Parallel Computation Models

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@cs.rice.edu

Acknowledgements for today’s lecture

- Section 2.4.1 of textbook
- Larry Carter, UCSD CSE 260 Topic 6 lecture: Models of Parallel Computers
  - http://www-cse.ucsd.edu/users/carter/260/260class06.ppt
- Michael C. Scherger, “An Overview of the BSP Model of Parallel Computation”
  - http://www.cs.kent.edu/~jbaker/PDA-Sp07/slides/bsp%20model.ppt

Outline

- PRAM (Parallel Random Access Machine)
- PMH (Parallel Memory Hierarchy)
- BSP (Bulk Synchronous Parallel)
- LogP

Review: The <u>R</u>andom <u>A</u>ccess <u>M</u>achine Model for Sequential Computing

<u>RAM</u> model of serial computers:

- Memory is a sequence of words, each capable of containing an integer.
- Each memory access takes one unit of time
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable
- Read-only input tape, write-only output tape

PRAM [Parallel Random Access Machine]

(Introduced by Fortune and Wyllie, 1978)

PRAM composed of:

- P processors, each with its own unmodifiable program.
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer.
- a read-only input tape.
- a write-only output tape.

PRAM model is a synchronous, MIMD, shared address space parallel computer.

- Processors share a common clock but may execute different instructions in each cycle.

Architecture of an Ideal Parallel Computer

- Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.
  - Exclusive-read, exclusive-write (EREW) PRAM.
  - Concurrent-read, exclusive-write (CREW) PRAM.
  - Exclusive-read, concurrent-write (ERCW) PRAM.
  - Concurrent-read, concurrent-write (CRCW) PRAM.

Architecture of an Ideal Parallel Computer

- What does concurrent write mean, anyway?
  - Common: write only if all values are identical.
  - Arbitrary: write the data from a randomly selected processor.
  - Priority: follow a predetermined priority order.
  - Sum: Write the sum of all data items.

Physical Complexity of an Ideal Parallel Computer

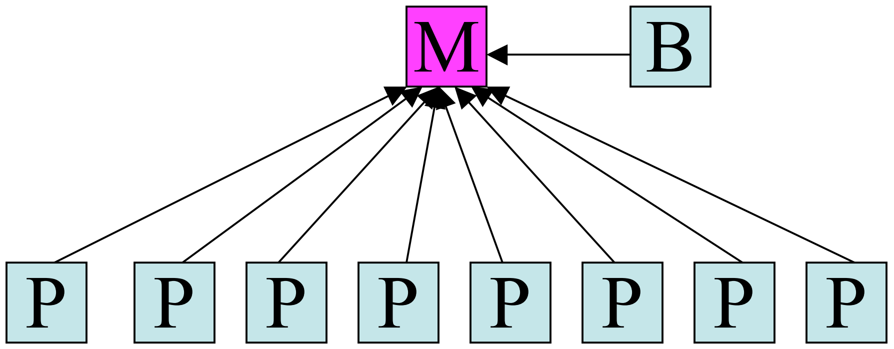
- Processors and memories are connected via switches.
- Since these switches must operate in *O(1)* time at the level of words, for a system of *p* processors and *m* words, the switch complexity is *O(mp).*
- Clearly, for meaningful values of *p* and *m*, a true PRAM is not realizable.

More PRAM taxonomy

- Different protocols can be used for reading and writing shared memory.
  - EREW - exclusive read, exclusive write
    - A program isn't allowed to have two processors access the same memory location at the same time.
  - CREW - concurrent read, exclusive write
  - CRCW - concurrent read, concurrent write
    - Needs protocol for arbitrating write conflicts
  - CROW – concurrent read, owner write
    - Each memory location has an official "owner"
- PRAM can emulate a message-passing machine by partitioning memory into private memories.

Broadcasting on a PRAM

- "Broadcast" can be done on CREW PRAM in O(1) steps:
  - Broadcaster sends value to shared memory
  - Processors read from shared memory



- Requires lg(P) steps on EREW PRAM.

Finding Max on a CRCW PRAM

- We can find the max of N distinct numbers x[1], ..., x[N] in constant time using N^2 procs!

  Number the processors P_{rs} with r, s ε {1, ..., N}.
  – Initialization: P_{1s} sets A[s] = 1.
  – Eliminate non-max's: if x[r] < x[s], P_{rs} sets A[r] = 0.
    Requires concurrent reads & writes.
  – Find winner: If A[r] = 1, P_{r1} sets max = x[r].

Some questions

1. What if the x[i]'s aren't necessarily distinct?
2. Can you sort N numbers in constant time?
   And only use only N^k processors (for some k)?
3. How fast can you sort on CREW?
4. Does any of this have any practical significance ????

PRAM is not a great success

- Many theoretical papers about fine-grained algorithmic techniques and distinctions between various modes.
- Results seem irrelevant.
  - Performance predictions are inaccurate.
  - Hasn’t lead to programming languages.
  - **Hardware doesn’t have fine-grained synchronous steps.**

