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

$P = F \times V$ 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

Distributed shared memory by hashing of addresses

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

Hash functions
$h_1: \{0, \ldots, m-1\} \rightarrow \{0, \ldots, p-1\}$ gives the module address
$h_2: \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m'-1\}$ 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
$h_1(x) = \left( \sum_{i=0}^{\lfloor \frac{m}{m'} \rfloor} a x^i \right) \mod P \mod p$
with $a$, randomly chosen, $P$ an appropriate prime (see [PPP 3.4])

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

Global sum

\[ s = s + \sum_{i=1}^{p} a_i \]

Prefix sums

\[ y_j = s + \sum_{i=1}^{j-1} a_i \]

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 \( s \)).

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 → scalable
- multiprefix on-the-fly [Ranade et al.’88]
- cost-efficient implementation: SB-PRAM [Abolhassan/Keller/Paul’91]
**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

**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 \( \langle 1, r' \rangle \)

**Phase 3:** Processor \( \langle 1, r' \rangle \) sends the request to processor \( \langle c', r' \rangle \).

**Phase 4:** Processor \( \langle c', r' \rangle \) sends the reply to processor \( \langle 1, r' \rangle \)

**Phase 5:** Processor \( \langle 1, r' \rangle \) 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 \( r \geq m/p \)
can be simulated on a \( p \)-processor butterfly in \( O(t \log p) \) time with high probability as \( p \rightarrow \infty \) and/or \( t \rightarrow \infty \).

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

```
Fork95 Program example.c
Assembler Program example.s
Object Module example.o
Executable File a.out
Simulator prassim
SB-PRAM Machine fccprass (compiler only)
Assembler Program prass
Object Module example.o
Executable File a.out
Simulator prassim
```

pramsim [optional parameters] [executable file]
uses file .pramsimrc

Commands:
init F V (Re)Initialize with F pP, vP VAL
q Start or continue the program
v 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 $P_0 = (p_0, v_0)$

Self-restoring exact barrier on SB-PRAM

```c
_bARRIER:
bmc 0 /*continue at modulo=0*/
getlo -1,par2 /*load constant -1 0*/
syncadd par2,gps,1 /*atomic decrement 1*/

FORKLIB_SYNCLOOP:
lgd 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*/

lgd gps,1,r30 /*load sync cell 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/Raubers/Röhrig’95]

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

<table>
<thead>
<tr>
<th>No.</th>
<th>type</th>
<th>name</th>
<th>parameters (C declaration)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>open</td>
<td>char *file, int mode, flags</td>
</tr>
<tr>
<td>1</td>
<td>int</td>
<td>read</td>
<td>int fd, void *buf, int num</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>write</td>
<td>int fd, void *buf, int num</td>
</tr>
<tr>
<td>3</td>
<td>int</td>
<td>close</td>
<td>int fd</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>lseek</td>
<td>int fd, int offst, int origin</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>sys_std_open</td>
<td>int fd open stdin/-out/-err</td>
</tr>
<tr>
<td>6</td>
<td>int</td>
<td>sys_abort</td>
<td>int pc, int reason</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>sys_getnr</td>
<td>get my physical processor ID</td>
</tr>
<tr>
<td>8</td>
<td>int</td>
<td>sys_exit</td>
<td>call OS program exit routine</td>
</tr>
<tr>
<td>9</td>
<td>int</td>
<td>sys_getct</td>
<td>get the global counter</td>
</tr>
<tr>
<td>10</td>
<td>int</td>
<td>sys_getbas</td>
<td>get BASE register</td>
</tr>
<tr>
<td>11</td>
<td>void</td>
<td>sys_putbas</td>
<td>int base write BASE register</td>
</tr>
</tbody>
</table>