IV. Physical organization and models

March 9th, 2009

Content

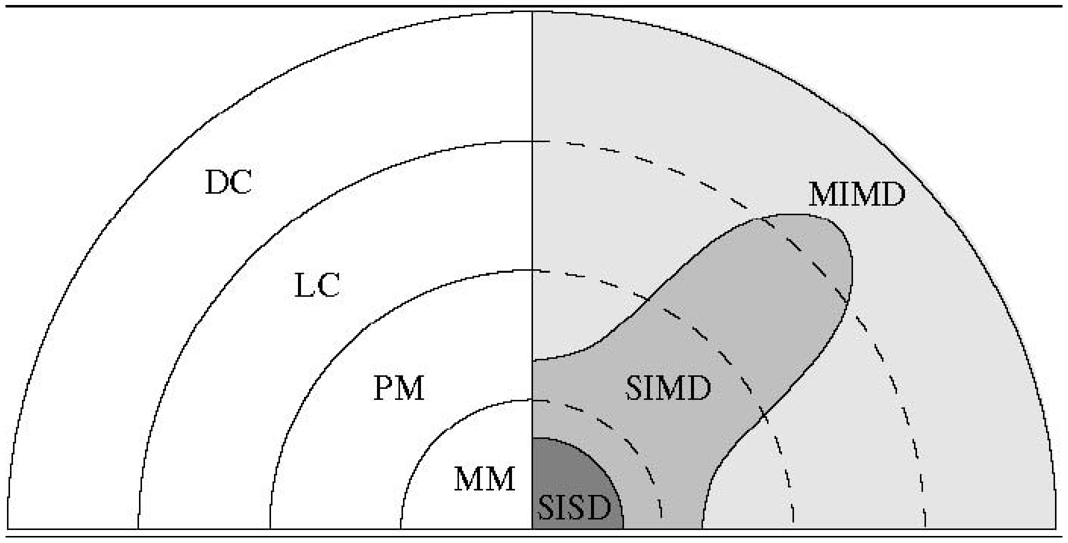
- Physical organization:
  - radius-based classification,
  - multicore,
  - clusters,
  - grids,
  - trends;
- Models:
  - early models,
  - PRAM

Physical organization

Radius-based Classification

- **Monoprocessor Machines (MM):** mainly representing the SISD computers available in the mass-market (PCs, workstations).
  - They also include SIMD machines containing one vector processor.
- **Parallel Machines (PM):** built as a single machine containing several processing units.
  - They include SIMD and MIMD architectures and potential combinations of them.
- **Local Clusters (LC):** collections of independent computers gathered in the same place and connected via a local network
  - Although they are intrinsically MIMD oriented, SIMD machines can be used at the node level.
- **Distributed Clusters (DC):** collections of local clusters scattered all around the world and linked together via the Internet.
  - Those systems mainly follow the MIMD model but they can include SIMD parts.

Links between Flynn's and radius-based classification



Examples of recent architectures

- **Intel Pentium D** –
  - introduced in 2005,
  - Intel’s first dual-core processor;
  - cores have their own caches and access the common memory via the frontside bus;
  - limited memory bandwidth in the case of memory-intensive computations.
  - its’s long pipelines allow for high clock frequencies (at the time of writing up to 3.73 GHz with the Pentium D 965), but may cause poor performance in the case of branches.
  - is not dual-CPU-capable. This capability is reserved for the rather expensive Xeon CPU.
- **Intel Core 2 Duo**
  - successor to the Pentium D
  - It abandons high clock frequencies in the favour of more efficient computation.
  - Like the Pentium D, it uses the frontside bus for memory access by both CPUs.
- **AMD Athlon 64 X2 & Opteron** A
  - MD’s dual-core CPUs Athlon 64 X2 (single-CPU only) and
  - Opteron (depending on model up to 8 CPUs in one system possible)
  - very popular CPUs for Linux clusters.
  - Each core has its own HyperTransport channel for memory access, making these CPUs well suited for memory intensive applications.
- **IBM pSeries**
  - IBM’s server- and workstation line based on the POWER processor.
  - The newer POWER processors are multi-core designs and feature large caches. IBM builds shared memory systems with up to 32 CPUs.

