Programming with Processes

Vlad Vlassov
Dept of Microelectronics and Information Technology (IMIT)
Royal Institute of Technology (KTH), Stockholm, Sweden

2nd Estonian Summer School on Computer and Systems Science (ESSCSS'03)
Taagepera, Estonia, August 2003

Contents

• Introduction
• Concurrent programming concepts, models, and paradigms
• Shared memory programming
  – Processes and synchronization.
  – Introduction to the formal semantics of concurrent programs with shared variables
  – Synchronization mechanisms: Locks and barriers; Condition variables; Semaphores; Monitors
• Distributed memory programming
  – Message passing, RPC, Rendezvous, RMI
  – Paradigms for process interaction.
• Parallelism in scientific computing
• Case studies: Pthreads, MPI (Message Passing Interface)
References (Books)


Sequential Program Versus Parallel Program

- Sequential program
  - Represents a sequence of actions that produces a result
  - Contains a single thread of control
  - Is called a *process* or a task
- Parallel (concurrent, distributed) program
  - Contains two or more processes that cooperate in solving a problem
    - Each process executes a sequential program and has its own state
    - Processes communicate via shared memory or/and by message passing
  - Can be executed
    - Concurrently on a single-processor machine or
    - In parallel on a multiprocessor or on several computers.
Programming with Processes

• **Concurrent Programming**
  – Several concurrent processes (threads) share a single CPU
  – Shared memory programming model
  – Multithreading for performance and scalability

• **Distributed Programming**
  – Processes distributed among computers communicate over network
  – Message passing programming model
  – For distributed computing: Users, data, machines are distributed
  – For high performance (scalable) computing

• **Parallel Programming**
  – Parallel (concurrent) processes execute on their own processors
  – Both, shared memory and message passing
  – For high performance computing: solve a problem faster or/and solve a larger problem

Speedup

• Major goal of applications in using a parallel machine is *speedup which is offered* by the parallel machine:

\[
\text{Speedup (n processors)} = \frac{\text{Performance (n processors)}}{\text{Performance (1 processor)}}
\]

• Major goal of applications in being executed in parallel is *speedup which can be achieved* due to parallel execution:
  • for a fixed problem size (input data set), performance = 1/time:

\[
\text{Speedup (fixed problem size, n processors)} = \frac{\text{Execution time (1 processors)}}{\text{Execution time (n processors)}}
\]
Limited Concurrency

• Amdahl’s Law:
  – If fraction $s$ of sequential execution is inherently serial, then speedup $\leq 1/s$
• Impediments to speedup
  – Inherently sequential parts (Amdahl’s law) – unavoidable
  – Load imbalance
  – Synchronization and context switch overheads: critical sections, delays, fork/JOIN

Hardware

• For multiprogramming and concurrent programming with shared memory
  – A single-CPU processor
• For concurrent and parallel programming with shared memory
  – Shared-memory multiprocessor
    • Centralized shared memory (UMA)
    • Distributed shared memory (NUMA)
• For distributed and parallel programming with message passing
  – Distributed memory multiprocessor
  – Computer clusters
  – Computer networks
• For distributed and parallel programming with message passing
  – **Hierarchical multiprocessor**: distributed memory MP with SMP nodes
Parallel Programming Concepts

- **Task** – an arbitrary piece of un-decomposed work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - Fine-grained versus coarse-grained tasks
- **Process (thread)** – an abstract entity that performs tasks assigned to processes
  - Each process has its state and a unique ID
  - Processes communicate and synchronize to perform their tasks via shared memory or/and by message passing
  - Three types of processes:
    - Heavy-weight processes
    - Light-weight processes
    - Threads

Parallel Programming Concepts (cont’d)

- A **process state (context)** includes all information needed to execute the process:
  - In CPU: contents of PC, NPC, PSW, SP, registers
  - In memory: contents of text segment, data segment, heap, stack
    - Can be cached in data and instruction caches
    - A portion (but not stack) can be shared with other processes
- A **context switch** – terminate or suspend the current process and start or resume another process
  - Performed by an OS kernel
  - The CPU state of the current process must be saved to the memory
  - “Dirty” cache lines can be written back to memory on replacement
Parallel Programming Concepts (cont’d)

- **Processor** – a physical engine on which process executes
  - Processes virtualize machine to programmer
    - First write program in term of processes, then map to processors
- **Concurrent program (execution)** is formed of several processes (threads) that share a (multi)processor
  - Usually more processes than processors
- **Parallel program (execution)** is formed of several processes each executes on its own processor
  - Usually more processors than processes
- **Distributed program (execution)** is formed of processes that are distributed among processors and communicate over network
  - No shared memory

Heavy-Weight Processes (HWP)

- A *heavy-weight process* has its own virtual address space not shared with other HWPs
  - Entire process state is private
  - A HW process can be multithreaded
  - A HWP can create another HWP (a copy of itself or a different process)
- Example: UNIX processes
  - `fork` creates a new process (child) which is an exact copy of the parent
  - `fork` followed by `exec` (or `reexec`) is used to start a completely different (detached) process
**Threads**

- A *thread* is essentially a program counter, an execution stack, and a set of registers — thread context.
  - All the other data structures and the code belong to a HWP where threads are created, and are shared by the threads.
  - Each thread is assigned a unique thread ID.

