# “Performance & Evaluation” (cont)

# Outline

- Techniques to increase speedup
- Performance evaluation tools
- Well-known performance models

# Factors that Determine Speedup

- Amount of sequential code (Amdahl's law)
- Amount of critical section code
- Characteristics of parallel code
  - granularity
  - load balance
  - locality
  - communication and synchronization

# Critical Section Code

- Not quite sequential
  - ?
- But still only one processor at a time in critical section
- Can result in contention for critical section lock

# Approach 1

- Minimize number of critical section accesses
- Goal: reduce contention
- General approach:
  - replace/approximate shared by local accesses
  - no need for critical section on local accesses

# Approach 1 cont.

```
computepi()
{
        h=1.0/n;
        sum =0.0;
        for (i=0;i<n;i++) {
                x=h*(i+0.5);
                sum=sum+4.0/(1+x*x);
        }
        pi=h*sum;
}
```

```
sum=0.0;
for(i=id;i<n;i=i+nprocs) {
        x=h*(i+0.5);
        sum=sum+4.0/(1+x*x);
}
localpi=sum*h;
use_tree_based_combining_for_critical_section();
        pi=pi+localpi;
end_critical_section();
```

# Approach 2

- Access shared data without critical section
- Dangerous, because of possible incorrectness
- Sometimes done to reduce overhead

# Approach 3

- Minimize amount of execution time inside critical section.
  - Minimize number of instructions.
  - Minimize number of data accesses, especially remote ones

# Approach 4

- Stagger accesses to critical sections in time
- Example: each process executes:

  Critical section A

  Critical section B

  Critical section C

  Critical section D

  …
- Solution?

# Factors that Determine Speedup

- Amount of sequential code (Amdahl's law)
- Amount of critical section code
- Characteristics of parallel code
  - granularity
  - load balance
  - locality
  - communication and synchronization

# Granularity

- Granularity = size of the program unit that is executed by a single processor
- May be a single loop iteration, a set of loop iterations, etc.
- Fine granularity pros & cons?

# Load Balance

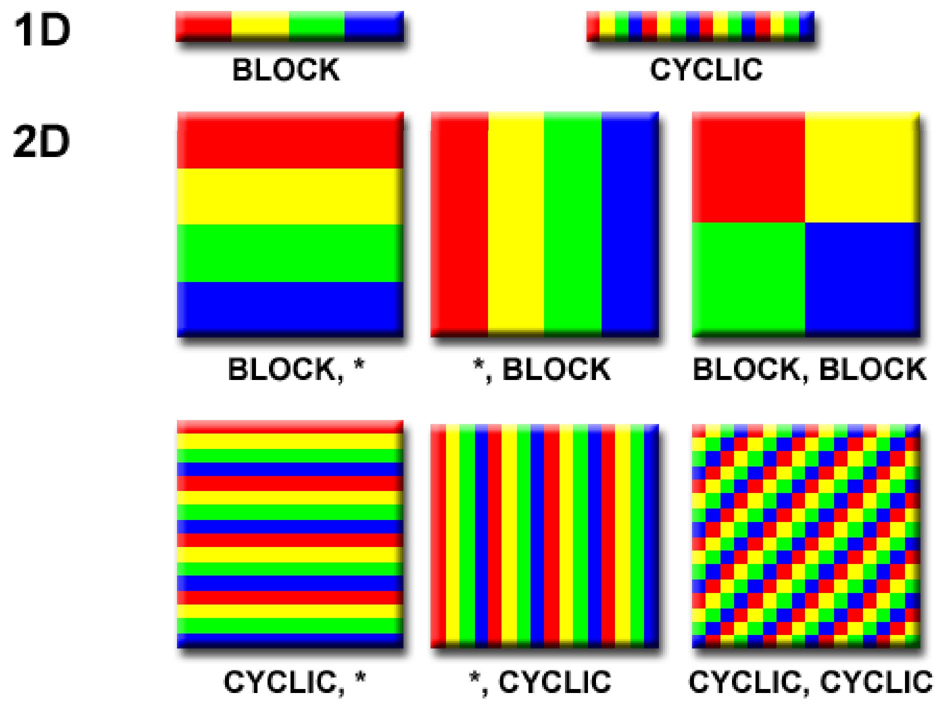
- Load imbalance = different in execution time between processors between barriers.