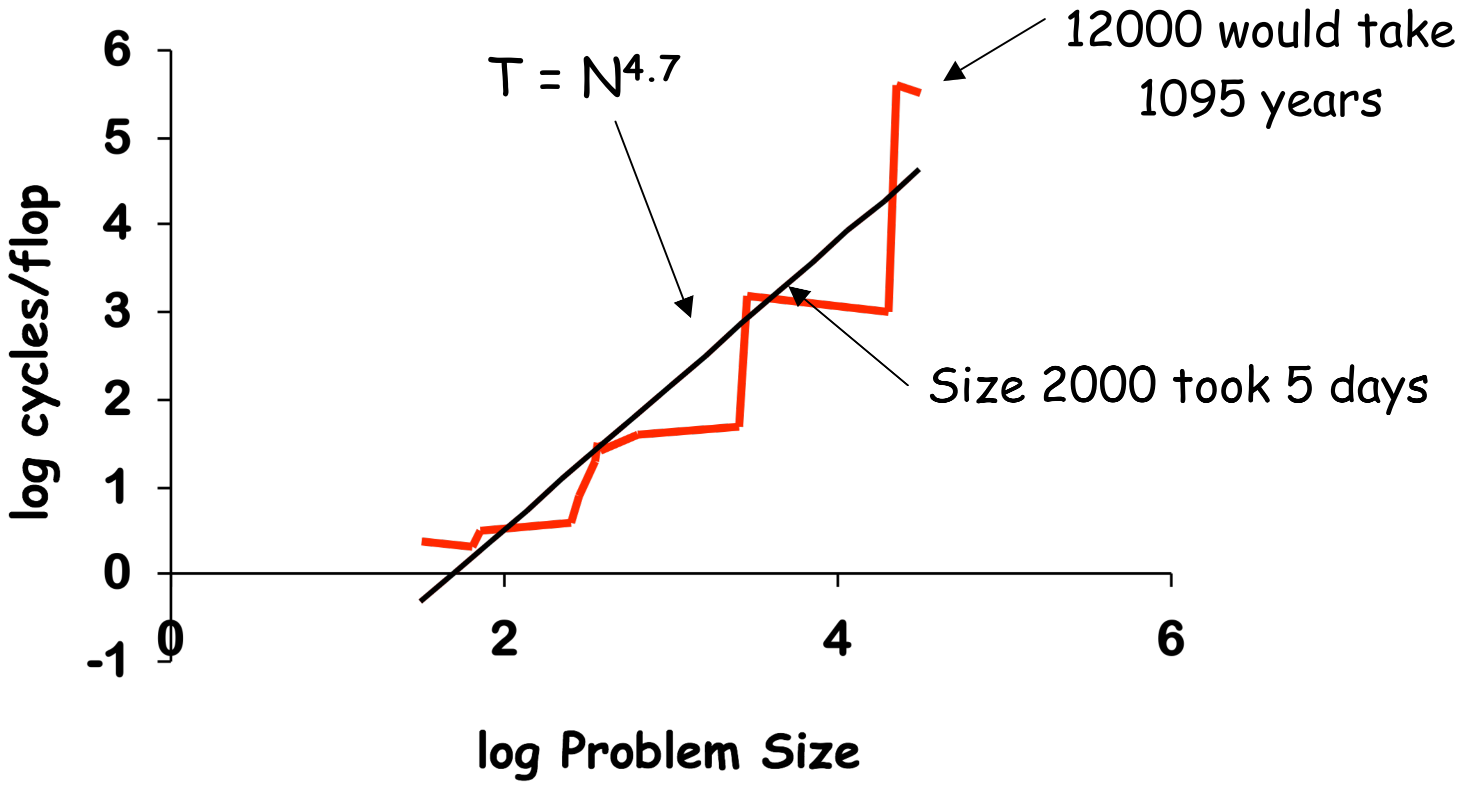
Another look at the RAM model

- RAM analysis says matrix multiply is O(N^3).

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      C[i,j] += A[i,k]*B[k,j]
```

- Is it??

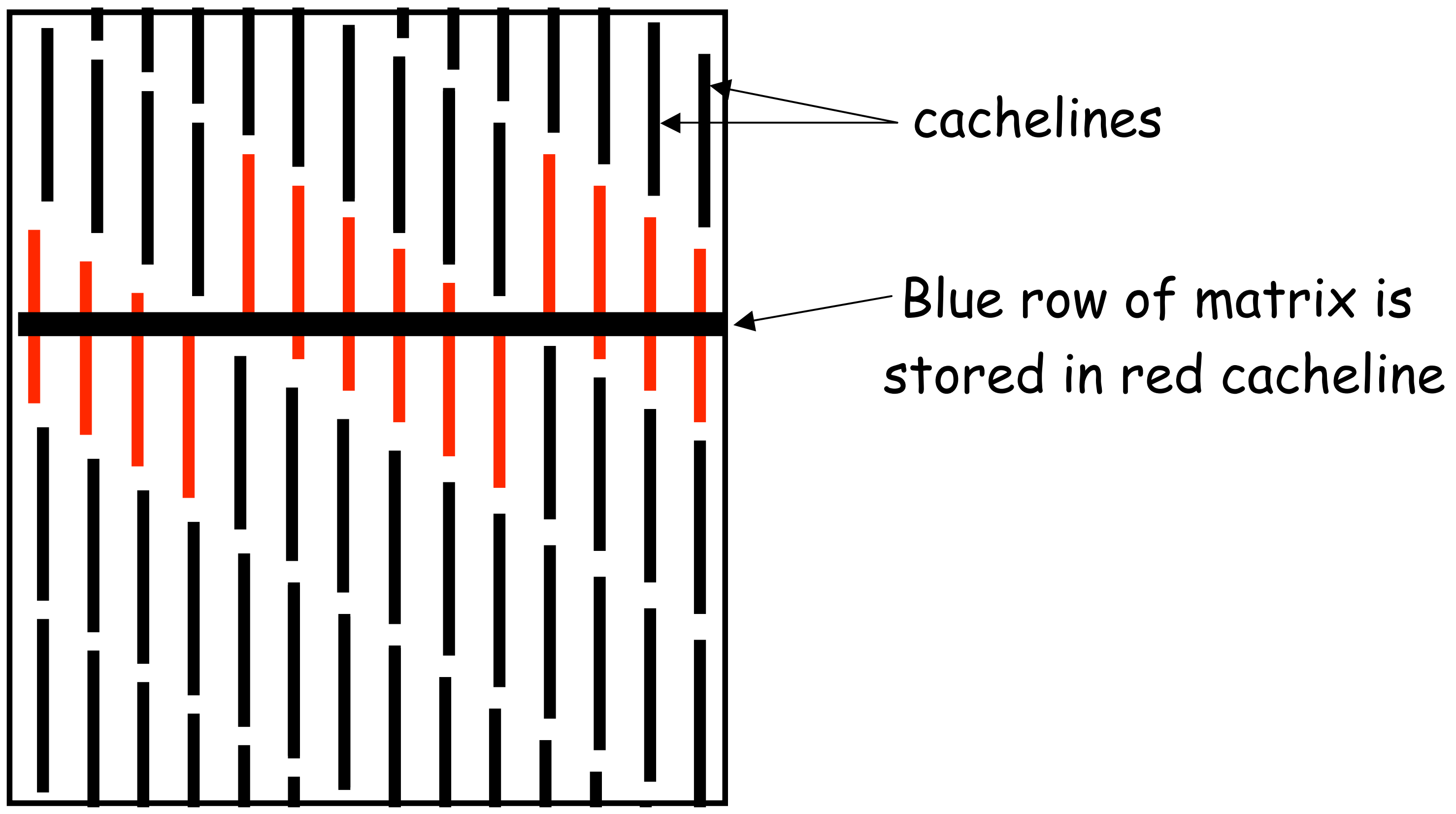
Matrix Multiply on RS/6000



O(N^3) performance would have constant cycles/flop

Performance looks much closer to O(N^5)

Column major storage layout



Memory Accesses in Matrix Multiply

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      C[i,j] += A[i,k]*B[k,j]
```