Examples of recent architectures

- **IBM BlueGene**
  - MPP (massively parallel processing) architecture by IBM.
  - It uses rather slow 700 MHz PowerPC processors.
  - these processors form very large, highly integrated distributed memory systems, with fast communication networks (a 3D-Torus, like the Cray T3E).
  - BlueGene/L consists of 131,072 CPUs, and delivers a performance of up to 360 TeraFLOPS.
- **NEC SX-8**
  - one of the few vector supercomputers in production at the moment.
  - It performs vector operations at a speed of 2 GHz, with eight operations per clock cycle.
  - One SX-8 node consists of eight CPUs, up to 512 nodes can be connected.
- **Cray XT3**
  - a massively-parallel system using AMD's Opteron CPU.
- **SGI Altix 3700**
  - ccNUMA system using Intel's Itanium 2 processor.
  - Itanium 2 has large caches and good floating point performance.
  - being ccNUMA, the Altix 3700 is easy to program.

Local Clusters

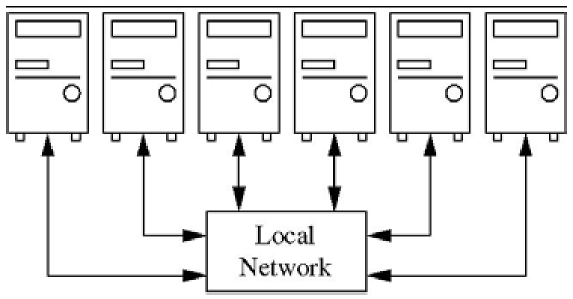
- A practical way of building efficient low cost distributed memory MIMDs.
- Clusters made high-performance parallel computing available to those with much smaller budgets.
- The idea is to combine commodity-off-the-shelf (COTS) components to create a parallel computer.
- Example, on PCs running the Linux operating system.
- Communication network connecting the PCs together may vary from Gigabit Ethernet to Myrinet and Infiniband that can broadcast messages at a rate of several Ggabits per second (Gbs)
  - Gigabit Ethernet has ~ 100 MB/s & cheaper than Myrinet, but the latency (travel time of a data package) is 100 mus forGigabit Ethernet > 10 - 20 mus forMyrinet.
    - At a clock speed of 2 GHz, one cycle takes 0.5 ns. A latency of 10 mus amounts to 20,000 cycles of travel time before the data package reaches its target.
- Possibility of combining higher-end shared mem.PCs & servers into clusters.
- Limitations:
  - Low-throughput switches can result in imbalanced systems & become a major performance bottleneck,
    - especially when more powerful nodes are used in the cluster.

The Beowulf project: the first PC cluster

- The first PC cluster was designed in 1994 at NASA Goddard Space Flight Center to achieve one Gigaflop.
  - 16 PCs were connected together using a standard Ethernet network.
  - Each PC had an Intel 486 microproc with sustained performance of ~ 70 Mflops.
  - Was built for only $40,000 compared to $1 million, which was the cost for a commercial equivalent supercomputer at that time.
  - Named *Beowulf* after the hero of medieval times who defeated the giant Grendel.
- In 1997 researchers at the Oak Ridge national laboratory built a Beowulf cluster from many obsolete PCs of various types;
  - for example, in one version it included 75 PCs with Intel 486 microprocessors, 53 Intel Pentium PCs and five fast Alpha workstations.
  - simulations producing detailed national maps of ecoregions based on almost 100 million degrees of freedom.
- When the computers of clusters are PCs running the Linux operating system, these clusters are called *Beowulf clusters*.
- combination of several desktop computers - known as a *network of workstations* (NOW) or clusters of workstations (COW)
- *vendor solution*: consists in providing specific integration facilities (racks and cabinets) with optimized network and software environment.