- **Static**: done once before computation
  - block, cyclic, etc.
  - fine for regular data parallel

- **Dynamic**: done at runtime
  - Centralized vs. distributed
  - fine for unpredictable execution
  - Usually high overhead

# Static Load Balancing

- Block
  - Pro & Con?
- Cyclic
  - Pro & Con?
- Block-cyclic
  - Pro & Con?



# Dynamic Load Balancing

- **Centralized**: single task queue.
  - Easy to program
  - Excellent load balance
- **Distributed**: task queue per processor.
  - Less communication/synchronization
- **Task stealing**:
  - Processes normally remove and insert tasks from their own queue
  - When queue is empty, remove task(s) from other queues
  - Discussion?

# Locality

- **Locality** (or re-use) = the extent to which a processor continues to use the same data or "close" data.

- ***Temporal* locality**: re-accessing a particular word before it gets replaced

- ***Spatial* locality**: accessing other words in a cache line before the line gets replaced

# Example 1

```
for( i=0; i<n; i++ )
   for( j=0; j<n; j++ )
        grid[i][j] = temp[i][j];
```

- Any locality?

# Example 2

```
for( j=0; j<n; j++ )
   for( i=0; i<n; i++ )
         grid[i][j] = temp[i][j];
```

- Locality?

# Example 3

```
for( i=1; i<n; i++ )
   for( j=1; j<n; j++ )
        temp[i][j] = 0.25 *
                     (grid[i+1][j]+grid[i+1][j]+
                     grid[i][j-1]+grid[i][j+1]);
```

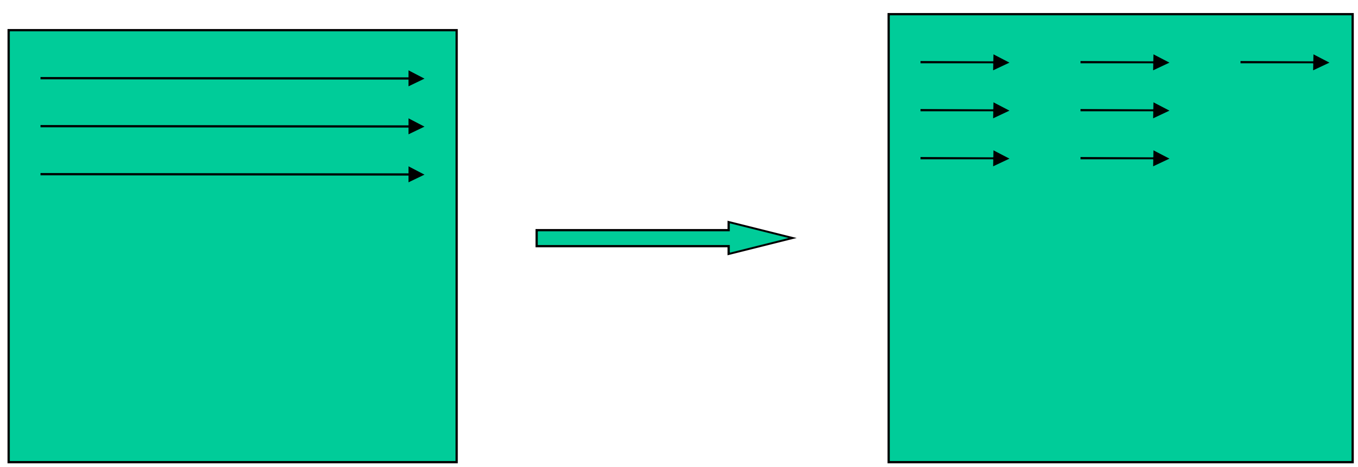
- Locality?