- Example: Pthreads – POSIX (IEEE Portable OS Interface) threads
  - `pthread_create` creates a new thread
    - The thread executes a given function, has its own stack and registers, but share global variables with other threads.
  - Threads can be “joined” by `pthread_join` in parent and `return` or `pthread_exit()` in child

---

**Process versus Thread**

![Diagram](a Process diagram)

![Diagram](b Threads diagram)
Four Steps in Creating a Parallel Program

(1) Decomposition; (2) Assignment; (3) Orchestration; (3) Mapping
- Done by programmer or system software (compiler, runtime, ...).
- The programmer does it explicitly

Parallel Programming Models

- A programming model defines how processes communicate and synchronize
- **Shared memory programming model**, a.k.a. Shared address space (SAS) model.
  - Portions of virtual address spaces of processes are shared (common)
  - Processes communicate via shared memory (common variables) by conventional reads and writes
  - An imperative language, e.g. C or Java, requires explicit synchronization mechanisms such as locks, barriers, semaphores, monitors.
- **Distributed memory programming model**, a.k.a. Message passing programming model
  - No shared memory, only communication channels are shared
  - Processes communicate by sending and receiving messages over communication channels.
  - Synchronization is implicit: a message to be received must be sent
Existing Synchronization and Communication Mechanisms

- Synchronization mechanisms – for shared memory programming
  - Locks
  - Barriers
  - Condition variables
  - Semaphores
  - Monitors

- Communication mechanisms – for distributed memory programming
  - Asynchronous message passing
  - Synchronous message passing (e.g. CSP: Communicating Sequential Processes)
  - Remote procedure call (RPC) and Rendezvous
  - Remote method invocation (RMI)

Programming Notation Used Here

- A subset of the SR (Synchronizing Resources) language similar to C
- Declarations – much like in C and Matlab
  - Primitive types (int, double, char, etc.)
  - Arrays: int c[1:n], double c[n, n]
  - Process declaration process – start one or several processes in background (detached thread)

- Statements:
  - Assignment statement
  - Control flow statements: if, while, for – much like in C
    - for with quantifiers:
      for [i=0 to n-1, j = 0 to n-1] ...;
      for [i=1 to n by 2] ...; # odd values from 1 to n
      for [i=1 to n at i=i] ...; # every value except zero; “at” stands for “such that”
  - Concurrent statement (co-statement) – starts two or more threads in parallel and then waits until all the threads complete
  - Await statement – used for synchronization – will be introduced later
Two Forms of co-Statement

1. Different statements in parallel arms
   \( S_0; \)
   \( \text{co } S_1; \ # \text{ thread 1} \)
   // ... // \( S_n; \ # \text{ thread } n \)
   \( \text{oc}; \)
   \( S_{n+1}; \)

2. With quantifiers: Same list of statements in parallel arms
   (for every combination of variables in quantifiers)
   \( S_0; \)
   \( \text{co} \{ i=1 \ \text{to } n, \ j=1 \ \text{to } n \} \{ \ # \text{ n x n threads} \)
   \( S(i, j); \)
   \} \)
   \( \text{oc}; \)
   \( S_{n+1}; \)

process Declaration

• Syntax is similar to co but with one arm and /or one quantifier
• Start a single process in background
  ```
  process p
  { body }
  ```
• Start array of processes in background
  ```
  process p[quantifier]
  { body }
  ```
• May declare local variables and access global variables
• Can appear where procedure declarations can appear
• Forked when its declaration is encountered
• No synchronization upon termination – detached processes
1. Iterative Parallelism

- Parallelism of \textit{independent iterations}.
- An iterative program uses loops to examine data and compute results. Some loops can be parallelized.
- Example: Matrix multiplication $C = A \cdot B$
  
  \hspace{1cm} \begin{minipage}{0.4\textwidth}
  \begin{verbatim}
  Double a[n,n], b[n,n], c[n,n];
  for [i=0 to n-1] {
    for [j=0 to n-1] {
      c[i,j]=0.0;
      for [k=0 to n-1]
        c[i,j]= c[i,j]+a[i,k]*b[k,j];
    }
  }
  \end{verbatim}
\end{minipage} \hspace{1cm} \begin{minipage}{0.4\textwidth}
  \begin{verbatim}
  Double a[n,n], b[n,n], c[n,n];
  co [i=0 to n-1] {
    for [j=0 to n-1] {
      c[i,j]=0.0;
      for [k=0 to n-1]
        c[i,j]= c[i,j]+a[i,k]*b[k,j];
    }
  } co;
  \end{verbatim}
\end{minipage}

- Replace \texttt{for} with \texttt{co}
- \texttt{n} threads in \texttt{co} are executed concurrently for different values of \texttt{i}
2. Recursive Parallelism

- Parallelism of *independent recursive calls*
  - A recursive procedure calls itself *more than once* in its body.
  - If the calls are independent, they can be executed in concurrent threads
- Examples: Quick sort, adaptive quadrature
  - “Divide and conquer” (domain decomposition)
    - Split a data region (e.g. list, interval) into several sub-regions to be processed recursively using the same algorithm

Example: The Quadrature Problem

- Compute approximation of the integral of a continues function $f(x)$ from $a$ to $b$