Stride-N access to one row*

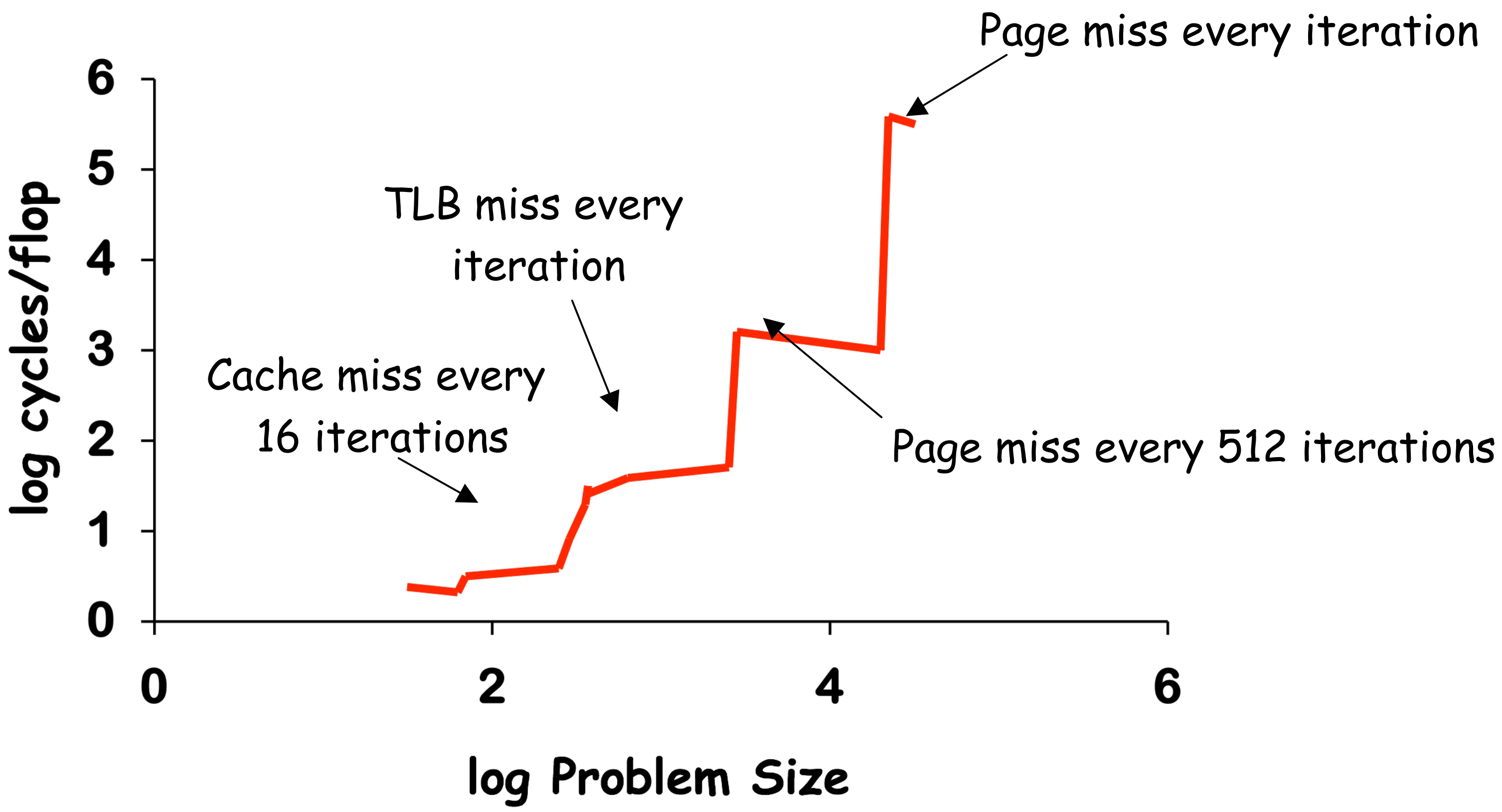
Sequential access through entire matrix

When cache (or TLB or memory) can't hold entire B matrix, there will be a miss on every line.

When cache (or TLB or memory) can't hold a row of A, there will be a miss on each access

* assumes data is in column-major order

Matrix Multiply on RS/6000



Where are we?

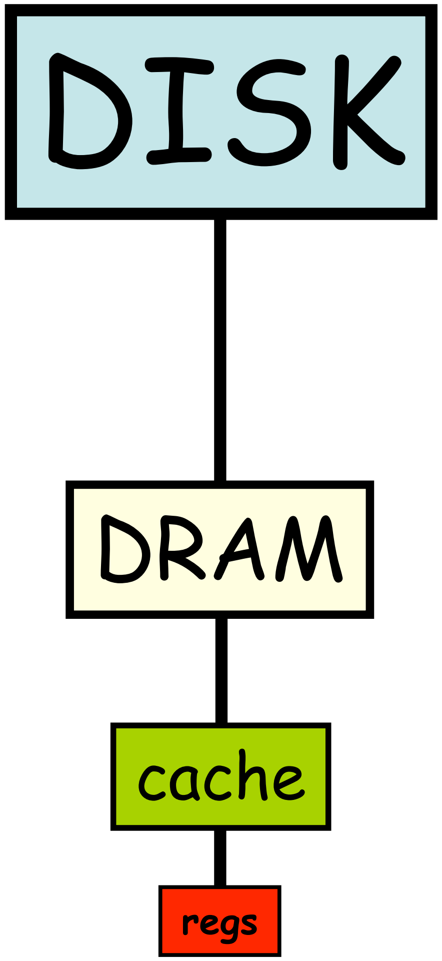
- RAM model says naïve matrix multiply is $O(N^3)$
- Experiments show it's $O(N^5)$-ish
- Explanation involves cache, TLB, and main memory limits and block sizes
- Conclusion: memory features are important and should be included in model.

Models of memory behavior

- Uniprocessor models looking at data access costs:
  - Two-level models (main memory & cache):
    - Floyd ('72), Hong & Kung ('81)
  - Hierarchical Memory Model
    - Accessing memory location i costs f(i)
    - Aggarwal, Alpern, Chandra & Snir ('87)
  - Block Transfer Model
    - Moving block of length k at location i costs k+f(i)
    - Aggarwal, Chandra & Snir ('87)
  - Memory Hierarchy Model
    - Multilevel memory, block moves, extends to parallelism
    - Alpern & Carter ('90)

Memory Hierarchy model

- A uniprocessor is
  - Sequence of memory modules
    - Highest level is large memory, low speed
    - Processor (level 0) is tiny memory, high speed
  - Connected by channels
    - All channels can be active simultaneously
  - Data are moved in fixed-sized blocks
    - A block is a chunk of contiguous data
    - Block size depends on level



Does MH model influence *your* thinking?

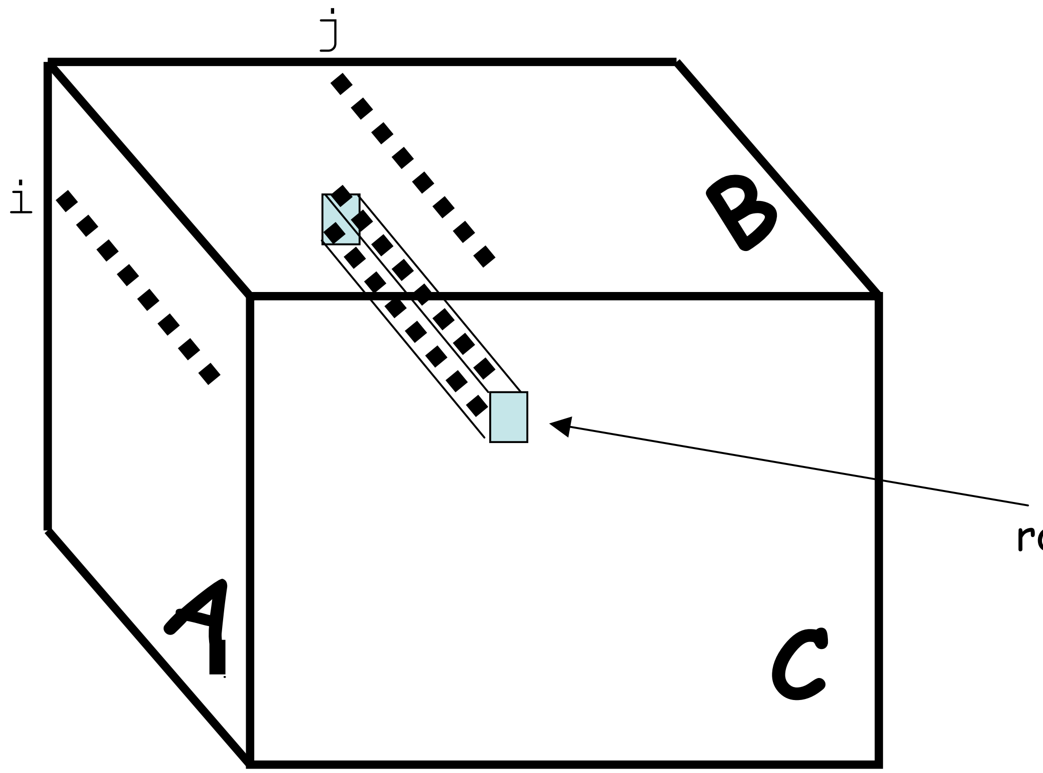
Say your computer is a sequence of modules:

- You want to move data to the fast one at bottom.
- Moving contiguous chunks of data is faster than moving inidividual words

How do you accomplish this??