# Access to grid[i][j]

- First time grid[i][j] is used: temp[i-1,j]
- Second time grid[i][j] is used: temp[i,j-1]
- Between those times, 3 rows go through the cache
- If 3 rows > cache size, cache miss on second access

# Fix

- Traverse the array in blocks, rather than row-wise sweep
- Make sure grid[i][j] still in cache on second access

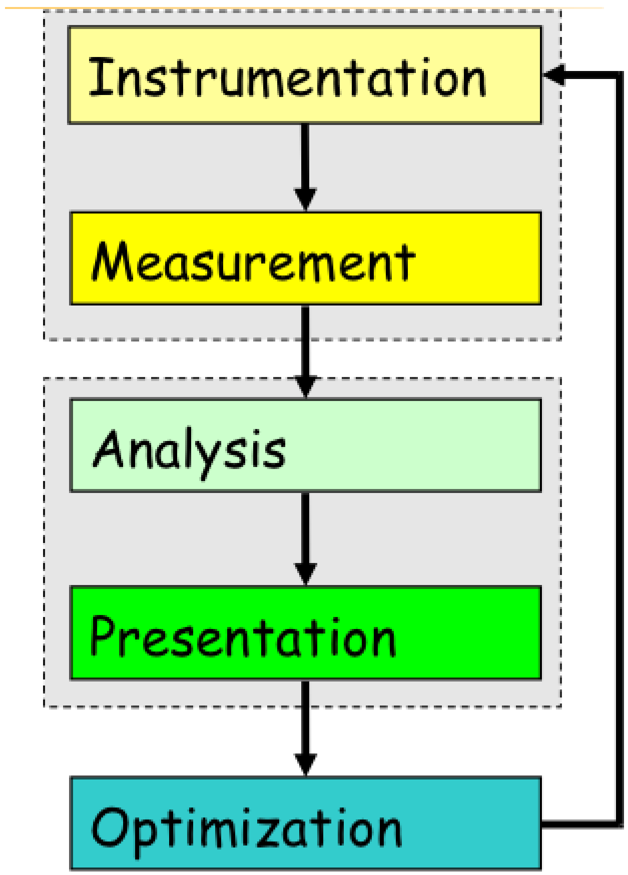


# Locality in Parallel Programming

- View parallel machine
  - not only as a multi-CPU system
  - but also as a multi-memory system
- Is even more important than in sequential programming, because the memory latencies are longer

# Performance Optimization

- Expose factors
- Collect performance data
- Calculate metrics
- Analyze results
- Visualize results
- Identify problems
- Turn performance
- This is an “art”



# Performance Questions

- How can we tell if a program is performing well?
- Or isn’t?
- If performance is not “good,” how can we pinpoint why?
- How can we identify the causes?
- What can we do about it?

# Performance Tuning

- The most important goal of performance tuning is to reduce a program's wall clock time
  - Reducing resource usage in other areas, e.g., memory or disk requirements or energy consumption, may also be a goal
- Performance tuning is an interactive process
  - Often involves finding your program's hot spots and eliminating bottlenecks
- Performance tuning usually involves profiling/tracing
  - Measure a program's runtime characteristics and resource utilization
- Use profiling/tracing tools to learn which areas of your code offer the greatest potential performance increase BEFORE you start the tuning

# Profiling

- Timing an entire program
  - UNIX *time* command outputs
    - user time
    - System time
    - Elapsed time
  - User time + system time = CPU time
  - Additional *time* output
    - Percent utilization
    - Average memory utilization
    - Blocked I/O operations
    - Page faults and swaps

