Lecture 3 – Emulation of PRAMs; SB-PRAM architecture

SB-PRAM overview
Hashing shared memory addresses
Parallel prefix computation on a tree
Multiprefix computation
Ranade's emulation algorithm
SB-PRAM architecture
SB-PRAM system software tools and simulator
Exact barrier implementation in SB-PRAM assembler

SB-PRAM: A realization of the PRAM model in hardware

based on "Fluent Machine" emulation approach [Ranade et al. '87, '88]
cost-efficient, scalable [Abolhassan/Keller/Paul'90, '91]

F physical processors

multithreading
each physical processor simulates $V = c \log_2 F$ PRAM processors (vPs)

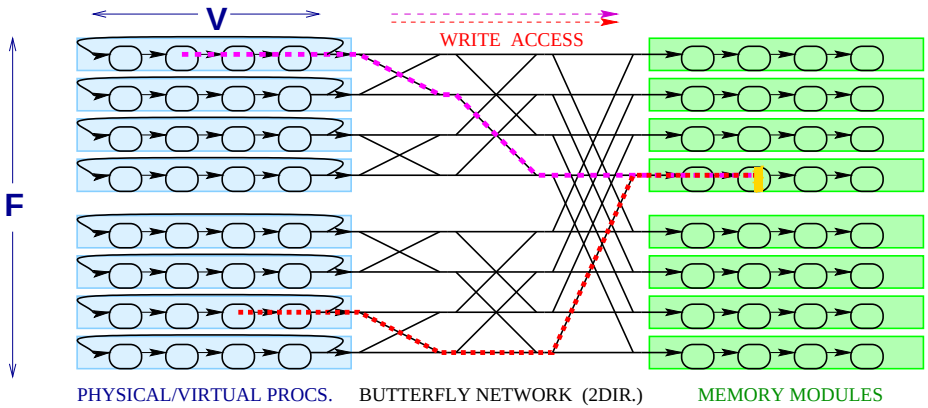
pipelined butterfly network
$F \log_2 F$ switches with simple ALU and memory

$P = F * V$ memory modules;
write conflicts resolved by combining

on-the-fly parallel reductions and multiprefix in the network

university prototypes:
(1) $F = 16$, $V = 32$ (finished 1998) (2) $F = 64$, $V = 32$ (finished 2000)
1991 ASIC design, 8 Mhz $\rightarrow$ 250 kFLOPs (also memory bandw.) per vP

SB-PRAM: A realization of the PRAM model in hardware (cont.)



Concurrent accesses to the same memory location:
requests meet at a butterfly switch
combine requests (depending on request type and priority)
for reading requests: on the way back, split up replies again

Special End-of-Step packets mark end of one PRAM step

Distributed shared memory by hashing of addresses

Map m shared memory addresses [PPP 3.4]
over p disjoint memory modules of size $m' = m/p$ each:

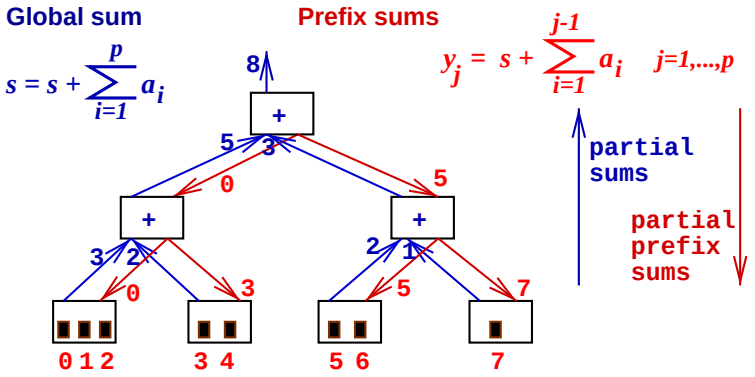
Hash functions
$h_1 : \{0,...,m-1\} \rightarrow \{0,...,p-1\}$ gives the module address
$h_2 : \{0,...,m-1\} \rightarrow \{0,...,m'-1\}$ gives the local address