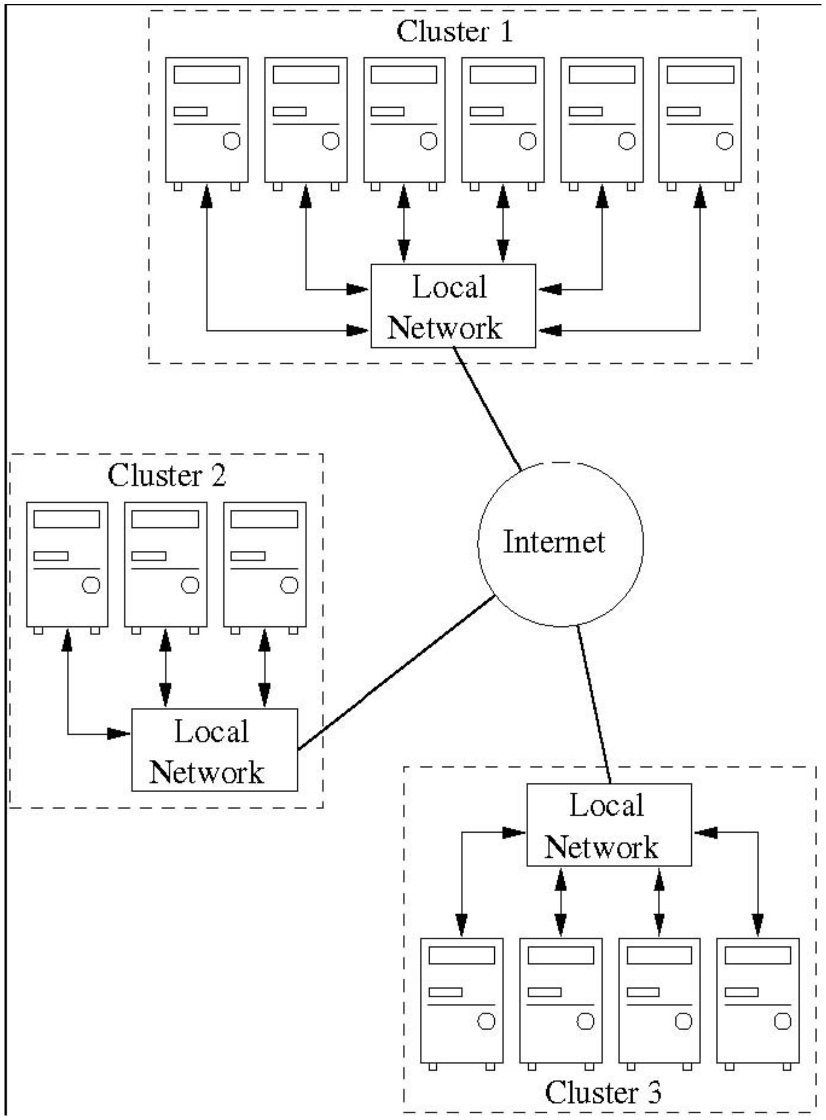
Advantages & disadvantages

- Adv:
  - Flexibility: Pus, network, RAM can be easily added or suppressed from the system
  - Cheaper than closed parallel machines
- Dis:
  - interconnection network which is often far slower than the fully integrated ones in parallel machines.
  - the connection of each node to the network is done through the connection bus of that node which often has restricted bandwidths and/or latencies