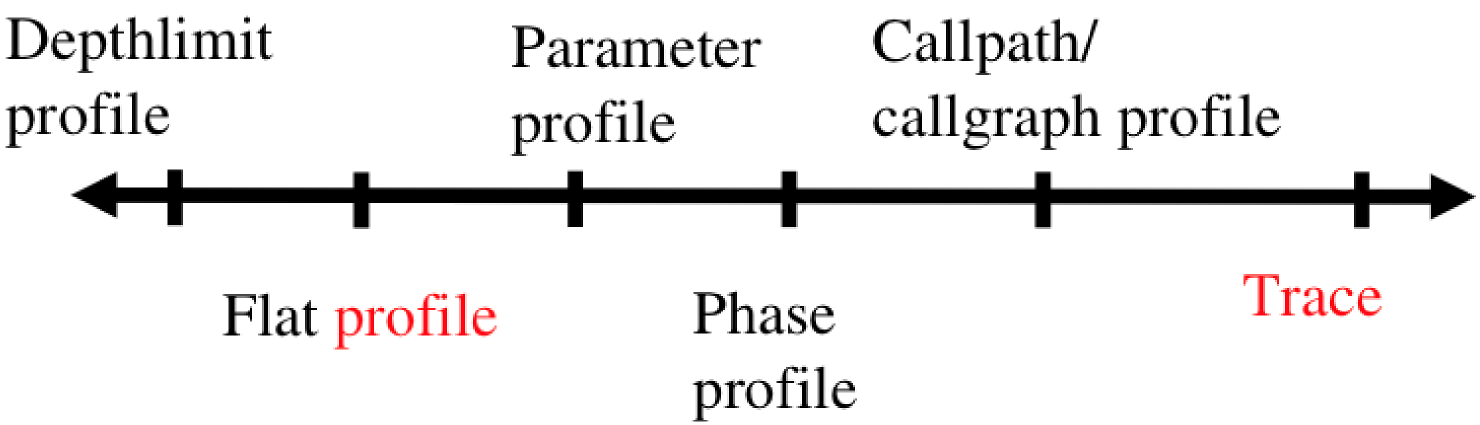
# Timing a Portion of a Program

- Record the time before you start doing x
- Do x
- Record the time at completion of x
- Subtract the start time from the completion time

# Types of Profiling

- Time-based
- Based on other metrics such as
  - Operation counts
  - Cache and memory event counts
- Types of Profile
  - Sharp profile
  - Flat profile

# Profiling & Tracing




Each alternative has:
- one metric/counter
- multiple counters

Volume of performance data

# Subroutine Profiling

- Most compilers provide a facility to automatically insert timing calls into your program at the entry and exit of each routine at compile time
- A separate utility (e.g. *prof, gprof*) produces a report showing the percentage of time spent in each routine
- Many performance analysis tools also provide this capability

# Gettimeofday()

- Part of the standard C library (libc.a) on most Unix/Linux systems
- Can be inserted anywhere within a C program and used to determine the start and end time of code fragments
- Timer resolution is hardware dependent

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

struct timeval start_time, end_time;

main()
{
   int total_usecs;

   gettimeofday(&start_time, (struct timeval*)0);

   /*   do some work */

   gettimeofday(&end_time, (struct timeval*)0);