- One possible answer: divide & conquer

Visualizing Matrix Multiplication

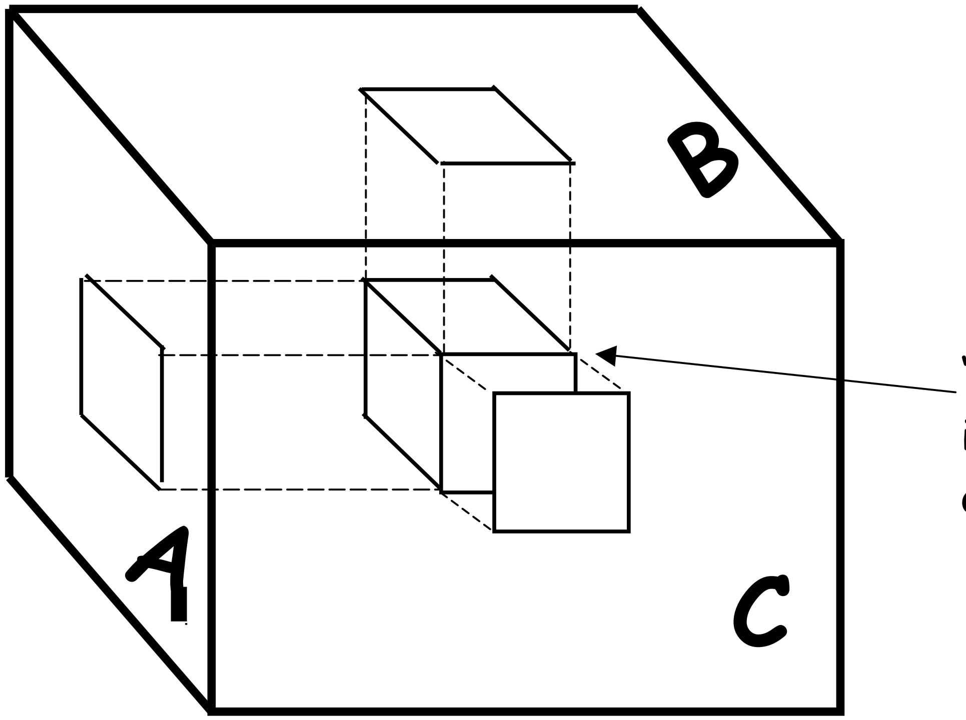


$C = A\,B$

"stick" of computation is dot product of a row of A with column of B

$c_{ij} = \Sigma\, a_{ik} * b_{kj}$

Visualizing Matrix Multiplication



"Cubelet" of computation is product of a submatrix of A with submatrix of B

- Data involved is proportional to surface area.
- Computation is proportional to volume.

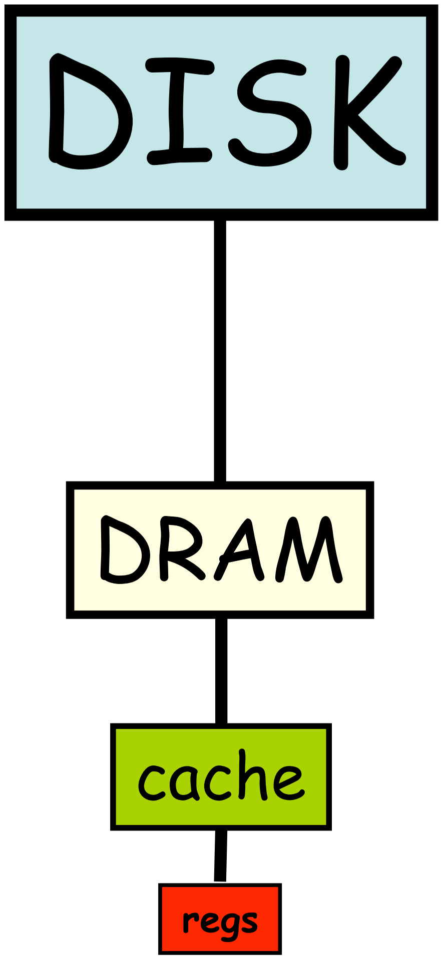
MH algorithm for C = AB

- Partition computation into “cubelets”
    - Each cubelet requires sxs submatrix of A and B
    - 3 s^2 data needed; allows s^3 multiply-adds
- Parent module gives child sequence of cubelets.
    - Choose s to ensure all data fits into child’s memory
- Child sub-partitions cubelet into still smaller pieces.
- Known as “blocking” or “tiling” long before MH model invented (but rarely applied recursively).

Theory of MH algorithm for C = AB

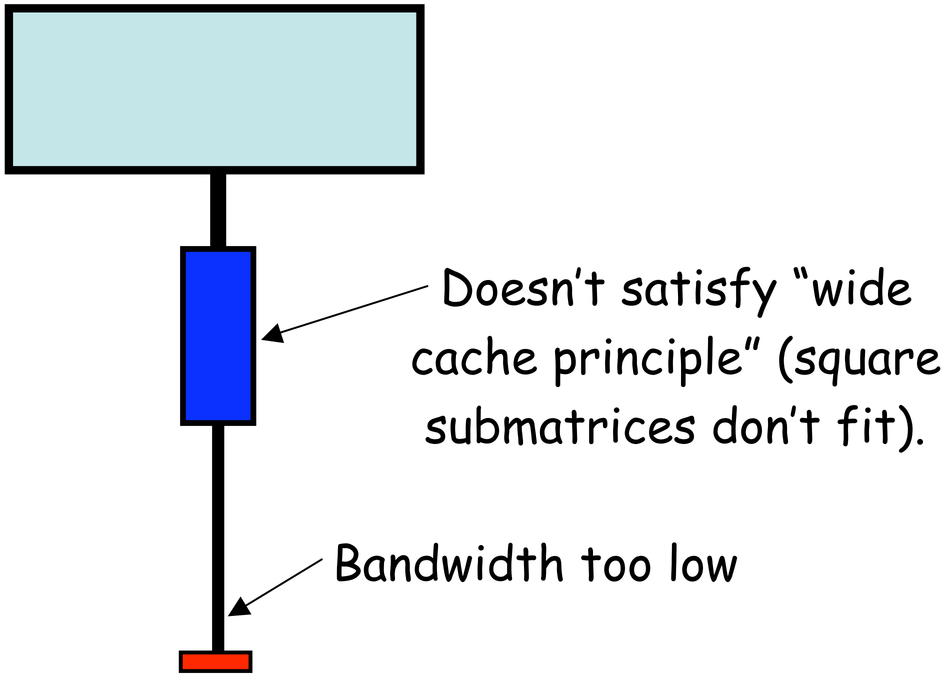
- "Uniform" Memory Hierarchy (UMH) model looks similar to actual computers.
  - Block size, number of blocks per module, and transfer time per item grow by constant factor per level.
- Naïve matrix multiplication is $O(N^5)$ on UMH.
  - Similar to observed performance.
- Tiled algorithm is $O(N^3)$ on UMH.
  - Tiled algorithm gets about 90% "peak performance" on many computers.
- Moral: good MH algorithm $\leftarrow\rightarrow$ good in practice.