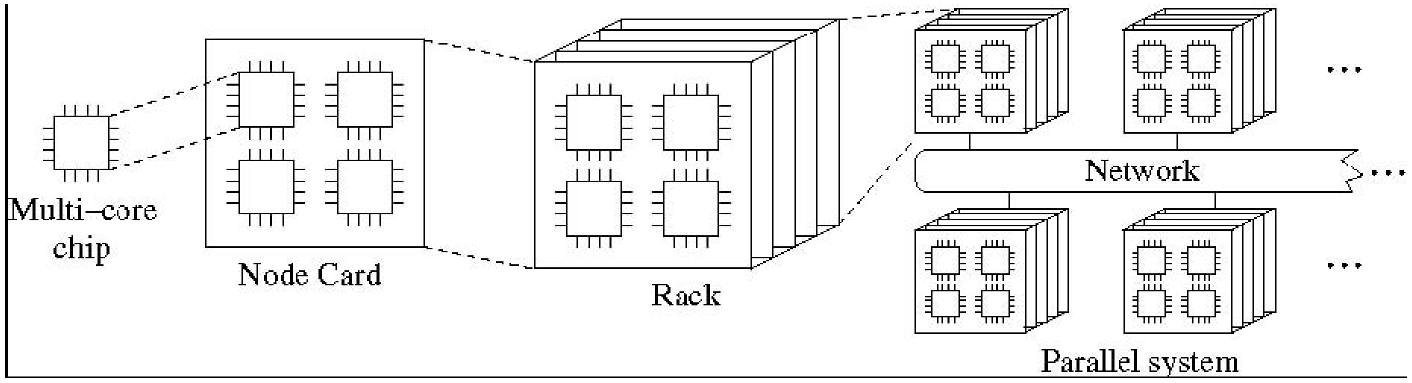
Distributed Clusters - Grids

- the cluster idea has been expanded to connecting computers all over the world via the Internet
- a logical result of
  - the great improvements in distant networks during the last few years
  - the stronger and stronger demand for more and more powerful computational systems.
- Adv:
  - gathering very large no. of machines => larger computational power & larger memory capacity.
- Drawbacks:
  - communications between clusters are generally much slower than those inside local clusters
  - security issues



Trends of used configurations

- Hierarchical parallel systems, mixing shared and distributed memory
  - E.g. IBM BlueGene



- Some vendors continue to develop specific processors to put in supercomputers such as vector, massively multi-threaded or VLIW (Very Large Instruction Word) processors.
- Re-use the parallel concepts usually taking place at the processor level at the scale of small groups of processors in order to design yet more powerful virtual processors.
  - IBM projects of Virtual Vector Architecture and Cell processor.

Trends of used configurations

- Networks:
  - Gigabit Ethernet has been intensively used in clusters, but high latencies => Other networks:SCI, Infiniband or Myrinet.
  - Bandwidths of the order of the Gb/s
  - Latencies the order of the microsecond.
  - Large flexibility in the possible topologies.
- Inclusion of heterogeneity at different levels of the parallel architectures.
  - Vendors, such as Cray or SGI, working on systems combining several kinds of processors (vector, scalar)
  - the networks are already heterogeneous in all the hierarchical architectures.
- The frontiers between the different parts of a parallel system are becoming less and less obvious!
  - Processors tend to become mini multi-processor systems
  - Clustering tends to be used at all the levels of multi-layer systems
    - the term cluster alone becomes more and more inaccurate outside any specific context.

Models of parallel computers

Why?

- Parallel processors come in many different varieties.
  - Not possible to discuss all varieties, including their distinguishing features, strong points within application contexts, and drawbacks.
  - ⇒ we often deal with abstract ***models of real machines.***
- Benefits of using abstract models:
  - technology-independent theories and algorithmic techniques that are applicable to a large number of existing and future machines.
  - The conceptual simplicity of such models makes the development of algorithms and the analysis of various trade-offs more manageable.
  - If automatic translation of these abstract algorithms into efficient programs for real machines is possible through the use of intelligent or optimizing compilers, then these models can indeed be enormously helpful.
- Disadvantages include
  - the inability to predict the actual performance accurately and
  - a tendency to simplify the models too much, so that they no longer represent any real machine.

Early models – Associative processing AP

- Associative or content-addressable memories (AMs, CAMs),
  - allow memory cells to be accessed based on contents rather than their physical locations within the memory
  - came in the 1950s when advances in magnetic and cryogenic memory technologies allowed the construction of reasonably sized prototypes.
- Based on incorporating simple processing logic into the memory array
  - ⇒ remove the need for transferring large volumes of data through the limited-bandwidth interface between the memory and the processor (the *von Neumann bottleneck*).
- Early associative memories provided two basic capabilities:
  1. masked search, or looking for a particular bit pattern in selected fields of all memory words and marking those for which a match is indicated, and
  2. parallel write, or storing a given bit pattern into selected fields of all memory words previously marked.
  - These capabilities + logical operations on mark vectors (e.g., ORing them together) suffice for the programming of searches or even parallel arithmetic ops.

AM/AP model

- Over the past half-century, the AM/AP model has evolved through the incorporation of additional capabilities, so that it is in essence converging with SIMD-type array processors.
- Early examples: Goodyear STARAN processor, commercial product of the 1970s, whose design was motivated by the computation-intensive problem of aircraft conflict detection;
  - O(n^2) pairwise checks required to avoid collisions & near misses for *n* aircraft in the vicinity of a busy airport.
- Modern incarnations of this model are seen in processor-in-memory (PIM) chips,
  - basically standard DRAM chips with a large no. very simple processors added on their data access paths,
  - and intelligent RAM (IRAM) architectures,
  - advantages in both performance and power consumption.