   total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
                 (end_time.tv_usec-start_time.tv_usec);
   printf("Total time was %d uSec.\n", total_usecs);
}
```

# Profilers - prof

- Included in most Unix/Linux systems
- Can profile program execution at the procedure level

| Name | %Time | Seconds | Cumsecs | #Calls | msec/call |
|---|---|---|---|---|---|
| .fft | 51.8 | 0.59 | 0.59 | 1024 | 0.576 |
| .main | 40.4 | 0.46 | 1.05 | 1 | 460. |
| .bit_reverse | 7.9 | 0.09 | 1.14 | 1024 | 0.088 |
| .cos | 0.0 | 0.00 | 1.14 | 256 | 0.00 |
| .sin | 0.0 | 0.00 | 1.14 | 256 | 0.00 |
| .catopen | 0.0 | 0.00 | 1.14 | 1 | 0. |
| .setlocale | 0.0 | 0.00 | 1.14 | 1 | 0. |
| ._doprnt | 0.0 | 0.00 | 1.14 | 7 | 0. |
| ._flsbuf | 0.0 | 0.00 | 1.14 | 11 | 0.0 |
| ._xflsbuf | 0.0 | 0.00 | 1.14 | 7 | 0. |
| ._wrtchk | 0.0 | 0.00 | 1.14 | 1 | 0. |
| ._findbuf | 0.0 | 0.00 | 1.14 | 1 | 0. |
| ._xwrite | 0.0 | 0.00 | 1.14 | 7 | 0. |
| .free | 0.0 | 0.00 | 1.14 | 2 | 0. |
| .free_y | 0.0 | 0.00 | 1.14 | 2 | 0. |
| .write | 0.0 | 0.00 | 1.14 | 7 | 0. |
| .exit | 0.0 | 0.00 | 1.14 | 1 | 0. |
| .memchr | 0.0 | 0.00 | 1.14 | 19 | 0.0 |
| .atoi | 0.0 | 0.00 | 1.14 | 1 | 0. |
| .__nl_langinfo_std | 0.0 | 0.00 | 1.14 | 4 | 0. |
| .gettimeofday | 0.0 | 0.00 | 1.14 | 8 | 0. |
| .printf | 0.0 | 0.00 | 1.14 | 7 | 0. |

# Hardware Performance Counters

- Specialized registers to measure the performance of various aspects of a microprocessor
- Can be used to profile:
  - Whole program timing
  - Cache behaviors
  - Branch behaviors
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
  - Subroutine resolution
  - Process or thread attribution

# PAPI

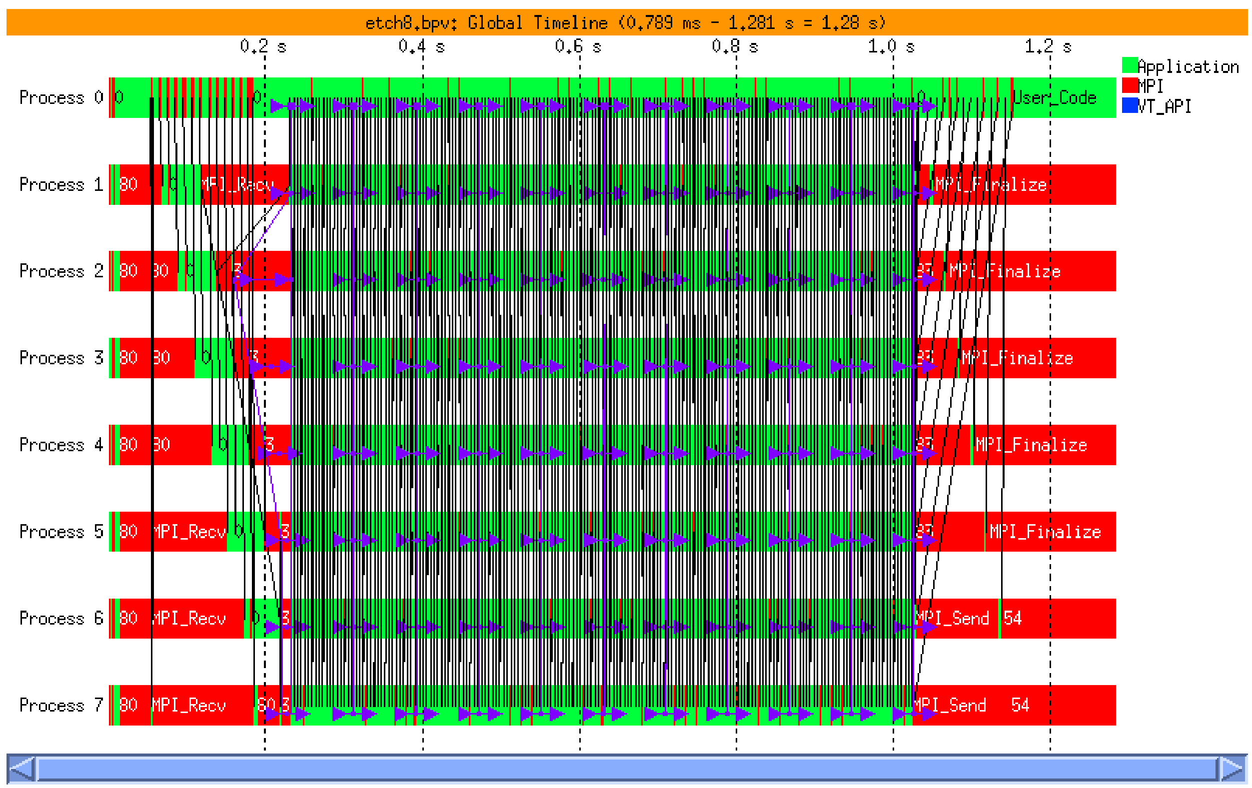
- PAPI (Performance API) provides a programming interface for accessing performance counters
  - http://icl.cs.utk.edu/papi/
- Countable events are defined in two ways
  - Platform-neural **Preset Events**
    - Standard set of over 100 events
    - Use papi_avail to see what preset events are available on a given platform
  - Platform-dependent **Native Events**
    - Any event countable by the CPU
    - Use papi_native_avail to see alll available native events
  - Use papi_event_chooser to select a set of events

# PAPI Example

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_OPS,PAPI_TOT_CYC},
int EventSet;
long long values[NUM_EVENTS];

/* Initialize the Library */
retval = PAPI_library_init (PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset (&EventSet);
/* Add Flops and total cycles to the eventset */
retval = PAPI_add_events (&EventSet,Events,NUM_EVENTS);

/* Start the counters */
retval = PAPI_start (EventSet);

do_work();  /* What we want to monitor*/

/*Stop counters and store results in values */
retval = PAPI_stop (EventSet,values);
```