Visualizing computers in MH model



This computer is reasonably well-balanced

Height of module = lg(blocksize)
Width = lg(number of blocks)
Length of channel = lg(transfer time)


Doesn't satisfy "wide cache principle" (square submatrices don't fit).
Bandwidth too low

This one isn't

Parallel Memory Hierarchy (PMH) model

Alpern & Carter: “Since MH model is so great, let’s generalize it for parallel computers!”

A computer is a *tree* of memory modules

- Largest memory is at root.
- Children have less memory, more compute power.

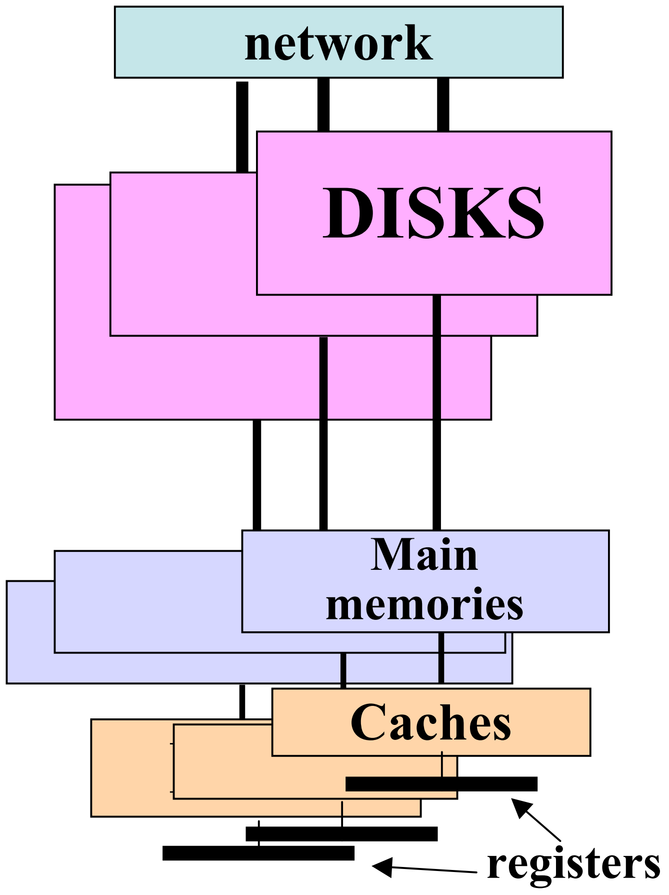
Four parameters per module

- Block size, number of blocks, transfer time from parent, and number of children.
- Homogeneous ←→ all modules at a level have same parameters

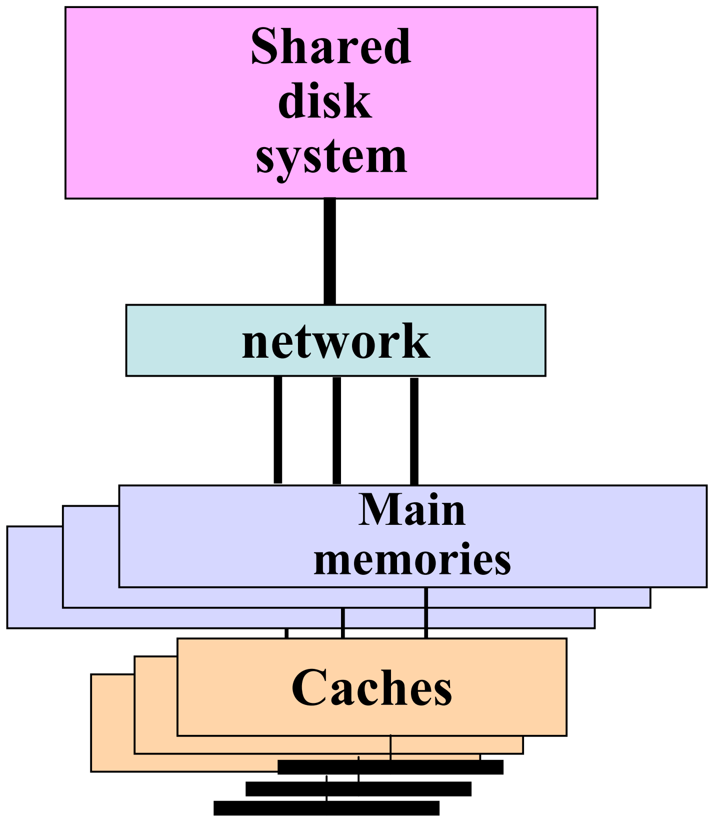
(PMH ignores difference between shared and distributed address space computation.)