Early models–neural networks & cellular automata

- *Neural networks*
  - introduced in the 1950s,
  - dealt with parallel processing for image understanding applications
  - the development of *perceptrons* (a neuron-like device in charge of processing a single pixel in a digital image) in1940s.
  - in the 1960s a flurry of research activities laid the foundation for the modern field of *neural networks.*
  - introduction of the back propagation learning algorithm put neural networks on the fast track so that today they are used for the solution of complex decision problems in a wide class of appls.
- *Cellular automata*
  - a collection of identical finite-state automata that are interconnected, through their input–output links, in a regular fashion, with the state transitions of each automaton controlled by its own state, the states of the neighbors to which it is connected, and its primary inputs, if any.
  - *Systolic arrays*, which form the basis of high-performance VLSI-based designs in some application areas, can be viewed as cellular automata.
  - Recent years: a resurgence of interest in CA as theoretical models of massively parallel systems& tools for modeling physical phenomena.

PRAM model - abstraction

- The theoretical model used for conventional or sequential computers (SISD class) is known as the *random-access machine* (RAM)
  - not to be confused with random-access mem, which has the same acronym
- The parallel version of RAM, PRAM (pea-ram), constitutes an abstract model of the class of global-memory parallel processors.
- The abstraction consists of
  - ignoring the details of the processor-to-memory interconnection network and
  - taking the view that each processor can access any memory location in each machine cycle, independent of what other processors are doing.
- Example:
  - PRAM algorithms might involve statements like

  "for $0 <= i < p$, Processor *i* adds the contents of memory location $2i + 1$ to the memory location $2i$" (different locations accessed by the various processors)

  "each processor loads the contents of memory location *x* into its Register 2"

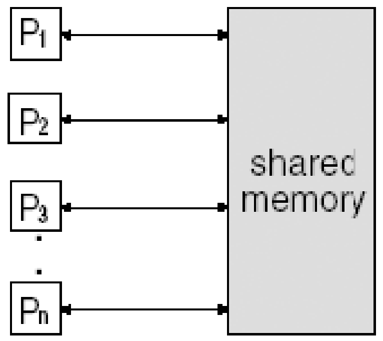
  (the same location accessed by all processors).
- Problem of multiple processors attempting to write into a common memory location must be resolved in some way.
  - various inhibition, priority, or combining schemes can be employed when concurrent write operations to a common location are attempted.

PRAM – functionality?

- a single processor is assumed to be active initially.
  - In each computation step, each active proc can read from and write into the shared memory and can also activate another processor.
- the abstract PRAM model can be SIMD or MIMD.
  - SIMD variant: all procs obey the same instruction in each machine cycle;
  - because of indexed and indirect (register-based) addressing, they often execute the operation that is broadcast to them on different data.
- Processors share a common clock but may execute different instructions in each cycle
- the PRAM model is highly theoretical.
  - If one were to build a physical PRAM, the processor-to-memory connectivity would have to be realized by an interconnection network
  - Because memory locations are too numerous to be assigned individual ports on an interconnection network, blocks of memory locations (or modules) would have to share a single network port.
    - Each instruction cycle would have to consume O(log *p*) real time.

PRAM as an ideal model

- Usually is assumed that the PRAM as seen as global memory of *unbounded size* that is uniformly accessible to all processors.
- Suppose a PRAM as shared-memory computer with *p* processors and a global memory of *m* words.
  - The processors are connected to the memory through a set of switches.
  - These switches determine the memory word being accessed by each processor.
  - Each of the *p* processors in the ensemble can access any of the memory words, provided that a word is not accessed by more than one processor simultaneously.
  - To ensure such connectivity, the total number of switches must be Θ(*mp*).
    - For a reasonable memory size, constructing a switching network of this complexity is very expensive.
    - ⇒ PRAM models of computation are impossible to realize in practice.



Memory access in PRAM model

- All the processors have read and write access to a shared global mem.
  - In the PRAM the access can be simultaneous.
  - Each of the theoretical processors can access the global shared memory in one uninterruptible unit of time
  - Each processor can perform various arithmetic& logical ops in parallel.
- The PRAM model has both concurrent and exclusive read algorithms.
  - Concurrent read algorithms are allowed to read the same piece of memory simultaneously with no data corruption.
  - Exclusive read algorithms are used to ensure that no two processors ever read the same memory location at the same time.
- The PRAM model also has both concurrent and exclusive write algs.
  - Concurrent write algorithms allow multiple processors to write to memory
  - Exclusive write algorithms ensure that no two processors write to the same memory at the same time.