# Tracing Tools

- The program is monitored while it is executed
- Monitoring produces performance data (TRACE) that is interpreted in order to reveal areas of poor performance
- Tracing tools
  - Vampir
  - Paraver



# TAU

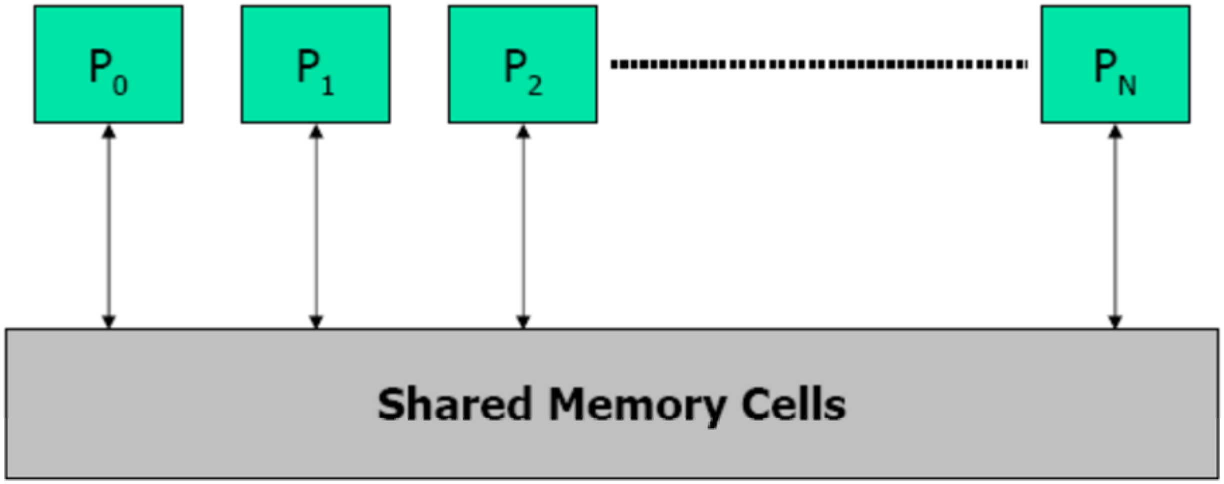
- Tuning and analysis utilities
  – http://www.cs.uoregon.edu/research/tau/home.php
- Integrated toolkit for performance tuning
  – Portable performance profiling and tracing
  – Instrumentation, measurement, analysis, and visualization
  – Supports several measurement options (profiling, tracing, profiling with hardware counters, etc.)
  – Can show how much time was spent in each routine
  – Can automatically instrument your source code
- To use it, you need to set a couple of environment variables and substitute the name of your compiler with a TAU shell script

# Well-Known Performance Models

- PRAM
- BSP
- AB
- LogP

# PRAM

- Parallel Random Access Memory
- Idealized abstraction of parallel systems




- Properties
  - SIMD (all processors perform the same op in a cycle)
  - Polynomial no. of processors
  - Polynomial amount of shared memory
  - Uniform latency op: read, write, compute
- Drawbacks?

# PRAM Variants

- Exclusive Read Exclusive Write (EREW)
  - At most one processor can read or write any memory cell in a step
- Concurrent Read Exclusive Write (CREW)
  - Any processor can read any location
  - Only one processor may write any one memory cell in a step
  - Admits a large class of algorithms
- Concurrent Read Concurrent Write (CRCW)
  - Each processor can read or write any one memory cell in a step
    - No consideration of memory contention
    - Variants depend on handling of write collisions
      - Common: assumes that all competing processor write the same value
      - Arbitrary: one arbitrary processor's write succeeds; all others fail
      - Priority: write by highest priority processor succeeds; all others fail
- Queue Read Queue Write (QRQW)
  - Permits concurrent reads and writes to shared memory locations