  - *Sequential iterative quadrature program*
    - using trapezoidal method:

    ```
    double fl = f(a), fr, area = 0.0;
    double dx = (b-a)/ni;
    for [x = (a + dx) to b by dx] {
        fr = f(x);
        area = area + (fl + fr) * dx / 2;
        fl = fr;
    }
    ```
Recursive Adaptive Quadrature Procedure

- **Sequential procedure:**

  ```java
  double quad(double l, r, fl, fr, area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
    larea = quad(l, m, fl, fm, larea);
    rarea = quad(m, r, fm, fr, rarea);
  }
  return (larea+rarea);
  }
  }
  ```

- **Parallel procedure:**

  ```java
  double quad(double l, r, fl, fr, area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
    larea = quad(l, m, fl, fm, larea);
    // rarea = quad(m, r, fm, fr, rarea);
  }
  return (larea+rarea);
  }
  ```

- Two recursive calls are independent and can be executed in parallel
- **Usage:**
  ```java
  area = quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2)
  ```

3. Producers and Consumers. Pipes

- Parallelism of production (of next data) and consumption (of previous data)
  - One-way data stream between Producer and Consumer
  - “Filters” can be placed in between
  - Processes can be organized in a pipeline
    - Parallelism of pipeline stages
    - Each consumes the output of predecessor and produces the input for its successor – true data dependence between stages
    - Data buffers (FIFO queues) are placed between processes

```
Producer → Filter_{f_1} → … → Filter_{f_n} → Consumer
```
4. Clients and Servers

- Parallelism of client and server processes
  - Client requests a service
  - Server provides the service
  - Two-way communication: request – reply pairs
- Parallelism in servicing of multiple clients in separate threads
  - Multithreaded servers. Synchronization might be required
- Implemented
  - Distributed-memory: using message passing, RPC, rendezvous, RMI
  - Shared-memory: using subroutines, monitors etc.
- Example: (Distributed) file systems
  - open, read, write, close – client requests
  - Acknowledgements – server replies

5. Interacting Peers

- Parallelism of “equal” peers
  - Each execute the same set of algorithms and communicate with others in order to achieve the goal
- Configurations
  - Grid
    - Master (coordinator) and slaves (workers)
    - Roles may change
  - A circular pipeline
  - Each to each
  - Mesh
  - Arbitrary

(a) Coordinator/worker interaction

(b) A circular pipeline
Shared Memory Programming

- Processes and synchronization.
- Introduction to the formal semantics of concurrent programs with shared variables.
- Synchronization mechanisms: Locks and barriers; Condition variables; Semaphores; Monitors

Processes and Synchronization
Process State. Actions. Process History

- **Process (thread)** is an abstract entity that performs tasks assigned to processes.
- **Process state** is formed of values of variables at a point in time.
- Each process executes a sequence of statements.
- Each statement consists of one or more *atomic (indivisible)* actions which transform one state into another.
  - Some actions allow processes to communicate with each other.
- Sequence of states makes up the process history.
- The **process history** is a trace of ONE execution.

\[ P: Q_0 \rightarrow Q_1 \rightarrow \ldots \rightarrow Q_m \]

Atomic Actions

- **Atomic action** – indivisible sequence of state transitions made atomically.
- Fine-grained atomic actions
  - Machine instructions (read, write, swap, etc.) – atomicity is guaranteed by HW.
- Coarse-grained atomic actions
  - A sequence of fine-grained atomic actions executed indivisibly (atomically).
    - Internal state transitions are not visible “outside”.
    - For example, a critical section of a code.
- Notation: `< statements >` – a list of statements to be executed atomically.
Interleaving Semantics of Concurrent Execution

• The concurrent execution of several processes can be viewed as the interleaving of histories of the processes
  – i.e., the interleaving of sequences of atomic actions of different processes.

• Individual histories:
  – Process 1: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$
  – Process 2: $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n$
  – Process 3: $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_n$

• Interleaved concurrent histories:
  – Trace 1: $s_0 \rightarrow p_0 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_0 \rightarrow s_2 \rightarrow q_1 \rightarrow \ldots$
  – Trace 2: $p_0 \rightarrow s_0 \rightarrow q_0 \rightarrow q_1 \rightarrow s_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2 \rightarrow \ldots$

Nondeterminism of Concurrent Execution

• The behavior of a concurrent program is not reproducible because different interleavings (histories) can be observed on different concurrent executions.
  – Each process executes at its own rate
  – Multiprogramming causes context switching

• A concurrent program of $n$ processes each of $m$ atomic actions can produce $(n \times m)! / (m!)^n$ different histories.
  – For example, $n = 3, m = 2$ give 90 different histories.

• Impossible to show the correctness of a program by testing (“run the program and see what happens”).
**Synchronization**

- Synchronization is a mechanism to delay a process until it may proceed
  - Allows reducing the entire set of possible histories to those which are desirable (correct).
    - To preserve (true) dependencies between processes
    - To avoid race conditions if any
- Two kinds of synchronization:
  - *Mutual exclusion*
    - Guarantees that only one process executes its critical section at a time
  - *Condition synchronization*
    - Delays a process until a certain condition is true.

