others ignored.
- CRCW-**Sum**: If multiple processors attempt to write, the values will be summed and the sum written instead.
- **Combining**: Variants on Sum: Any binary operator (or, and, xor, min, max, product, …)

Why does it matter?

- To physically realize any approximation to a PRAM requires an understanding of the memory conflict model.
- There is a time cost to resolving memory conflicts, which varies depending on the model.

PRAM Preferences

- It is preferable to use as little machinery as possible for algorithms.
- Therefore, prefer
  - CREW over CRCW
  - CRCW-arbitrary over CRCW-common
  - CRCW-common over CRCW-sum
  - etc.

PRAM Algorithm Examples

- Computing max of n numbers:
  - log n time on EREW (and by implication CREW, CRCW, …)
  - Assume the numbers are in shared memory locations 0, 1, …, n-1.
  - Even numbered processors fetch “their” numbers to their local memory (other processors are idle).
  - Even numbered processes fetch “their neighbors” numbers to their local memory.
  - Even numbered processors write the max of the two numbers to “their” locations.
  - Repeat with processors divisible by 4, 8, 16, ...

PRAM Max

Essentially we have a subtree of the prefix-sum tree (using max instead of add).

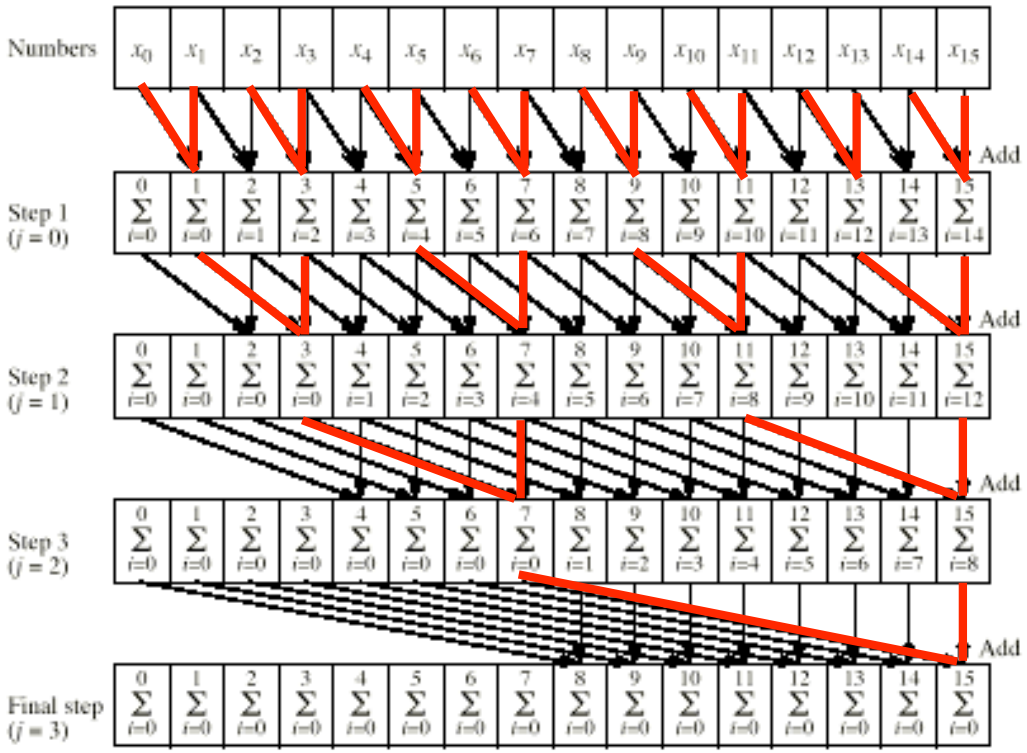


Figure 6.8 Data parallel prefix sum operation.

PRAM Prefix Sum

- Obviously an EREW PRAM can compute **any prefix-sum** type computation in O(log n).
- More processors are busy here than in the max case.

Better (?) ways to do max

- Intuitively $\Omega(\log n)$ seems like a lower bound on the max computation of n numbers.
- However, a CRCW-arbitrary PRAM can do better.