Bad access sequence
a lot of requests go to the same module (not same location)
$\rightarrow$ overloaded, maybe request queue overflow

Prob(access sequence bad) is very low $\rightarrow$ choose a random hash function
$h_1(x) = \left(\left(\sum_{i=0}^{\zeta-1} a_i x^i \right) \bmod P \right) \bmod p$
with a_i randomly chosen, P an appropriate prime (see [PPP 3.4]

On SB-PRAM: linear hash function $h_1(x) = a \cdot x \bmod p$, default: $a = 1$ [EK93]

Parallel prefix on the SB-PRAM



Parallel prefix "on" shared memory location s

virtually, by adding up all contributions a_i to s in sequential

can be implemented using a binary tree rooted at s

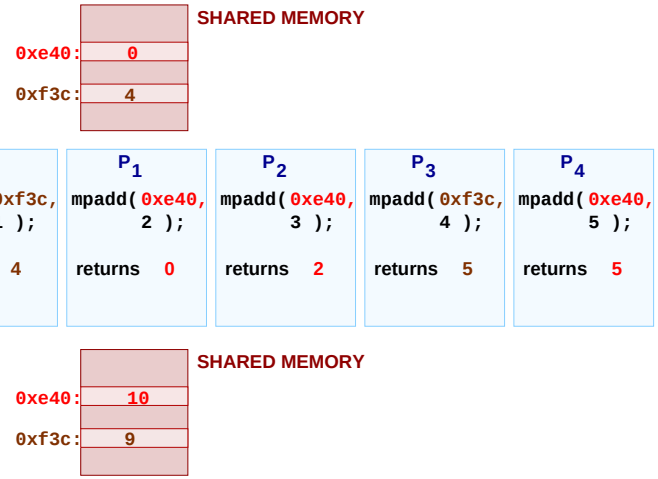
(slightly suboptimal variant of odd-even prefix)

This binary tree is embedded in the SB-PRAM network

(request paths towards memory module hosting s).

Multiprefix on the SB-PRAM

Multiprefix: Multiple parallel prefix operations on different shared memory locations s_1, s_2, ... can be processed simultaneously.



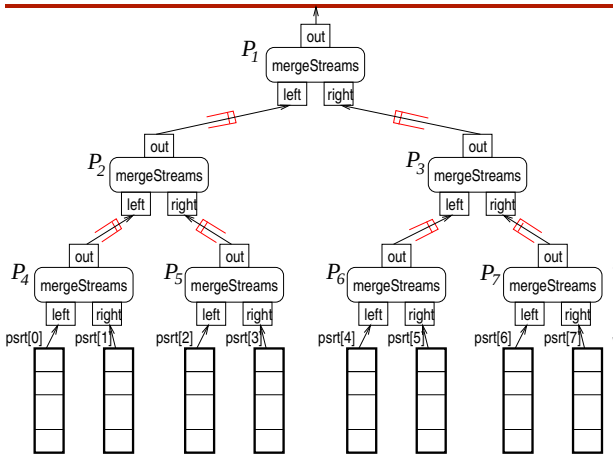
On SB-PRAM: for integer addition, maximum, bitwise AND, bitwise OR

Ranade's simulation algorithm

[Ranade'87,'88,'91]

+ very simple
+ requires only constant sized queues to store access requests
+ slowdown factor only $O(\log p)$
+ can be made optimally efficient
+ randomization employed only to distribute the shared memory across the memory modules.
+ pipelined butterfly network as the underlying communication network $\rightarrow$ scalable
+ multiprefix on-the-fly [Ranade et al.'88]
+ cost-efficient implementation: SB-PRAM [Abolhassan/Keller/Paul'91]

Prerequisite: pipelined merging of sorted access sequences



simpler case:
1 memory module

Combining:

Tree of $p-1$ merge processors working synchronously in parallel

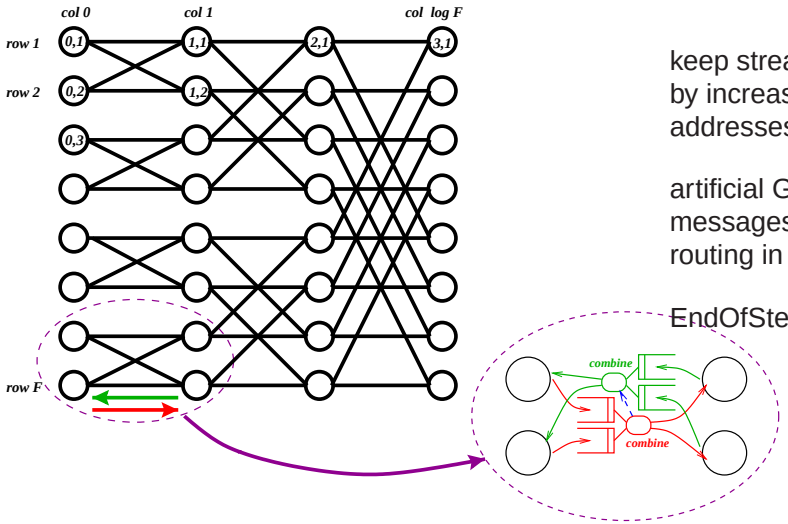
Initial arrays (addresses of memory requests) sorted in increasing order.

Per global step, each comparator moves the larger operand upwards, the other waits $\rightarrow$ FIFO queues along edges needed.

Combine requests with identical addresses.

Fluent Abstract Machine

Arrange $p = F(\log F + 1)$ PRAM processors
on a $F \times (\log F + 1)$ bidirectional butterfly network