---

**Specifying Synchronization: The Await Statement**

```
< await(B) S; >
```
- Wait for B to be true, then execute S atomically
- Combines condition synchronization with mutual exclusion
  - Executed as an atomic action
  - B is guaranteed to hold when S begins
  - S is guaranteed to terminate

```
< await(B); >
```
- Wait for a condition B to be true.
- Used to express condition synchronization

```
< S; >
```
- Execute a list of statements S atomically (indivisibly).
- Can be used to express mutual exclusion
The At-Most-Once Property

• Unnecessary explicit coarse-grained atomicity causes unnecessary overhead and may reduce parallelism.
• A critical reference is a reference to a variable which is (can be) changed by another process.
• When a statement appears to be executed atomically? – When it satisfies the At-Most-Once (AMO) property, i.e., when either of the following is true:
  a) It reads at most one critical reference and does not write any critical references,
  b) It writes at most one critical reference and does not read any critical references.

Implication of the AMO Property

< S; > ≡ S;
– as long as S contains at most one critical reference seeing by other processes,
– In this case, one cannot tell the difference, so S will appear to execute atomically w.r.t. to other processes.

< await(B); > ≡ while (not B) continue;
– if B contains (reads) at most one critical reference
Parallelizing a Sequential Program

• How? Identify independent and dependent tasks, assign tasks to processes, identify shared variables, synchronize processes.

• Data accessed by a process:
  – Read set (rs) – variables that are only read
  – Write set (ws) – variables that are written (and possibly read)

• Independence: processes are independent iff the write set of one proc is disjoint from the read and write sets of another proc:

\[
ws_1 \cap (rs_2 \cup ws_2) = \emptyset \text{ AND } ws_2 \cap (rs_1 \cup ws_1) = \emptyset
\]

Example of Parallelization and Synchronization: Finding Patterns in a File

• Find all instances of a pattern in a file: `grep pattern filename`
  – Sequential program:
    ```
    string line;
    open the file; read a line of input from the file into line;
    while (!EOF) {  # EOF is end of file
      look for pattern in line;
      if (pattern is in line) print line;
      read next line from the file into line;
    }
    ```

• Parallelization:
  – Tasks: read a line, compare with the pattern, print if match
  – Apply Producer-Consumer paradigm:
    • P reads a line from the file and stores it to a shared buffer
    • C gets a line from the buffer, tests it against pattern, prints if match.
Example (cont’d): How to Synchronize

- True/anti dependence between P and C via the shared buffer.
- Let the buffer provides place for only one string
- Synchronization:
  - Mutual exclusion: The buffer is accessed by one process at a time
  - Condition synchronization
    - To put a line, Producer waits for the buffer to be empty
    - To get a line, Consumer waits for a buffer to become full
- Let’s use two counters:
  - \( p \) – number of produced lines (stored to the buffer by Producer)
  - \( c \) – number of consumed (fetched from the buffer by Consumer)
- The synchronization requirement (predicate):
  \[ PC: c \leq p \leq c + 1 \]

Example (cont’d): Parallel Program

```c
# shared variables:
string buf;  /* buffer (input line) */
bool done = false;  /* termination */
int p = 0, c = 0;  /* counters */

Process Producer {  /* reads new lines */
    string line;
    open the file;
    while (true) {
        read a line of input from the file into line;
        if (EOF) { done = true; break; }
        /* wait for buffer to be empty */
        < await (p == c);>
        buf = line;
        p++;  /* signal that buffer is full */
    }
}

Process Consumer {  /* finds patterns */
    string line;
    while (true) {
        /* wait for buffer to be full */
        /* or done to be true */
        < await ((p > c) || done); >
        if (done) break;
        line = buf;
        c++;  /* signal that buffer is empty */
        if (pattern in line) write line;
    }
}
```

Properties of a Program

- A property of a program is an attribute of ALL histories of a program, e.g., correctness.
- Two kinds of properties:
  - Safety properties
    - A safety property is one in which the program never enters a “bad” state
  - Liveness properties
    - A liveness property is one in which the program eventually enters a “good” (desirable) state

Safety and Liveness Properties

- Safety properties:
  - Partial correctness
    - The final state is correct, assuming that the program terminates
  - Mutual exclusion of critical sections
    - “Bad” when more than one proc. are in critical sessions
  - Absence of deadlocks
    - Deadlock: Wait for a condition that never occurs
- Liveness properties:
  - Termination
    - Every history (trace) is finite.
  - Eventually enter a critical section (if any)
- Total correctness combines termination and partial correctness
  - A program always terminates with a correct answer
Proving Properties

Three main approaches
1. Testing or debugging
   – Run and see what happens
   – Limited to considered cases
2. Operational reasoning
   – “Exhaustive case analysis”
   – Considers enormous number of histories: $n \cdot m!/(m!)^n$
   – Helps in development
3. Assertional reasoning
   – Based on axiomatic semantics: axioms, inference rules
   – Work is proportional to the number of atomic actions

Introduction to Axiomatic Semantics of Concurrent Programs with Shared Variables
Formal Logical System

- A set of symbols
- A set of formulas constructed from symbols
- A set of axioms, i.e. formulas which are (assumed to be) true
- A set of inference rules used to derive new true formulas (conclusions $C$) from other true formulas (hypotheses $H$)

$$H_1, H_2, \ldots, H_n \quad \vdash \quad C$$

- Proof is a sequence of lines each of which either axiom or can be derived from previous by applying one of inference rules
- Theorem is a line in a proof