CRCW-arbitrary max computation

- O(1)
- using n^2 processors

CRCW-arbitrary max computation setup

- Let the data be in shared memory locations x[0], …, x[n-1].
- Use n bit locations: b[0], …, b[n-1], all set to 1 (in one step).
- b[i] is associated with x[i].

CRCW-arbitrary max computation

- The meaning is that, at the end of the computation, b[i] will be 0 iff x[i] is less than **some** x[j] where j != i.
- So elements x[i] where b[i] == 1 will be the max.
- In three steps: n*(n-1)/2 processors each
  - fetch, then
  - compare a different x[i] with an x[j].
  - If x[i] < x[j], the processor sets b[i] to 0, and vice-versa.

CRCW-arbitrary max computation

- Each processor either writes 0 or does nothing.
- If two processors write to the same location, they will both be writing the same thing.
- Therefore the CRCW-arbitrary assumption is honored.

What happened to $\Omega(\log n)$?

- In an implementation of CRCW-common, it isn't physically realizable to have an **arbitrary** number of processors write to the same location at once, even if they do write the same value.

- We have replaced what would have been binary ops with a single op of arbitrary arity.

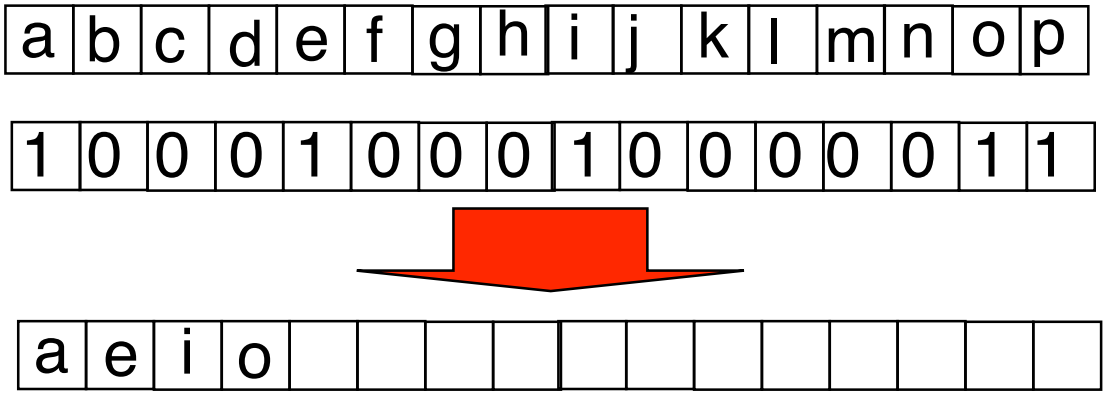
- We could implement this op as a **fan-in tree**, which would recover the $\Omega(\log n)$.
  [$O(n^2)$ processors fanning in, $\log(n^2) = O(\log n)$].

Simulation Theorem (see Cormen, et al., p706-708)

- Any CRCW-common PRAM algorithm using p processors can be simulated by an EREW PRAM with a **slowdown** factor of **log(p)**.

