Lecture 10

Coarse grain parallel computation models

BSP Model of Parallel Computation
LogP Model of Parallel Computation

Models of PP

- Main goal of parallel computation models is to provide a realistic representation of the costs of programming.
- Model provides algorithm designers and programmers a measure of algorithm complexity which helps them decide what is “good” (i.e. performance-efficient)

- We need a model that takes into account
  - Latency
  - Bandwidth
  - Communication Delay
  - Efficiency from coupling communication with computation

Fine versus coarse granularity

- PRAM- The most widely used parallel model
  - Overly simplistic in that it assumes:
    - Processors work synchronously
    - Inter-processor communication is not considered (analyzed)

  (considering classical variant)

- This leads to the creation of overly **fine grained** algorithms that in practice perform poorly since they spend so much time communicating/synchronizing

- Coarse grain models should be analyzed also…

Bulk Synchronous Parallelism

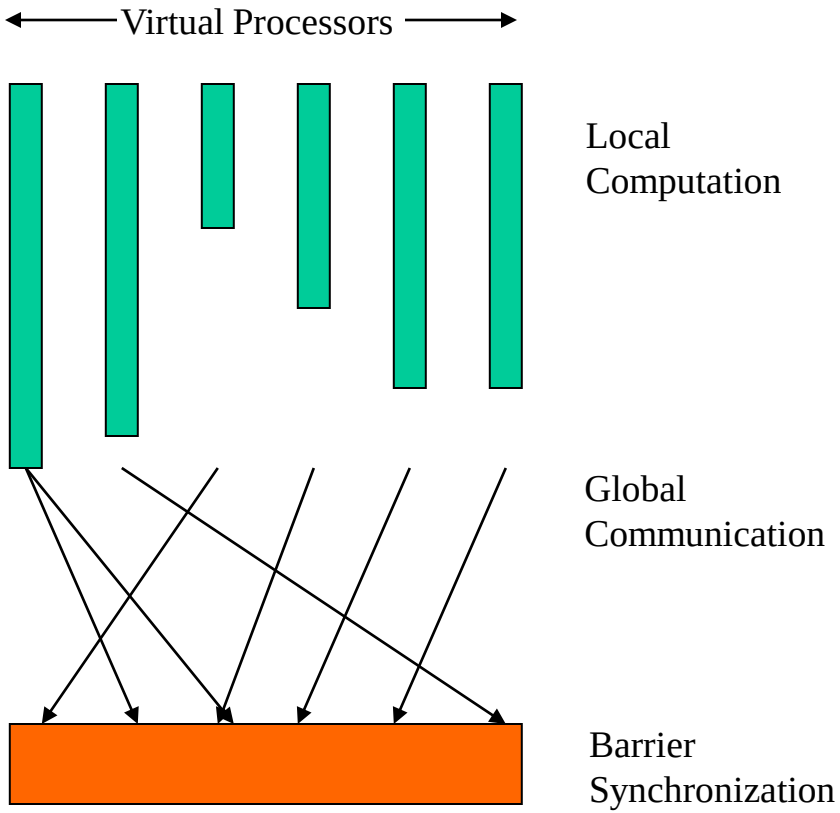
- BSP was proposed by L. Valiant as a ``bridging model'' in 1990 that provides a standard interface between parallel architectures and algorithms.

- BSP is a model of parallel computation
  - BSP computer
- BSP is a parallel programming model with coarse granularity

- BSP computer contains:
  - 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.

BSP Programming Style

- 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.



Locality – not very important

- the model only considers two levels of locality,
  - local (inside the processor) and
  - remote (outside a processor),

with remote access usually being more expensive than local ones.

when interconnection networks is considered into a model
the locality of a node is given by the set of its neighbors
- neighbors = nodes connected with maximum k edges (e.g. k=1)

efficient mappings should take into account locality

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 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 the programs/computation where the locality is critical – e,g. low-level image processing.

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*

Cost in the BSP Model

- Characteristics:
  - p = number of processors
  - s = processor computation speed (flops/s) … used to calibrate g & l

  (could be ignored by expressing the remaining parameters in its units)
  - l = synchronization periodicity; the time required to synchronize all processors
  - g = total number of words delivered by the communications network in one unity of time
  - w_i = work executed by the processor i
  - h_i = number of incoming or outgoing messages of processor i

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

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

parameter g

- The parameter g is related to the *bisection bandwidth* (=bandwidth available between two partitions of the network topology) of the communication network but it is not equivalent.
- It also depends on other factors such as
  - the protocols used to interface with and within the communication network
  - the buffer management by both the processors and the communication network
  - the routing strategy used in the communication network and
  - the BSP runtime system

Barrier Synchronization – parameter *l*

- It has been considered that it is:
  "*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 (=> idle times).
  - The cost of reaching a globally-consistent state in all processors.