Interpretation of a Logic. Soundness and Completeness

- **Interpretation** maps each formula to true or false
  - Formulas: statements about some domain of discourse
  - Interpretation is a model for the logic if the logic is sound w.r.t. interpretation
- Logic is sound w.r.t. interpretation if all axioms and inference rules are sound
  - An axiom is sound if it maps to true
  - An inference rule is sound if $C$ maps to true assuming $H$s
- A logic is complete w.r.t. interpretation if every formula that maps to true is provable in the logic
Programming Logic (PL)

- What? Formal logical system that allows to state and to prove properties of programs.
- Why?
  - Allows proving correctness of concurrent programs.
  - Provides a systematic way to understand and to develop correct concurrent programs.

PL (cont’d)

- Symbols are predicates, braces and prog. statements
- Formulas are *triples* of the form
  \[
  \{ \, P \, \} \quad S \quad \{ \, Q \, \}
  \]
  - \(P, Q\) are predicates (assertions)
  - \(S\) is a statement list
- Predicates tells about state
  \[
  \{ \, x == 0 \; \text{and} \; y == 0 \, \}
  \]
- Interpretation defines relation between \(P, Q\) and \(S\)
  - \(P\) is called the *precondition* (state before execution)
  - \(Q\) is called the *postcondition* (state after execution)
  - Extreme assertions: true (all states), false (no state)
Interpretation of a Triple

- The triple \( \{ \mathbf{P} \} \ \mathbf{S} \ \{ \mathbf{Q} \} \) is true if, whenever execution of \( \mathbf{S} \) is begun in a state satisfying \( \mathbf{P} \) and execution of \( \mathbf{S} \) terminates, the resulting state satisfies \( \mathbf{Q} \).
  - Partial correctness

- Examples:
  \[
  \begin{align*}
  \{ x == 0 \} \quad &x = x + 1; \quad \{ x == 1 \} \\
  &\quad \text{Should be a theorem}
  \\
  \{ x == 0 \} \quad &x = x + 1; \quad \{ y == 1 \} \\
  &\quad \text{Should not be a theorem (does not sound!)}
  \end{align*}
  \]

Assignment Axiom

\[
\{ \mathbf{P} \ x \leftarrow e \} \quad x = e \quad \{ \mathbf{P} \}
\]

- \( \mathbf{P} \ x \leftarrow e \) specifies textual substitution: replace all free occurrences of the variable \( x \) in \( \mathbf{P} \) by expression \( e \).
  - If you want a final state to satisfy \( \mathbf{P} \), then the prior state must satisfy \( \mathbf{P} \) with \( x \) textually replaced by \( e \)

- For example:
  \[
  \begin{align*}
  \{ 1 == 1 \} \quad &x = 1 \quad \{ x == 1 \} \\
  \{ x == 0 \} \quad &x = 1 \quad \{ x == 1 \}
  \end{align*}
  \]
Inference Rules in PL

- Used to characterize the effects of prog. statements

Composition rule:

\[
\begin{align*}
(P) & \text{ S1 } (Q), \quad (Q) \text{ S2 } (R) \\
\implies & \quad (P) \text{ S1; S2 } (R)
\end{align*}
\]

If Statement rule:

\[
\begin{align*}
(P \land B) & \text{ S } (Q), \quad (P \land \neg B) \implies Q \\
\implies & \quad (P) \text{ if } (B) \text{ S; } (Q)
\end{align*}
\]

While Statement rule:

\[
\begin{align*}
(I \land B) & \text{ S } (I) \\
\implies & \quad (I) \text{ while}(B) \text{ S; } (I \land \neg B)
\end{align*}
\]

Rule of Consequence:

\[
\begin{align*}
P' & \Rightarrow P, \quad (P) \text{ S } (Q), \quad Q \Rightarrow Q' \\
\implies & \quad (P') \text{ S } (Q')
\end{align*}
\]

- The rule of consequence allows to strengthen preconditions and to weaken postconditions.

Semantics of Synchronization and Concurrent Execution

Await statement rule:

\[
\begin{align*}
(P \land B) & \text{ S } (Q) \\
\implies & \quad (P) \{ \text{ await } (B) \text{ S; } \} (Q)
\end{align*}
\]

Co statement rule:

\[
\begin{align*}
(P_i) & \text{ S}_i (Q_i) \text{ are inference free} \\
(P_1 \land \ldots \land P_n) & \text{ S}_1; \quad || \ldots || \text{ S}_n; \quad \text{ooc } (Q_1 \land \ldots \land Q_n)
\end{align*}
\]

- For conclusion to be true, proofs of hypotheses must not interfere each other.
- A proc interferes with another proc if the former executes an assignment that invalidates an assertion of the latter.
  - Arises because of shared variables.
Noninterference

• Define:
  – An assignment action is an assignment statement or an await statement that contains assignments.
  – A critical assertion is a precondition or postcondition that is not within an await statement.