PMH ideas have influenced Sequoia language at Stanford

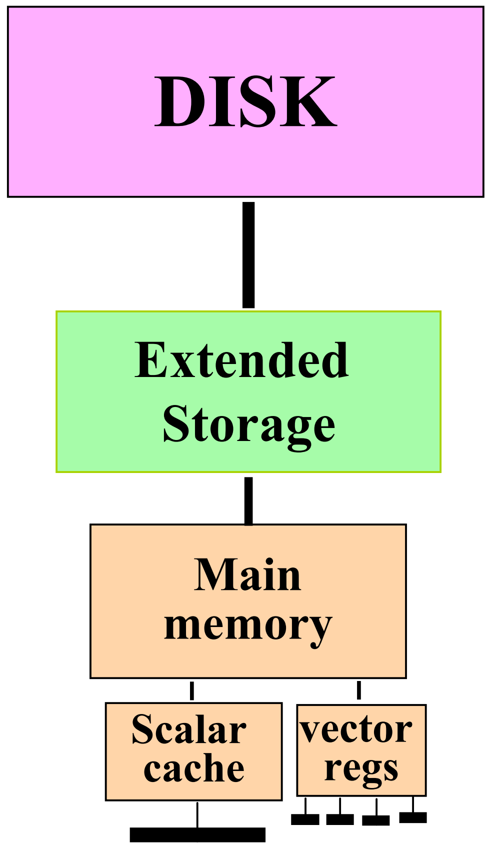
Some Parallel Architectures



The Grid

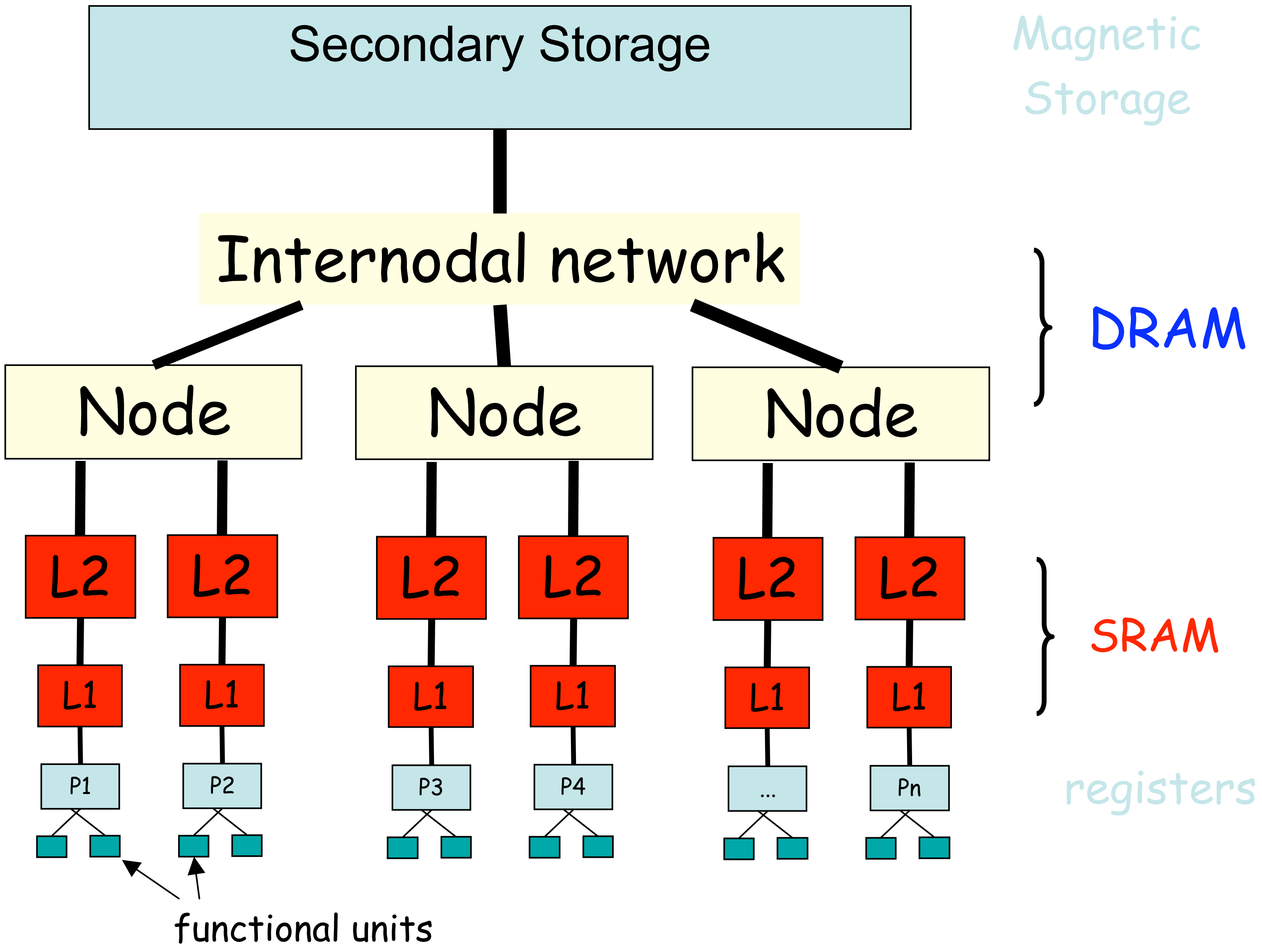


NOW



Vector supercomputer

PMH model of multi-tier computer



Observations

- PMH can model heterogeneous systems as well as homogeneous ones.
- More expensive computers have more parallelism and higher bandwidth near leaves
- Computers getting more levels & more branching.
- Parallelizing code for PMH is very similar to tuning it for a memory hierarchy.
  - Break computation into *independent* blocks
  - Send blocks of work to children

Needed for parallelization

BSP (Bulk Synchronous Parallel) Model

- Overview of the BSP Model
- Predictability of the BSP Model
- Comparison to Other Parallel Models
- BSPlib and Examples
- Comparison to Other Parallel Libraries

References

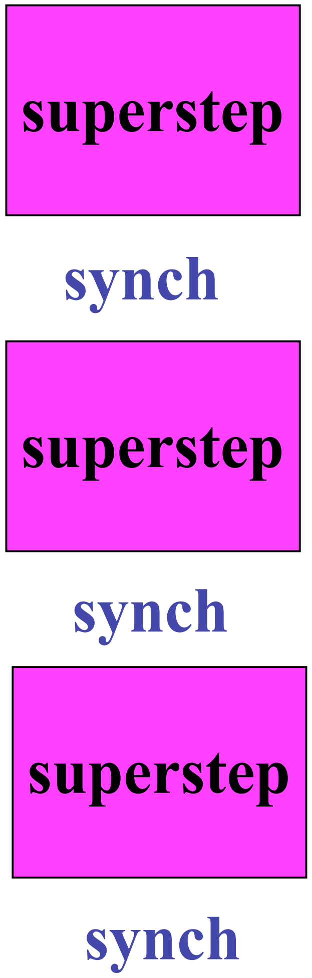
- “BSP: A New Industry Standard for Scalable Parallel Computing”, http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/oparl.html
- Hill, J. M. D., and W. F. McColl, “Questions and Answers About BSP”, http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/oparl.html
- Hill, J. M. D., et. al, “BSPlib: The BSP Programming Library”, http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/oparl.html
- McColl, W. F., “Bulk Synchronous Parallel Computing”, Abstract Machine Models for Highly Parallel Computers, John R. Davy and Peter M. Dew eds., Oxford Science Publications, Oxford, Great Brittain, 1995, pp. 41-63.
- McColl, W. F., “Scalable Computing”, http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/oparl.html
- Valiant, Leslie G., “A Bridging Model for Parallel Computation”, Communications of the ACM, Aug., 1990, Vol. 33, No. 8, pp. 103-111
- The BSP Worldwide organization website is http://www.bsp-worldwide.org and an excellent Ohio Supercomputer Center tutorial is available at www.osc.org.

What Is Bulk Synchronous Parallelism?

- The model consists of:
  - A set of processor-memory pairs.
  - A communications network that delivers messages in a point-to-point manner.
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes.
  - There are no special combining, replicating, or broadcasting facilities.
- In BSP, each processor has local memory. “One-sided” communication style is advocated.
  - There are globally-known “symbolic addresses” (like VSM)
    - Data may be inconsistent until next barrier synchronization
    - Valiant suggests hashing implementation of puts and gets.

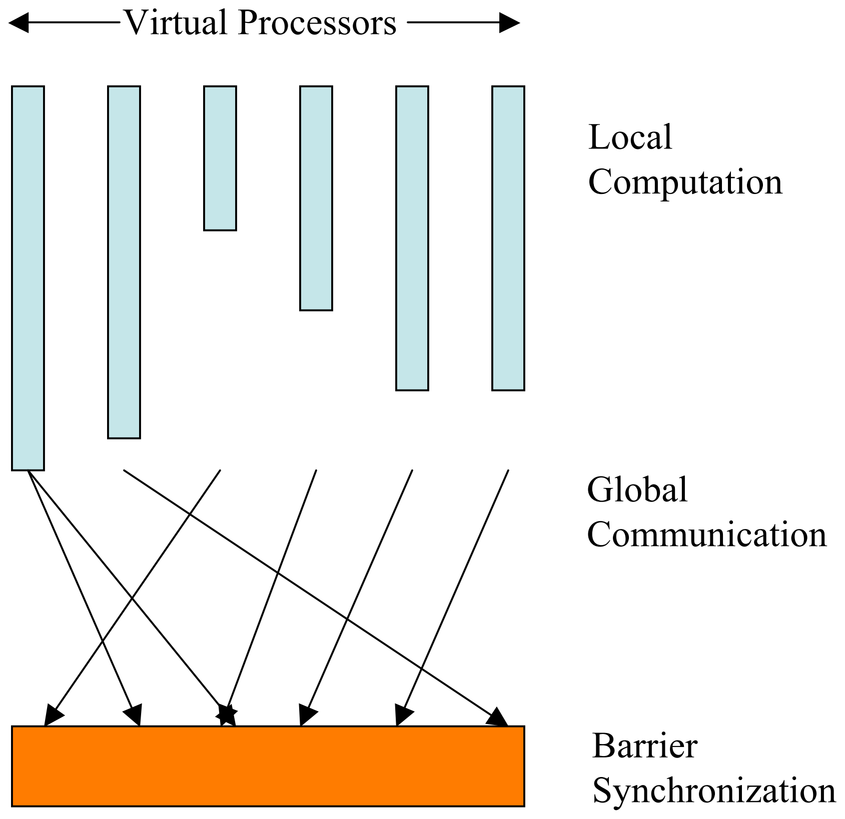
BSP Programs

- BSP programs composed of supersteps.
- In each superstep, processors execute up to L computational steps using locally stored data, and also can send and receive messages
- Processors synchronize at end of superstep (at which time all messages have been received)
- Oxford BSP is a library of C routines for implementing BSP programs. It provides:
  - Direct Remote Memory Access (a VSM layer)
  - Bulk Synchronous Message Passing (sort of like non-blocking message passing in MPI)



What does the BSP Programming Style Look Like?