Submodels of PRAM

1. **Exclusive-read, exclusive-write (EREW)** PRAM.
   - Access to a memory location is exclusive.
   - No concurrent read or write operations are allowed.
   - The weakest PRAM model: minimum concurrency in memory access.
   - The most realistic of the four submodels
2. **Concurrent-read, exclusive-write (CREW)** PRAM.
   - Multiple read accesses to a memory location are allowed.
   - Multiple write accesses to a memory location are serialized.
   - Default submodel: assumed when nothing is said about the submodel,
3. **Exclusive-read, concurrent-write (ERCW)** PRAM.
   - Multiple write accesses are allowed to a memory location,
   - Multiple read accesses are serialized.
4. **Concurrent-read, concurrent-write (CRCW)** PRAM.
   - Multiple read and write accesses to a common memory location.
   - This is the most powerful PRAM model.
   - The least restrictive submodel,
   - Require a conflict resolution mechanism to define concurrent writes

Protocols for concurrent write

- Common:
  - the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- Arbitrary:
  - an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- Priority:
  - all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- Sum,
  - the sum of all the quantities is written (the sum-based write conflict resolution model can be extended to any associative operator defined on the quantities being written).

CRCW PRAM is further classified

- Undefined (CRCW-U) :
  - In case of multiple writes, the value written is undefined.
- Detecting (CRCW-D):
  - A special code representing “detected collision” is written.
- Common (CRCW-C):
  - Multiple writes allowed only if all store the same value.
  - This is sometimes called the *consistent-write submodel.*
- Random (CRCW-R):
  - The value written is randomly chosen from among those offered.
  - This is sometimes called the *arbitrary-write submodel.*
- Priority (CRCW-P):
  - The processor with the lowest index succeeds in writing its value.
- Max/Min (CRCW-M):
  - The largest/smallest of the multiple values is written.
- Reduction:
  - The arithmetic sum (CRCW-S), logical AND (CRCW-A), logical XOR (CRCW-X), other combination of the multiple values is written.

Order the submodels

- One way to order these submodels is by their computational power.
- Two PRAM submodels are equally powerful if each can emulate the other with a constant-factor slowdown.
- A PRAM submodel is (strictly) less powerful than another submodel (denoted by the "<" symbol) if there exist problems for which the former requires significantly more computational steps than the latter.
- Example:
    - CRCW-D PRAM submodel < CRCW-M (maxim),
        - CRCW-M can find the largest number in a vector *A* of size *p* in a single step:
            - Processor *i* reads *A*[*i*] and writes it to an agreed-upon location *x*, which will then hold the maximum value for all processors to see
        - CRCW-D needs at least O(log *n*) steps.
- The "less powerful or equal" relationship "≤" between submodels can be similarly defined.
- EREW < CREW < CRCW-D < CRCW-C < CRCW-R < CRCW-P
- EREW can simulate CRCW submodel with at most logarithmic slowdown.
    - A p-processor CRCW-P (priority) PRAM can be simulated by a p-processor EREW PRAM with a slowdown factor of O(log p).

Alg.1: Data broadcasting – how?

- One processor needs to send a data value to all other processors.
  - In the CREW/CRCW, broadcasting trivial: the sending proc write the data value into a memory location, with all processors reading that data value in the following cycle.
    - => simple broadcasting is done in O(1) steps.
- Multicasting within groups is equally simple if each processor knows its group membership(s) & only members of each group read the multicast data
  - All-to-all broadcasting: each of the *p* procs needs to send a data value to all other processors - can be done through *p* separate broadcast operations in O(*p*) steps.
- The above scheme is clearly inapplicable to broadcasting in the EREW model.
- The simplest scheme for EREW broadcasting is
  - make *p* copies of the data value, say in a broadcast vector *B* of length *p*,
  - and then let each processor read its own copy by accessing *B*[*j*].
  - Initially, Processor *i* writes its data value into *B*[0].
  - ***Recursive doubling*** is used to copy *B*[0] into all elements of *B* in O (log 2 *p*) steps.
  - Finally, Processor *j*, $0 \leq j < p$, reads *B* [*j*] to get the data value broadcast by Processor *I*

```
Making p copies of B[0] by recursive doubling
for k = 0 to log2 p– 1 Processor j, 0 ≤ j < p, do
   Copy B[j] into B[j + 2k]
Endfor
```