• Noninterference:
  – Let \( a \) be an assignment action in one process and let \( \text{pre}(a) \) be its precondition.
  – Let \( C \) be a critical assertion in another process.
  – Then \( a \) does not interfere with \( C \) if the following is a theorem in programming logic
    \[ \{ C \land \text{pre}(a) \} ; \{ C \} \]

Ways to Avoid Interference

1. Disjoint variables
   – Avoid false dependences.
2. Weakened assertions
   – Say less than you could in isolation, take into account concurrency.
3. Global invariants
   – Predicates that are true in all visible states.
4. Synchronization
   – Hide states and/or delay execution.
1. Disjoint Variables

- Recall:
  - $P_1$ and $P_2$ do not depend on each other if $ws_1 \cap (rs_2 \cup ws_2) = \emptyset$ AND $ws_2 \cap (rs_1 \cup ws_1) = \emptyset$
  - Reference set \text{(refs)} is formed of variables that appear in assertions in a proof
  - Assertions should not capture false dependences!
- This implies:
  $P_1$ and $P_2$ do not interfere with each if $ws_1 \cap \text{refs}_2 = \emptyset$ AND $ws_2 \cap \text{refs}_1 = \emptyset$

2. Weakened Assertions

- Take into account concurrency and say less than you could in isolation
  - A weakened assertion admits more program states than another assertion of a process in isolation
- For example:
  \begin{verbatim}
  { x == 0 } co ( x == 0 v x == 2 ) < x = x + 1 ; > { x == 1 v x == 3 } \\
  |  | { x == 0 v x == 1 } < x = x + 2 ; > { x == 2 v x == 3 } \\
  oc ( x==3 )
  \end{verbatim}
  - The pre(post)condition of the program is the conjunction of the pre(post)conditions of the processes.
3. Global Invariants

- Suppose \( I \) is a predicate that references global (shared) variables.
- The predicate \( I \) is a global invariant for a set of processes if:
  - \( I \) is true when the processes begin execution,
  - \( I \) is preserved by every assignment action.
- If every critical assertion \( C \) in the proof of every process \( P_i \) has the form \( I \land \bigwedge L \),
  - where \( L \) is a predicate on local vars in \( P_i \), i.e. every var in \( L \) is assigned by only \( P_i \),
  then the proof of the processes \( P_i \) is interference-free.

4. Synchronization

- Used to delay processes by strengthening preconditions with additional constraints to avoid interference.
  - A process waits until a stronger precondition is true, i.e. until its precondition is true and there is no interference with other processes.
- Used to execute a set of statements as an atomic action.
  - To hide internal states.
  - To access shared variables with mutual exclusion.
Example: Producer-Consumer

- Copy \( a[n] \) in **Producer** to \( b[n] \) in **Consumer** using a shared single-slot buffer \( buf \)
  - **Producer** writes \( a[p], p = 0, 1, 2, \ldots, n \) to the \( buf \)
  - **Consumer** reads \( buf \) to \( b[c], c = 0, 1, 2, \ldots, n \)
  - Condition synchronization to alternate access to the buffer
  - The synchronization requirement (predicate):
    \[ PC: c \leq p \leq c + 1 \]
    where \( p \) – number of produced (stored)
    \( c \) – number of consumed (fetched)

Copying an Array From a To b

```c
int buf, p = 0, c = 0;
process Producer {
    int a[n];
    while (p < n) {
        < await (p == c);>
        buf = a[p];
        p = p+1;
    }
}
process Consumer {
    int b[n];
    while (c < n) {
        < await (p > c);>
        b[c] = buf;
        c = c+1;
    }
}
```
Global Invariant of the Application

The synchronization requirement (predicate):

\[ c \leq p \leq c + 1 \]

Global invariant:

\[ \text{PC}: (c \leq p \leq c + 1) \land (a[0:n-1] = A[0:n-1]) \land (p = c+1) \Rightarrow (\text{buf} = A[p-1]) \]

where \( A[n] \) are values stored in \( a[n] \)

1) \( \text{buf} \) is either full \( (p = c+1) \) or empty \( (p = c) \)
2) \( a \) is not altered
3) When \( \text{buf} \) is full \( (p = c+1) \), it contains a value \( (A[p-1]) \)

Proof Outline

\begin{verbatim}
int buf, p = 0, c = 0;
{PC}: c <= p <= c + 1 \land a[0:n-1] = A[0:n-1] \land (p = c+1) \Rightarrow (buf = A[p-1])

process Producers()
    int a[0]; # assume a[1] is initialized to A[1] {PC}: PC \land p <= n
    while (p <= n) [
        {PC}: p < n \land \text{buf} = A[p];
        {PC}: p < n \land p <= c \land buf = A[p]
        p = p+1;
    ]

process Consumers()
    int b[n];
    {IC}: IC \land c <= n \land b[0:c-1] = A[0:c-1]
    while (c <= n) [
        {IC}: c <= n \land b[c] = A[c];
        {IC}: c <= n \land p > c \land buf == A[p]
        c = c+1;
    ]

{IP}

\end{verbatim}

- This proof outline captures the essence of what is true at each point
- It’s an encoding of an actual proof in PL
Proving Safety Properties

• A safety property: nothing bad ever happens – no bad states

1. Avoid “bad” states
   – Assume $\text{BAD}$ is a predicate that defines a “bad” program state according to some property $P$
     • The program deadlocks
     • More than one process enter critical section
   – The program satisfies $P$ if $\text{BAD}$ is false in every history.