- Vertical Structure
  - Sequential composition of "supersteps".
    - Local computation
    - Process Communication
    - Barrier Synchronization
- Horizontal Structure
  - Concurrency among a fixed number of virtual processors.
  - Processes do not have a particular order.
  - Locality plays no role in the placement of processes on processors.
  - p = number of processors.



BSP Programming Style

- Properties:
  - Simple to write programs.
  - Independent of target architecture.
  - Performance of the model is predictable.
- Considers computation and communication at the level of the entire program and executing computer instead of considering individual processes and individual communications.
- Renounces locality as a performance optimization.
  - Good and bad
  - BSP may not be the best choice for which locality is critical

How Does Communication Work?

- BSP considers communication *en masse*.
  - Makes it possible to bound the time to deliver a whole set of data by considering all the communication actions of a superstep as a unit.

- If the maximum number of incoming or outgoing messages per processor is *h*, then such a communication pattern is called an *h-relation*.

- Parameter *g* measures the permeability of the network to continuous traffic addressed to uniformly random destinations.
  - Defined such that it takes time *hg* to deliver an *h-relation*.

- BSP does not distinguish between sending 1 message of length *m*, or *m* messages of length 1.
  - Cost is *mgh*

Barrier Synchronization

- "Often expensive and should be used as sparingly as possible."
- Developers of BSP claim that barriers are not as expensive as they are believed to be in high performance computing folklore.
- The cost of a barrier synchronization has two parts.
  - The cost caused by the variation in the completion time of the computation steps that participate.
  - The cost of reaching a globally-consistent state in all processors.
- Cost is captured by parameter l ("ell") (parallel slackness).
  - lower bound on l is the diameter of the network.

Predictability of the BSP Model

- Characteristics:
  - p = number of processors
  - s = processor computation speed (flops/s) … used to calibrate g & l
  - l = synchronization periodicity; minimal number of time steps between successive synchronization operations
  - g = total number of local operations performed by all processors in one second / total number of words delivered by the communications network in one second

- Cost of a superstep (standard cost model):
  - $\text{MAX}(w_i) + \text{MAX}(h_i\, g) + l$ (or just $w + hg + l$)

- Cost of a superstep (overlapping cost model):
  - $\text{MAX}(w, hg) + l$

Predictability of the BSP Model

- Strategies used in writing efficient BSP programs:
  - Balance the computation in each superstep between processes.
    - "***w***" is a maximum of all computation times and the barrier synchronization must wait for the slowest process.
  - Balance the communication between processes.
    - "***h***" is a maximum of the fan-in and/or fan-out of data.
  - Minimize the number of supersteps.
    - Determines the number of times the parallel slackness appears in the final cost.

BSP Notes

- Number of processors in model can be greater than number of processors of machine.

  Easier for computer to complete the remote memory operations
- Not all processors need to join barrier synch
- Time for superstep = 1/s $\times$

  (max (operations performed by any processor)

  + g $\times$ max (messages sent or received by a processor, h_0)

  + L)

Some representative BSP parameters

Machine (all have P=8)	MFlop/s s	Flops/synch L	Flops/word g	words (32b) $n_{1/2}$ for h_0
Pentium II NOW switched Ethernet	88	18300	31	32
Cray T3E	47	506	1.2	40
IBM SP2	26	5400	9	6
Pentium NOW serial Ethernet 1	61	540,000	2800	61

From oldwww.comlab.ox.ac.uk/oucl/groups/bsp/index.html (1998)

NOTE: Benchmarks for determining s were not tuned.

BSPlib

- Supports a SPMD style of programming.
- Library is available in C and FORTRAN.
- Implementations available (several years ago) for:
  - Cray T3E
  - IBM SP2
  - SGI PowerChallenge
  - Convex Exemplar
  - Hitachi SR2001
  - Various Workstation Clusters
- Allows for direct remote memory access or message passing.
- Includes support for unbuffered messages for high performance computing.

BSPlib

- Initialization Functions
  - `bsp_init()`
    - Simulate dynamic processes
  - `bsp_begin()`
    - Start of SPMD code
  - `bsp_end()`
    - End of SPMD code
- Enquiry Functions
  - `bsp_pid()`
    - find my process id
  - `bsp_nprocs()`
    - number of processes
  - `bsp_time()`
    - local time
- Synchronization Functions
  - `bsp_sync()`
    - barrier synchronization
- DRMA Functions
  - `bsp_pushregister()`
    - make region globally visible
  - `bsp_popregister()`
    - remove global visibility
  - `bsp_put()`
    - push to remote memory
  - `bsp_get()`
    - pull from remote memory

BSPlib

- BSMP Functions
  - `bsp_set_tag_size()`
    - choose tag size
  - `bsp_send()`
    - send to remote queue
  - `bsp_get_tag()`
    - match tag with message
  - `bsp_move()`
    - fetch from queue
- Halt Functions
  - `bsp_abort()`
    - one process halts all
- High Performance Functions
  - `bsp_hpput()`
  - `bsp_hpget()`
  - `bsp_hpmove()`
  - These are unbuffered versions of communication primitives

BSPlib Examples

- Static Hello World

```
void main( void )
{
  bsp_begin( bsp_nprocs());

  printf( “Hello BSP from %d of %d\n”,
            bsp_pid(), bsp_nprocs());

    bsp_end();
}
```

- Dynamic Hello World

```
int nprocs;   /* global variable */
void spmd_part( void )
{
  bsp_begin( nprocs );
  printf( “Hello BSP from %d of %d\n”,
            bsp_pid(), bsp_nprocs());
}
void main( void )
{
  bsp_init( spmd_part, argc, argv );
  nprocs = ReadInteger();
  spmd_part();
}
```

BSPlib Examples

- Serialize Printing of Hello World (shows synchronization)

```
void main( void )
{
  int ii;
  bsp_begin( bsp_nprocs());
  for( ii=0; ii<bsp_nprocs(); ii++ )
  {
    if( bsp_pid() == ii )
      printf( “Hello BSP from %d of %d\n”, bsp_pid(), bsp_nprocs());
    fflush( stdout );
    bsp_sync();
  }
  bsp_end();
}
```