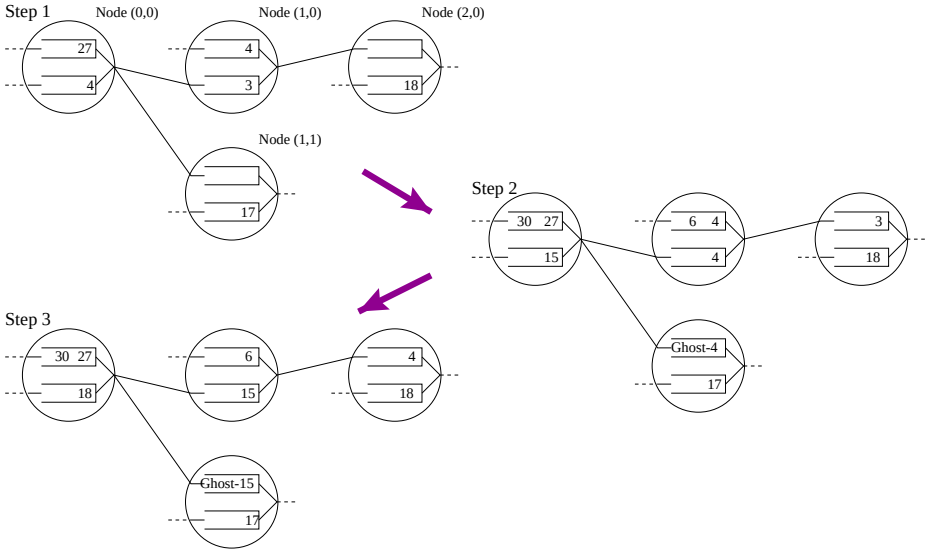


keep streams sorted by increasing addresses

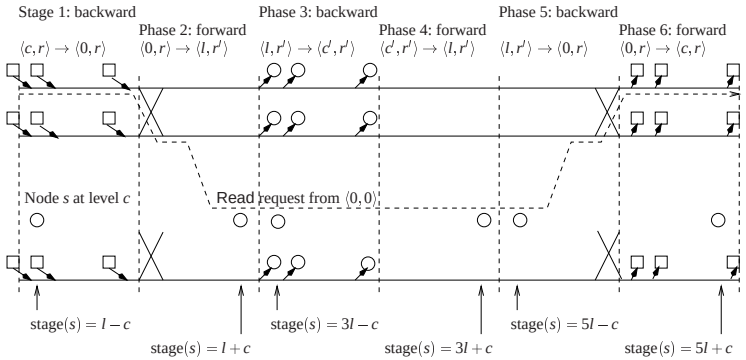
artificial Ghost messages to keep routing in flux

EndOfStep msgs

Routing of messages



Simulating one PRAM time step



Phase 1: Processor $\langle c,r\rangle$ sends the request to processor $\langle 0,r\rangle$
Phase 2: Processor $\langle 0,r\rangle$ sends the request to processor $\langle l,r'\rangle$
Phase 3: Processor $\langle l,r'\rangle$ sends the request to processor $\langle c',r'\rangle$.
Phase 4: Processor $\langle c',r'\rangle$ sends the reply to processor $\langle l,r'\rangle$
Phase 5: Processor $\langle l,r'\rangle$ sends the reply to processor $\langle 0,r\rangle$
Phase 6: Processor $\langle 0,r\rangle$ sends the reply to processor $\langle c,r\rangle$.

Simulating several PRAM time steps

If the hash function h_1 chosen turns out to be bad:
choose new h_1 and rehash the memory
in parallel in time $O(m/p \log p)$ time with high probability.
expected value for timeout + rehashing
to be balanced by expected simulation time for t steps

Theorem [Ranade'87]
An arbitrary t step program for a p-processor Combining CRCW PRAM with $t \geq m/p$
can be simulated on a p-processor butterfly in $O(t \log p)$ time
with high probability as $p \to \infty$ and/or $t \to \infty$.
The size of the memory required at each butterfly node is $O(m/p)$.

Making Ranade's emulation algorithm cost-optimal

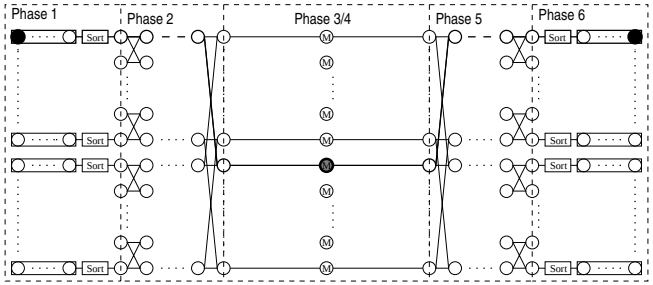
Efficiency of Ranade's algorithm: $\Omega(1/\log p)$.

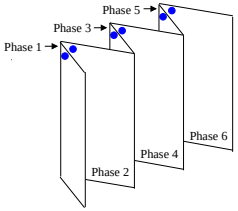
Improvement:

Each physical processor simulates $\log p$ PRAM processors.

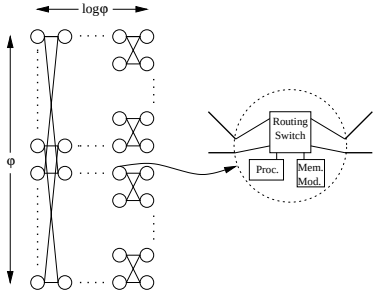
→ Phases 3 and 4 become superfluous

→ Phases 1 and 6 can be replaced by linear sorting arrays [PPP 4.2.1]