2. Be always in “good” states
   – Assume $\text{GOOD} = \neg \text{BAD}$ is a predicate that defines a "good“ program state according to some property $P$
   – The program satisfies $P$ if $\text{GOOD}$ is its global invariant.

Synchronization.

• Critical sections
• Locks, flags and barriers
• Condition variables
• Semaphores
Accessing Shared Data

• Assume:
  – Shared counter \( x \) is initially 0
  – Two concurrent threads increment \( x \)
  – Expected result is \( x == 2 \)

• Process histories:
  P1: …; load \( x \) to reg; incr reg; write reg to \( x \); …
  P2: …; load \( x \) to reg; incr reg; write reg to \( x \); …
  – Without synchronization, the final result can be \( x == \{1, 2\} \)

• The statements accessing shared variables are critical sections
  that must be executed one at a time, i.e. atomically (with
  mutual exclusion)

The Critical Section Problem

• Critical section is a section of code that access a shared resource and can
  only be executed by one process at a time

• The Critical Section Problem: To find a mechanism that guarantees
  execution of critical sections one at a time.
  – The problem arises in most concurrent programs. For example: shared
    linked lists in OS, database records, shared counters, etc.

• Model for the CS problem:
  ```
  process \( P[i = 0 \text{ to } n-1] \) {
      while (true) {
          CSentry: entry protocol;
          critical section;
          CSexit: exit protocol;
          non-critical section;
      }
  }
  ```
Locks

- Locks provide a solution for the CS problem
  - Lock on entry, unlock on exit
- Types of locking mechanisms:
  - Spin locks – efficient but unfair
    - Short latency and low memory demand
    - Poor fairness, may cause starvation
    - Good in case of low contention (a few processes)
    - Examples: Test&set lock, test-test&set lock, test&set lock with backoff
  - Queuing locks – fair solutions but more expensive
    - Longer latency, more memory – the price for fairness
    - Examples: tie-breaker lock, ticket lock, bakery lock

Critical Sections Using Locks

- A spin lock is a Boolean variable that indicates whether one of the processes is in its critical section:
  - lock == 1 – the lock is locked, i.e. some process is in its CS
  - lock == 0 – the lock is unlocked, i.e. no process in CS
- Solution to the CS problem using spin locks:
  ```
  bool lock = false;
  process P[i = 0 to n-1] {
    while (true) {
      Lock(lock); /* CSentry */
      critical section;
      Unlock(lock); /* CSexit */
      non-critical section;
    }
  }
  
  procedure Lock(var bool location) {
    <wait (!location) location = true;>
  }
  
  procedure Unlock(var bool location) {
    location = false;
  }
  ```
Implementation of Spin Locks

- Spin lock requires atomicity is its own implementation:
  `<await (!location) location = true;>
- Unlock is implemented with ordinary store operation
  `location = false;
- HW support for synchronization – a special atomic memory
  instruction <read-modify-write>, such as Test&Set, swap, 
  compare&swap, fetch&increment
- Simple spin lock using Test&Set instruction (t&s):
  
  **Lock:**
  
  `t&register, location // try to lock the location
  bnz lock // if not 0, try again
  ret // return to caller
  
  **Unlock:**
  
  `st location, #0 // write 0 to location
  ret // return to caller

Implementing Atomic Actions as Critical Sections

- Observation: Mutual exclusion guarantees atomicity
- Any solution to the CS problem (CSenter and CSexit) can be
  used to implement atomic actions

<table>
<thead>
<tr>
<th>Atomic action</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt; S;&gt;</code></td>
<td>CSenter; <code>S; CSexit;</code></td>
</tr>
<tr>
<td><code>&lt;await(B);&gt;</code></td>
<td>CSenter; <code>while (!B) { CSexit; Delay; CSenter; } CSexit;</code></td>
</tr>
</tbody>
</table>
| `<await(B) S;>`| CSenter; `while (!B) { CSexit; Delay; CSenter; }
  `S; CSexit;` |

- To avoid a deadlock, a proc awaiting `B`, must repeatedly exit and enter
  its CS to allow other processes to enter CS and alter variables in `B`
Flag Synchronization

- **Flag synchronization** delays a process until a binary flag is set (or cleared) by another proc
  - A point-to-point signaling mechanism used as condition synchronization (to wait for a condition, to signal the condition).
- To reuse a synchronization flag:
  - The process that waits for a flag to be set is the one that should clear that flag.
  - The flag should not be set until it is known that it is cleared.
- Typical usage:
  - Proc 1 sets a flag ("sends a signal"):
    ```
    while (flag) continue; flag = 1;
    ```
  - Proc 2 clears the flag ("receives a signal"):
    ```
    while (!flag) continue; flag = 0;
    ```

Barrier Synchronization

- In iterative computation, often an inner loop (e.g. over data) can be parallelized, whereas an outer loop (e.g. over computation stages) cannot.
  - To synchronize stages – barrier at the end of each stage
- A barrier is a point that all processes must reach before any proceed
  ```
  bool done = false;
  process P[i = 0 to n - 1] {
    while (!done) {
      code to implement task i;
      Barrier(n); /* Wait for all tasks to complete */
    }
  }
  ```
  - Barriers can be implemented using:
    - Locks and flags – busy-waiting
    - Locks and condition variables – blocking
    - Semaphores - blocking
Waiting for Synchronization