BSPlib Examples

- All sums version 1 (lg(p) supersteps)

```
int bsp_allsums1( int x ){
    int ii, left, right;
    bsp_pushregister( &left, sizeof( int ));
    bsp_sync();
    right = x;
    for( ii=1; ii<bsp_nprocs(); ii*=2 ){
        if( bsp_pid()+I < bsp_nprocs())
            bsp_put( bsp_pid()+I, &right, &left, 0, sizeof( int ));
        bsp_sync();
        if( bsp_pid() >= I )
            right = left + right;
    }
    bsp_popregister( &left );
    return( right );
}
```

BSPlib Examples

- All sums version 2 (one superstep)

```
int bsp_allsums2( int x ) {
      int ii, result, *array = calloc( bsp_nprocs(), sizeof(int));
      if( array == NULL )  bsp_abort( “Unable to allocate %d element array”, bsp_nprocs());
      bsp_pushregister( array, bsp_nprocs()*sizeof( int));
      bsp_sync();
      for( ii=bsp_pid(); ii<bsp_nprocs(); ii++ )
              bsp_put( ii, &x, array, bsp_pid()*sizeof(int), sizeof(int));
      bsp_sync();
      result = array[0];
      for( ii=1; ii<bsp_pid(); ii++ )  result += array[ii];
      free(array);
      bsp_popregister(array);
      return( result );
}
```

BSPlib vs. PVM and/or MPI

- MPI/PVM are widely implemented and widely used.
  - Both have ***<u>HUGE</u>*** API's!!!
  - Both may be inefficient on (distributed-)shared memory systems. where the communication and synchronization are decoupled.
    - True for DSM machines with one sided communication.
  - Both are based on pairwise synchronization, rather than barrier synchronization.
    - No simple cost model for performance prediction.
    - No simple means of examining the global state.

- BSP could be implemented using a small, carefully chosen subset of MPI subroutines.

Summary of BSP

- BSP is a computational model of parallel computing based on the concept of supersteps.
- BSP does not use locality of reference for the assignment of processes to processors.
- Predictability is defined in terms of three parameters.
- BSP is a generalization of PRAM.
- BSPlib has a much smaller API as compared to MPI/PVM.

Parameters of BSP Model

P = number of processors.

s = processor speed (steps/second).

observed, not “peak”.

L = time to do a barrier synchronization (steps/synch).

g = cost of sending message (steps/word).

measure g when all processors are communicating.

h_0 = minimum # of messages per superstep.

For $h \geq h_0$, cost of sending h messages is hg.

h_0 is similar to block size in PMH model.

LogP Model

- Developed by Culler et al from Berkeley
- Models communication costs in a multicomputer.
- Influenced by MPP architectures (circa 1993), notably the CM-5.
  - each node is a powerful processor with large memory
  - interconnection structure has limited bandwidth
  - interconnection structure has significant latency

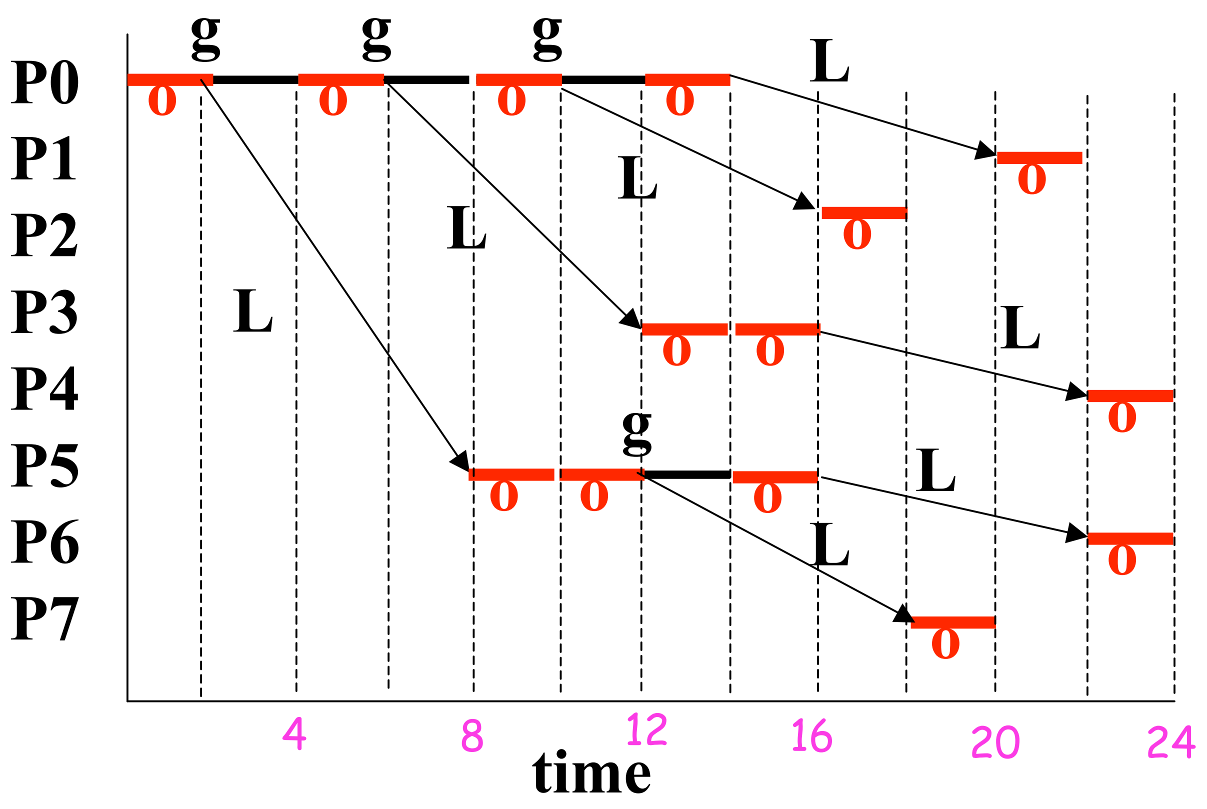
LogP parameters

- L: latency – time for message to go from P_{sender} to $P_{receiver}$
- o: overhead - time either processor is occupied sending or receiving message
  - Processor can't do anything else for o cycles.
- g: gap - minimum time between messages
  - Processor can have at most $\lceil L/g \rceil$ messages in transit at a time.
  - Gap includes overhead time (so overhead ≤ gap)
- P: number of processors

L, o, and g are measured in cycles

Efficient Broadcasting in LogP

Picture shows P=8, L=6, g=4, o=2



BSP vs. LogP

- BSP differs from LogP in three ways:
  - LogP uses a form of message passing based on pairwise synchronization.
  - LogP adds an extra parameter representing the overhead involved in sending a message. Applies to every communication!
  - LogP defines g in local terms. It regards the network as having a finite capacity and treats g as the minimal permissible *gap* between message sends from a single process. The parameter g in both cases is the reciprocal of the available per-processor network bandwidth: BSP takes a global view of g, LogP takes a local view of g.

BSP vs. LogP

- When analyzing the performance of LogP model, it is often necessary (or convenient) to use barriers.
- Message overhead is present but decreasing…
  - Only overhead is from transferring the message from user space to a system buffer.
- LogP + barriers - overhead = BSP
- Both models can efficiently simulate the other.

BSP vs. PRAM

- BSP can be regarded as a generalization of the PRAM model.
- If the BSP architecture has a small value of g (g=1), then it can be regarded as PRAM.
  - Use hashing to automatically achieve efficient memory management.
- The value of l determines the degree of parallel slackness required to achieve optimal efficiency.
  - If l = g = 1 … corresponds to idealized PRAM where no slackness is required.