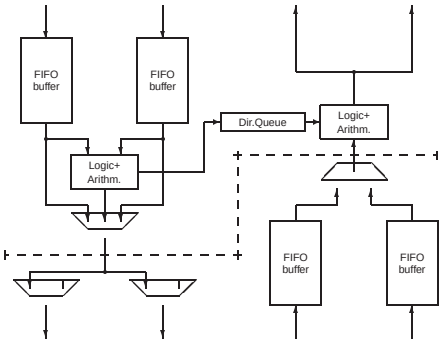




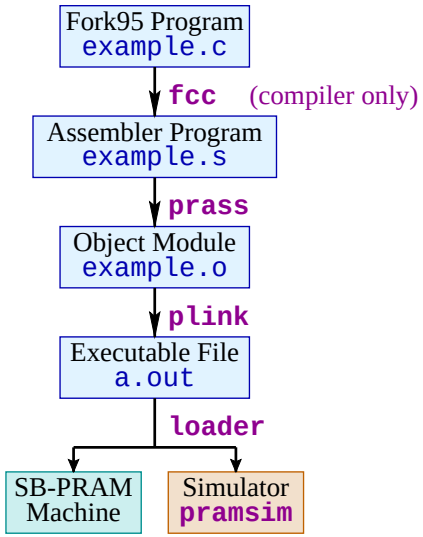
SB-PRAM: switch design



Block structure of a routing switch



Program design flow; system software tools



Configuration file: `.ldrc`

pramsim [Bosch/Franziskus'94]

`pramsim` [optional parameters] [executable file]

uses file `.pramsimrc`

`-M, --globmem`: shared memory size in words

`-P, --progmem`: program memory size in words

`-v, --virtProz`: # vP per pP

`-p, --physProz`: # pP

`--net-er`: warning on concurrent read

`--net-ew`: warning on concurrent write

Commands:

`init` *F V*	(Re)Initialize with *F* pP, *V* vP
`g`	Start or continue the program
`t` VAL	Trace. Execute 1 or VAL steps
`help`	Print help text
`r` *PROC*	Show all registers of vP *PROC*
`c` VAL	Show value VAL as decimal, hex, float, bin
`d` *MEM*	Disassemble the memory range *MEM*
`m` *MEM*	Show the memory area *MEM* as hex
`k` *MEM*	Show the memory area *MEM* as hex and ascii
`break` VAL	Set breakpoint at adress VAL
`q`	Quit from simulator

```
PRAM P0 = (p0, v0)>
```

Self-restoring exact barrier on SB-PRAM

```
    _barrier:
    bmc      0               /*continue at modulo=0*/
    getlo    -1,par2         /*load constant -1   0*/
    syncadd  par2,gps,1      /*atomic decrement   1*/
FORKLIB_SYNCLOOP:
    ldg      gps,1,r30       /*load sync cell     0*/
    getlo    1,par1          /*load constant 1    1*/
    add      r30,0,r30       /*compare sync cell  0*/
    bne      FORKLIB_SYNCLOOP/*all procs there?   1*/
    ldg      gps,1,r30       /*sync:cmp Sync,     0*/
    syncadd  par1,gps,1      /*restore sync cell  1*/
    add      r30,0,r30       /*compare with 0,    0*/
    bne      FORKLIB_SYNCHRON/*late wave skips nops*/
    nop                      /*early wave delayed 0*/
    nop                      /*early wave delayed 1*/
FORKLIB_SYNCHRON:
```


PRAMOS – Syscalls

SB-PRAM operating system PRAMOS [Grün/Rauber/Röhrig'95]

- no direct support for synchronous execution at user level
- Fork runtime system only uses the syscalls (esp., host file system I/O)

No.	type	name	parameters (C declaration)
0	int	open	char *file, int mode, flags
1	int	read	int fd, void *buf, int num
2	int	write	int fd, void *buf, int num
3	int	close	int fd
4	int	lseek	int fd, int offst, int orign
5	int	sys_std_open	int fd open stdin/-out/-err
6	int	sys_abort	int pc, int reason
7	int	sys_getnr	get my physical processor ID
8	int	sys_exit	call OS program exit routine
9	int	sys_getct	get the global counter
12	int	sys_getbas	get BASE register
13	void	sys_putbas	int base write BASE register