- Cost is captured by parameter ***l*** (*parallel slackness*).
  - lower bound on l is the diameter of the network.

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.

Advantages

- Rather than considering individual processes and individual communication actions BSP consider computation and communication at the level of the entire program and executing computer

- Considering communication actions en masse both simplifies their treatment and makes it possible to bound the time it takes to deliver a whole set of data BSP does this by considering all of the communication actions of a superstep as a unit

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.

BSPlib vs. MPI

- MPI is widely implemented and widely used.
  - *huge* API's!!!
  - May be inefficient on (distributed-)shared memory systems. where the communication and synchronization are decoupled.
    - True for DSM machines with one sided communication.
  - 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.*

Conclusions

- ***It is simple to write BSP programs*** they look much the same as sequential programs. Only a bare minimum of extra information needs to be supplied to describe the use of parallelism.

- ***It is independent of target architectures:*** Unlike many parallel programming systems BSP is designed to be architecture independent, so that programs run unchanged when they are moved from one architecture to another. Thus, BSP programs are portable in a strong sense.

- ***The performance of a program on a given architecture is predictable.*** The execution time of a BSP program can be computed from the text of the program and a few simple parameters of the target architecture. This makes design possible since the effect of a decision on performance can be determined at the time it is made.

BSP implementation

- BSP Worldwide is an association
  - interested in the development of the Bulk Synchronous Parallel (BSP) computing model for parallel programming.
  - https://bsp-worldwide.science.uu.nl

  BSPlib standard

Oxford BSPlib toolkit – not sustained any more

- Supports a SPMD style of programming.
- Library is available in C and FORTRAN.
- Implementations were available 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.

,,

- **BSP Pro: a Java-based BSP performance profiling system**

Weiqun Zheng ; Sch. of Inf. Technol., Murdoch Univ., WA, Australia ;
Shamim Khan ; Hong Xie