# BSP Model

- Bulk Synchronous Parallel Model
- Execution as series of supersteps
  - In one superstep, a processor
    - Sends limited no. of messages
    - Performs local computation
    - Receives all messages
    - Performs a global barrier
- Efficient BSP algorithms
  - Overlap comm and comp
  - Comm bandwidth and latency can be ignored
- Strengths
  - Simple enough to be used for design of portable algs
  - Enable designer to address key performance issues for algs
  - Evaluate algs using machine performance characteristics

# Latency and Bandwidth Model

- Time to send message of length n is roughly

$$\text{Time} = \text{latency} + n*\text{cost\_per\_word}$$
$$= \text{latency} + n/\text{bandwidth}$$

- Topology is assumed irrelevant
- Often called "$\alpha$ -$\beta$ model" and written

$$\text{Time} = \alpha + n*\beta$$

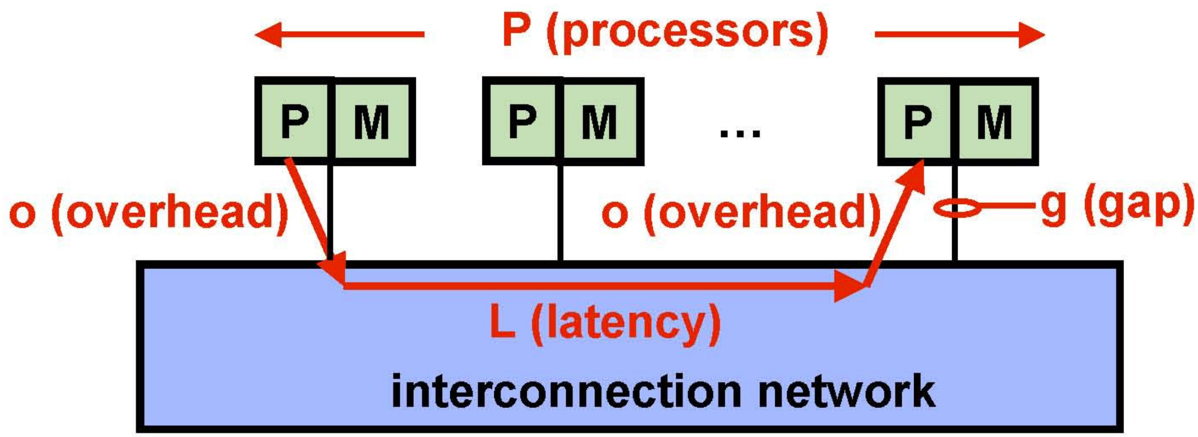
- Usually $\alpha$ >> $\beta$ >> time per flop
  - One long message is cheaper than many short ones

$$\alpha + n*\beta \ll n*(\alpha + 1*\beta)$$

  - Can do hundreds or thousands of flops for cost of one message
- Useful lesson from the model?

# LogP Model

- Abstract machine model




- Four parameters
  - L: latency experienced in each comm event
    - Time to communicate word or small # of words
  - o: send/recv overhead experienced by processor
    - Time processor fully engaged in transmission or reception
  - g: gap between successive sends or recvs by a processor
    - 1/g = comm BW
  - P: no. of processor/memory modules