Alg.1: Data broadcasting in PRAM

- Note: in Step *k* of the above recursive doubling process, only the first 2*k* processors need to be active.
- The complete EREW broadcast algorithm with this provision

EREW PRAM algorithm for broadcasting by Processor *I*

```
Processor $i$ write the data value into $B[0]$
$s := 1$
while $s < p$ Processor $j$, $0 \leq j < \min(s, p - s)$, do
    Copy $B[j]$ into $B[j + s]$
    $s := 2s$
endwhile
Processor $j$, $0 \leq j < p$, read the data value in $B[j]$
```

- The parameter *s* can be interpreted as the "span" of elements already modified or the "step" for the copying operation.
- O(log *p*)-step broadcasting alg. - is optimal for EREW PRAM.

Alg.2: All-to-all broadcasting in PRAM

- To perform all-to-all broadcasting,
  - each processor broadcasts a value that it holds to each of the other $p-1$ processors
  - Processor j write its value into $B[j]$, rather than into $B[0]$.
  - In one memory access step, all of the values to be broadcast are written into the broadcast vector *B.*
  - Each processor then reads the other $p-1$ values in $p-1$ memory accesses.
  - Ensure that all reads are exclusive: Procs j begins reading the values starting with $B[j+1]$, wrapping around to $B[0]$ after reading $B[p-1]$.

EREW PRAM data broadcasting without redundant copying.

```
EREW PRAM algorithm for all-to-all broadcasting
Processor $j$, $0 \leq j < p$, write own data value into $B[j]$
for $k = 1$ to $p - 1$ Processor $j$, $0 \leq j < p$, do
    Read the data value in $B[(j + k) \bmod p]$
endfor
```

Alg. 3: Naïve sorting algorithm

- Given a data vector S of length p,
- Let Proc j compute the rank $R[j]$ of the data element $S[j]$ and then store $S[j]$ into $S[R[j]]$.
- The rank $R[j]$ of $S[j]$
  - = total no. data elements that are smaller than $S[j]$,
  - computed by each proc examining all other data elements& counting no. elements $S[l] < S[j]$.
- Each data element must be given a unique rank => ties broken by using the proc ID.
  - If Processors i and j ($i < j$) hold equal data values, the value in Processor i is "smaller" for ranking purposes.

```
Naive EREW PRAM sorting algorithm using all-to-all broadcasting
Processor $j$, $0 \leq j < p$, write 0 into $R[j]$
for $k = 1$ to $p - 1$ Processor $j$, $0 \leq j < p$, do
    $l := (j + k) \bmod p$
    if $S[l] < S[j]$ or $S[l] = S[j]$ and $l < j$
            then $R[j] := R[j] + 1$
    endif
endfor
Processor $j$, $0 \leq j < p$, write $S[j]$ into $S[R[j]]$
```

- Not optimal in that the $O(p^2)$ computational work involved in it is significantly greater than the $O(p \log p)$ work required for sorting p elements on a single processor.

Semigroup and prefix comp.

- **Semigroup computation or fan in computation**:
  - is define based on associative binary operator o.
  - trivial for a CRCW PRAM of the “reduction” variety
  - Examples:
    - computing the arithmetic sum (logical AND, logical XOR) of *p* values, one per processor,
      - trivial for the CRCW-S (CRCW-A, CRCW-X) PRAM;
        - it can be done in a single cycle by each proc writing its corresponding value into a common location that will then hold the arithmetic sum of all of the values.
  - The recursive doubling scheme can be used on an EREW PRAM
    - the only difference appearing in the final broadcasting step.

- **Parallel prefix computations:**
  - consists of the first phase of the semigroup computation.
  - The divide-and-conquer paradigm:
    - Problem as composed of two subproblems:
    1. computing the odd-indexed results *s*1,*s*3,*s*5,...
    2. computing the even-index.results *s*0,*s*2,*s*4,...
    - The first subproblem is solved as follows.
      - Pairs of consecutive elements in the input list (*x*0 and *x*1, *x*2 and *x*3, *x*4 and *x*5, and so on) are combined to obtain a list of half the size.
    - Performing parallel prefix computation on this list yields values for all odd-indexed results.
    - The even indexed results are then found in a single PRAM step by combining each even-indexed input with the immediately preceding odd-indexed result.
    - The total computation time is given by the recurrence $T(p) = T(p/2) + 2$ whose solution is$T(p) = 2 \log2\ p$.