Array Compression

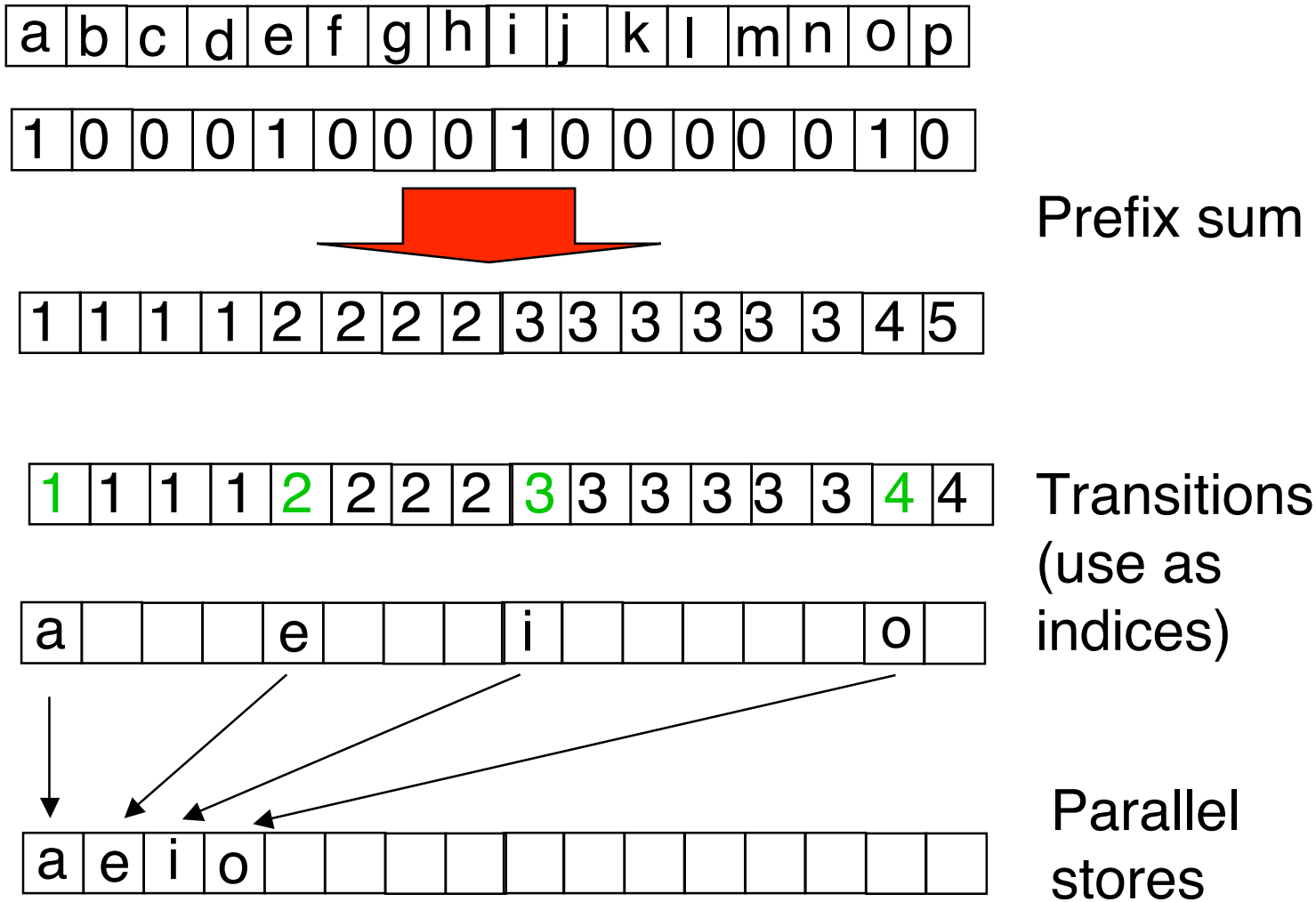
- Problem:
  Given an array in shared memory and a bit vector indicating the elements to be compressed, create an array containing only those elements **contiguously**.



Technique

- Use prefix-sum + indexing (parallel).
- Compute the prefix sum of the bit array.
- Use the computed values as **indexes** of where to store the corresponding item.
- Only use the index at transitional values (to avoid the need to use CRCW).

Array Compression Exposed



Exercise

- How would you do array **expansion**: distribute array elements according to a bit vector?

References

- Selim Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997.
- Selim Akl, The Design of Efficient Parallel Algorithms, Chapter 2 in "Handbook on Parallel and Distributed Processing" edited by J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, Springer Verlag, 2000.
- Selim Akl, Design & Analysis of Parallel Algorithms, Prentice Hall, 1989.
- Cormen, Leisterson, and Rivest, Introduction to Algorithms, 1st edition (i.e., older), 1990, McGraw Hill and MIT Press, Chapter 30 on parallel algorithms.
- Phillip Gibbons, Asynchronous PRAM Algorithms, Ch 22 in Synthesis of Parallel Algorithms, edited by John Reif, Morgan Kaufmann Publishers, 1993.
- Joseph JaJa, An Introduction to Parallel Algorithms, Addison Wesley, 1992.
- Michael Quinn, Parallel Computing: Theory and Practice, McGraw Hill, 1994.