- **JBSP: A BSP Programming Library In Java**

Yan Gu, Bu-Sung Lee, Wentong Cai , Johns Hopkins University

https://www.sciencedirect.com/science/article/abs/pii/S0743731501917356

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());
  bsp_end()
}
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();
}
```

AllSums: collective communication (i.e., all processes have to call the function), such that when process *i* calls the function with an array xs containing $\texttt{nelem}_i$ elements, then the result on *all* the processes will be the sum of all the arrays from all the processes.

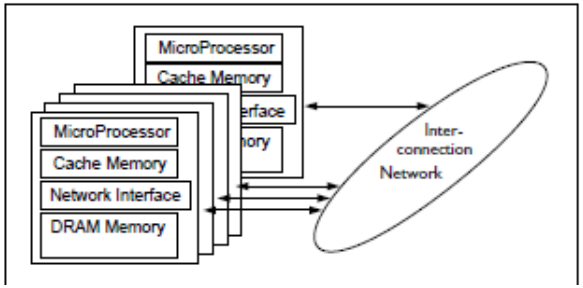
```
int bsp_sum(int *xs, int nelem) {
  int *local_sums,i,j,result=0;
  for(j=0;j<nelem;j++) result += xs[j];  //local sum
  bsp_push_reg(&result, sizeof(int));
  bsp_sync();
  local_sums = calloc(bsp_nprocs(), sizeof(int));
  if (local_sums==NULL)
    bsp_abort("{bsp_sum} no memory for %d int",bsp_nprocs());
  for(i=0;i<bsp_nprocs();i++)
    bsp_hpget(i,&result,0,&local_sums[i],sizeof(int));
//the result is taken from each proc
  bsp_sync();
  result=0;
  for(i=0;i<bsp_nprocs();i++) result += local_sums[i];
  bsp_pop_reg(&result);
  free(local_sums);
  return result;
}
```

LogP – a similar model for parallel computation

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.umd.edu/class/fall2019/cmsc714/readings/Culler-LogP.pdf

LogP machine model

- Model of distributed memory multicomputer
- Developed by [Culler, Karp, Patterson, &all.]
- Authors tried to model prevailing parallel architectures.
- David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: towards a realistic model of parallel computation. In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '93). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/155332.155333
- Machine model represents prevalent MPP(massive parallel programming) organization:
  - machine constructed with a few thousand nodes -
  - each node contains a powerful processor
  - each node contains substantial memory
  - interconnection structure has limited bandwidth
  - interconnection structure has significant latency



LogP – main parameters

- The main parameters of the model are:
  - L: Upper bound for the latency, or delay, incurred in communicating an empty message
  - o: Overhead, defined as the time the processor is engaged in sending or receiving a message, during which time it cannot do anything else
  - g: Gap, the minimum time interval between consecutive messages (so at most L/g messages can be in transit from any process or to any process at any time)
  - P: Number of processors

- The basic model assumes all messages are of small sizes.

Advantages of LogP

- LogP addresses Loop holes found in other models:
  - Interprocess communication: unlike PRAM, LogP does not hide the cost
  - Multithreading: LogP is able to express the upper bound of this technique (max L/g virtual processors)

- Encourages good practices like:
  - Coordinating work assignment (such that to reduce the communication bandwith)
  - Overlapping the computation and communication!!!

General analysis of Communication Closeup

Four main parts common to all communication:

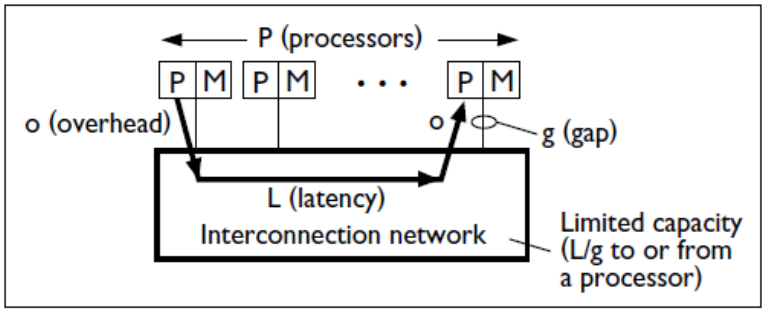
1. Tsnd = *Send Overhead*, time spent before the first bit can leave
2. Transition, time needed to get data onto the network
3. Transmission, time required for the bits to travel the network
4. Trcv=*Receive Overhead*, time after last bit arrives until the processor is free to work on something else

T(M, H) = Tsnd + M/w + Hr + Trcv

- M = size of message
- w = width of the network channel
- H = distance of the taken route
- r = delay through each node

Equation assumes lightly loaded networks

LogP – parameters



T(M, H) = Tsnd + M/w + Hr + Trcv

- M = size of message
- w = width of the network channel
- H = distance of the route taken
- r = delay through each node

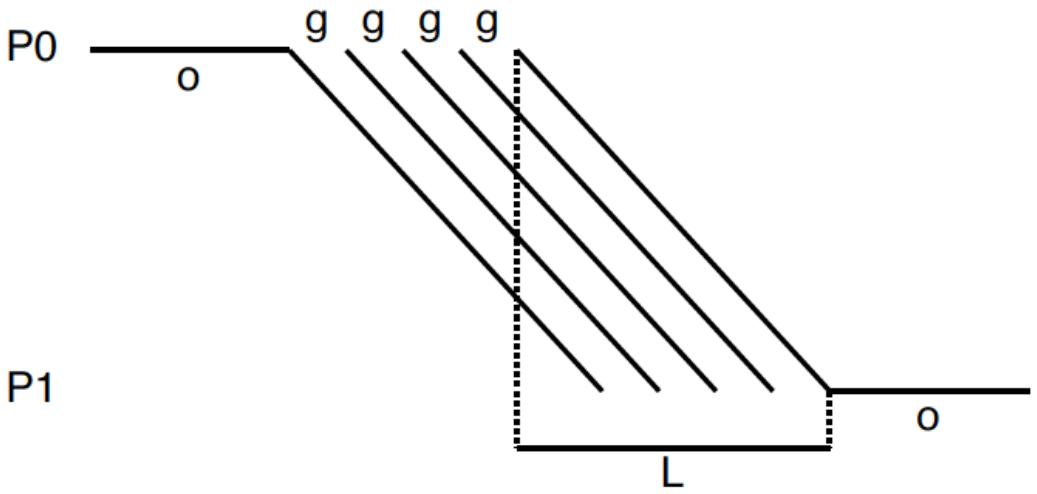