Matrix multiplication

- Given m x m matrices A and B, with elements $a[i,j]$ and $b[i,j]$,, their product C can be obtained with a O(m^3)-step sequential algorithm.
- If the PRAM has $p = m^3$ processors, then matrix multiplication can be done in O(log m) time
    - one processor compute $a[i,k] \times b[k,j]$ and then allow groups of m procs to perform m-input +s (semigroup comp) in O(log m) time.
    - Because we are usually not interested in parallel processing for matrix multiplication unless m is fairly large, this is not a practical solution!
- Assume that the PRAM has $p = m^2$ processors.
    - In this case, matrix multiplication can be done in O(m) time by using one processor to compute each element $c[i,j]$ of the product matrix C.
    - Processor responsible for computing $c[i,j]$
        - reads the elements of Row i in A and the elements of Column j in B,
        - multiplies their corresponding kth elements, and
        - adds each of the products thus obtained to a running total t.
    - Parallelize the i and j loops in the sequential algorithm.
    - Label the m^2 processors with two indices (i, j), each ranging from 0 to $m - 1$, rather than with a single index ranging from 0 to $m^2 - 1$.

CREW implementation of A x B

PRAM matrix multiplication algorithm using m^2 processors

```
Processor (i, j), 0 ≤ i, j < m, do
begin
    t := 0
    for k = 0 to m – 1 do
    t := t + a[i,k]b[k,j]
    endfor
    cij := t
end
```

- In a given iteration of the *k* loop,
  - all processors (i, y), $0 \leq y < m$, access the same element *a[i,k]* of *A*
  - all processors (x, j) access the same element *b[j,k]* of *B.*

A x B using *m* processors

- Matrix multiplication can be done in O(m^2) time
  - Processor *i* to compute the *m* elements in Row *i* of the product matrix *C* in turn.
  - Processor *i* will
    - read the elements of Row *i* in *A* and the elements of all columns in *B,*
    - multiply their corresponding *k*th elements, and
    - add each of the products thus obtained to a running total *t.*
- Parallelize the *i* loop in the sequential algorithm.

  PRAM matrix multiplication algorithm using *m* processors

```
for $j$ = 0 to $m – 1$ Processor $i$, $0 \leq i < m$, do
        $t := 0$
        for $k$ = 0 to $m – 1$ do
                $t := t + a[b,i]\ b[k,j]$
        endfor
        $cij := t$
Endfor
```

- Each processor reads a different row of *A* => no concurrent reads are attempted
- *B:* all *m* processors access the same element *bk j* a t the same time.
- For both $p = m^2$ and $p = m$ procs: efficient algorithms with linear speed-ups.

A x B using less than *m* processors

- Naïve alg:
  - We can let Processor *i* compute a set of *m/p* rows in the result matrix *C*; say Rows *i, i* + *p, i* + 2*p*, . . . , *i* + (*m/p* – 1)*p.*
  - Parallelizing the *i* loop as this is preferable
    - if *k* loop -- has data dependencies
    - if *j* loop -- imply *m* synchronizations of the processors, once at the end of each *i* iteration, assuming the SPMD model
  - Drawback: each element of *B* is fetched *m/p* times, with only two arithmetic operations (+,x) performed for each such element.
- Block matrix multiplication:
  - increases the computation to memory access ratio
  - divide the *m* x *m* matrices *A, B,* and *C* into *p* blocks of size *q* x *q.*
  - multiply the *m* x *m* matrices by using matrix x with processors, where the terms in the algorithm statement *t* := *t* + *a[i,k]b[k,j]* are now *q* x *q* matrices
  - Processor (*i, j*) computes Block (*i, j*) of the result matrix *C.*
  - Each multiply–add computation on *q* x *q* blocks needs $2q^2 = 2m^2/p$ memory accesses to read the blocks and $2q^3$ arithmetic operations.
  - *q* arithmetic operations are performed for each memory access and better performance will be achieved as a result of improved locality.