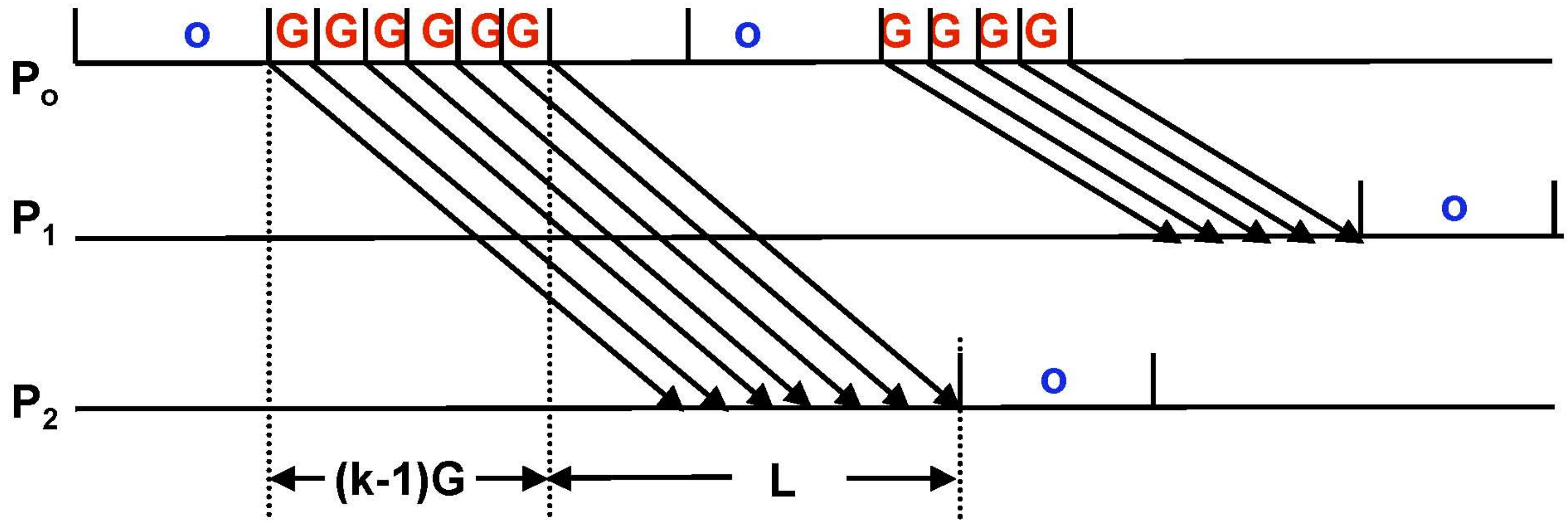
# Characteristics of LogP

- Asynchronous processors that work independently
- Messaging assumptions
  - All messages of small fixed size
  - Network has finite capacity
    - ≤ ceiling(L/g) messages in transit from p to q at once
    - Attempting to transmit more causes processor to stall
- No topology considerations
  - Assumes a fully connected network
- Unpredictable msg latency
  - Bounded from above by L in absence of stalls
- Notable missing aspect: local computation
  - Does not model local computation
  - Ignores cache size, pipeline structure

# LogGP: Account for Long Msgs

- Motivation
  - LogP: predicts performance for fixed-size short msgs only
  - Modern machines support long msgs with higher BW
- Goal: model performance with both short and long msgs
- Modeling long msgs:
  - Transmission time: $t=t_o+t_B*n$
  - Insufficiently detailed for short msgs
- LogGP
  - Extend LogP with additional parameter G
    - G=gap per byte for long msg – time per byte for long msg
    - 1/G=BW for long msg

# LogGP




- Sending a small msg: 2o+L cycles
  - o cycles on sender + L cycles in network + o cycles in recv
- Under LogP, sending k bytes msg requires

$$\text{time}=\text{o}+\lceil (\text{k-1})/\text{w} \rceil \times \max(g, o) + L + o \text{ cycles}$$

- Under LogGP

$$\text{time}=\text{o}+(\text{k-1})*\text{G} + L + o \text{ cycles}$$

# Discussion of LogGP

- o captures time main processor is involved in sending/recving
- G reflects network BW for long msgs
- g captures startup bottleneck of network
  – E.g., time for comm co-proc to open comm channel
- Simplified models
  – For short msgs only: LogGP reduces to LogP
  – For very long msgs only: approximate xfer-time as kG
- Impact on algorithm design
  – Aggregate short msgs into long msgs for higher BW

# References


- D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K. E. Schauser, R. Subramonian, T. von Eicken. *LogP: A practical model of parallel computation. Communications of the ACM,* 39(11):78 - 85, November 1996.
- A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. *LogGP: Incorporating long messages into the LogP model. JPDC,* 44(1):71–79, 1997.
- P.B. Gibbons, Y. Matias, V. Ramachandran. *The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms. SIAM J. Computing 28(2):733-769, 1998.*
- Bell et al. *An evaluation of current high-performanc networks, IPDPS, Nice, France, April 2003.*

# Summary

- Techniques to increase speedup
- Performance evaluation tools
- Well-known performance models