Equation assumes lightly loaded networks

Approx:

- o = *(Tsnd* + *Trcv*) / 2
- L = Hr + (M/w)
- g = M / (per processor bisection bandwith)
- P = Number of physical processors

Bisection width = minimum number of links cut to divide the network into two halves

Visualization



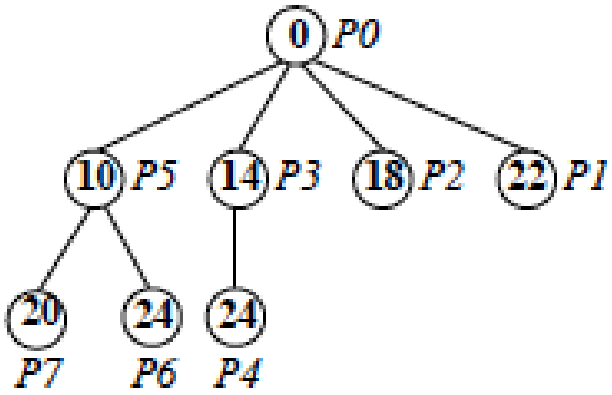
broadcasting a single datum from one processor to p-1 others

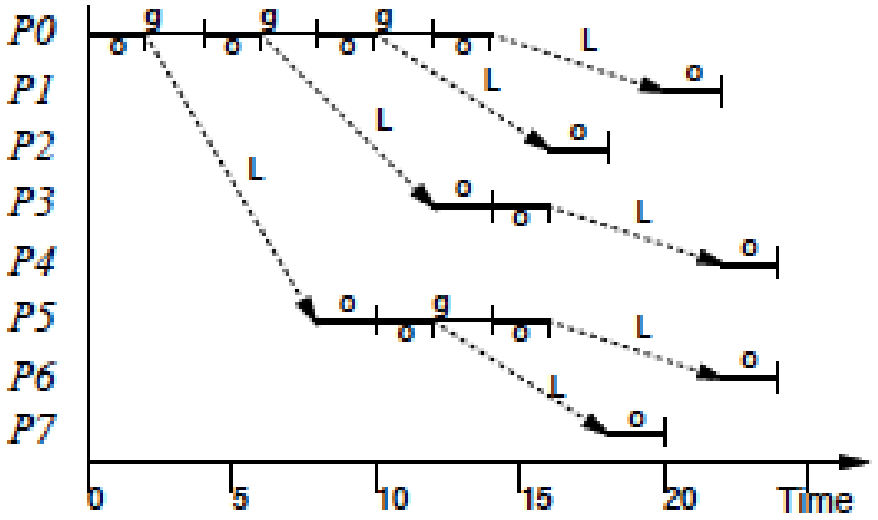
- Main idea:
  - all processors that have received the datum transmit it as quickly as possible, while ensuring that no processor receives more than one message.
- The source of the broadcast begins transmitting the datum at time 0.
- The first datum enters the network at time o , takes L cycles to arrive at the destination, and is received by the node at time L+ 2g
- Meanwhile, the source will have initiated transmission to other processors at time g, 2g, …

assuming g>=o, each of which acts as the root of a smaller broadcast tree.

Concrete example: which is the optimal broadcast tree for P=8 L= 6 g= 4 o= 2, and the activity of each processor over time

The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.





Notes:

- the optimal broadcast tree for processors is an unbalanced tree with the fan-out at each node determined by the relative values of L, o, g .
- The processor overhead of successive transmissions overlaps the delivery of previous messages.
- Nodes may experience idle cycles at the end of the algorithm while the last few messages are in transit.

Working with LogP

- **Short messages** (single message packet):

♦ 2o+L

- **Finite capacity of network**

♦ Ceil(L/g) messages in transit between any pair of nodes

- **Long messages**

♦ Pipeline of depth L with rate g and overhead o
(at each end)

• Depth L because it takes L units of time for message to travel through network and one message every g units of time. It is preferable that g = 1, but it might not.

Why Separate Latency and Overhead?

- Latency is Hardware – including time for data to traverse network
- Overhead is involvement of CPU
  - Significant difference between message passing (matching) and put/get (e.g., PGAS)
  - Message passing: receiver must find matching receive in a queue of posted but unmatched receives or save information on the message in a queue of unexpected messages
    - Overhead typically scales linearly with the number of messages in the queue

Linear algorithms fastest when queues nearly empty

What is the difference in distance (measured in clock cycles) between close and far nodes in large machines?

- **Nearby nodes** are less than 15cm apart
  - for 2GHz clock, that is 1 clock cycle
- **Far away nodes** may be
  2*100*30cm = 6000cm
- 6000cm/15cm/clock = 400 clock cycles
- Only 0.2 usec

The LogGP model introduces an additional parameter G used for long messages

Remark: speed of signal in wire < speed of light;
distance is minimum possible rather than typical

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.

Conclusion

- 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.
- BSP = LogP + barriers - overhead
- BSPlib has a much smaller API as compared to MPI

References

- Valiant, Leslie G., "A Bridging Model for Parallel Computation", Communications of the ACM, Aug., 1990, Vol. 33, No. 8, pp. 103-111
- "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
- The BSP Worldwide organization website is http://www.bsp-worldwide.org and an excellent Ohio Supercomputer Center tutorial is available at www.osc.org.
- "LogP: Towards a Realistic Model of Parallel Computation", PPOPP, May 1993

Presentation ref.

- Overview of BSP Model of Parallel Computation,

Michael C. Scherger,

  - Kent State University

- LogP: Towards a Realistic Model of Parallel Computation ☐

David Culler, Richard Karp, David Patterson,

Abhijit Sahay, Klaus Erik Schauser, Eunice Santos,

Ramesh Subramonian, and Thorsten von Eicken

  - Computer Science Division, University of California, Berkeley