- Busy waiting
  - A proc spins on a variable, e.g. lock, flag
  - Does not need OS to be involved
- Shortcomings of Busy Waiting
  - Inefficiency: waste CPU time
  - Complexity
    - Difficult to specify conditions for waiting
    - Difficult to reuse synch variables
- Blocking
  - A proc is blocked to be resumed on synchronization
  - Controlled by OS (or runtime system)
- Blocking synchronization mechanisms:
  - Condition variables
  - Semaphores

Condition Variables

- **Condition variable** is an opaque object that represents a queue of suspended processes waiting to be resumed (when the variable is signaled)
  - A mechanism to wait and signal conditions
  - Allows to suspend and to resume processes holding locks
- Operations:
  - Blocking wait
  - Signal to resume
- Locks and condition variables – synchronization mechanisms in Pthreads
  - Locks for mutual exclusion
  - Condition variables – for condition synchronization
- Implicit condition variables are used in Java monitors
Operations on Condition Variables

- Declaration: `cond cv;
- Operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wait(cv, lock)</code></td>
<td>Release the lock and wait at end queue</td>
</tr>
<tr>
<td><code>wait(cv, rank, lock)</code></td>
<td>Release lock and wait in order of increasing value of rank</td>
</tr>
<tr>
<td><code>signal(cv)</code></td>
<td>Awaken process at front of queue then continue</td>
</tr>
<tr>
<td><code>signal_all(cv)</code></td>
<td>Awaken all processes on queue then continue</td>
</tr>
<tr>
<td><code>empty(cv)</code></td>
<td>True if queue is empty; false otherwise</td>
</tr>
<tr>
<td><code>minkrank(cv)</code></td>
<td>Value of rank of proc at front of wait queue</td>
</tr>
</tbody>
</table>

- When used in monitors, `lock` is not specified, the implicit monitor lock is assumed

Signaling Disciplines

- **Signal and Continue (SC)** – the signaling process continues, the resumed process reacquires the lock
- **Signal and Wait (SW)** – the signaling process passes the lock to the resumed process and reacquires the lock
- **Signal and Urgent Wait (SUW)** – as SW but the signaling process is placed to the head of the lock queue
Semaphores

- Semaphores provide a low-level but unified and efficient signaling mechanism for both mutual exclusion and condition synchronization.
- Origin of semaphores:
  - Invented in 1968 by Edsger Dijkstra, Dutch computer scientist.
  - Inspired by a railroad semaphore – Up/Down signal flag.
  - THE system – the first OS that used semaphores.
- Semaphore operations in Dutch:
  - P (decrement when nonnegative) stands for “proberen” (test) or “passeren”.
  - V (increment) stands for “verhogen” or “vrijgeven”.

Syntax and Semantics

- A semaphore is a special kind of shared integer variable that is manipulated by two atomic operations, P and V.

\[
\begin{align*}
  & P(s) :< \text{await} \ (s > 0) \ s = s - 1; > \ # \ \text{wait, down} \\
  & V(s) :< s = s + 1; > \ # \ \text{signal, up}
\end{align*}
\]

- The value of a semaphore is nonnegative integer.
- Declaration and initialization:

```plaintext
sem s = expr; \ # \ \text{single semaphore}
sem s[1:n] = ([n] expr) \ # \ \text{array of semaphores}
```
- Defaults to zero.
Implementation

- Blocking instead of busy waiting
- The P(s) operation:
  
  ```c
  if (s > 0) s--; 
  else { /* wait until the semaphore is up */ 
       Block the proc: add it to the semaphore queue; 
  }
  ```
  
  – The suspended process is resumed by V(s) when it reaches the head of the queue;
- The V(s) operation:
  
  ```c
  if (queue is empty) s++; 
  else { /* "pass the baton " */ 
       Resume the process in the head of the semaphore queue: move the process from the semaphore queue to the ready queue; 
  }
  ```

Types of Semaphores

- **Binary semaphore** takes the value 0 or 1
  
  – Can be used for mutual exclusion (mutex) and/or condition synchronization (signaling semaphores – much like flags)

- **Split Binary semaphore** is a set of binary semaphores where at most one semaphore is 1 at a time:
  
  \[ 0 \leq s_0 + s_1 + \ldots + s_{n-1} \leq 1 \]

  – Can be used for both mutex and condition synchronization

- **General (counting) semaphore** takes any nonnegative integer value

  – Can be used for condition synchronization.

  – Serves as a resource counter: counts the number of recourse units
Use of Binary Semaphores for Synchronization

- Mutual exclusion <\textit{S};>
  
  \begin{verbatim}
  sem entry = 1; /* CS entry semaphore */
  P(mutex); S; V(mutex);
  \end{verbatim}

- Condition synchronization <\textit{await} (B) \textit{S};>
  
  \begin{verbatim}
  sem entry = 1; /* CS entry semaphore */
  sem cond = 0; /* to signal condition */
  P(entry);
  while (!B) { V(entry); P(cond); P(entry); } 
  S;
  V(entry);
  \end{verbatim}

  “Passing the baton” technique can be used to optimize:
  - When signal \textit{cond.entry} can be “passed” with \textit{cond}

General Semaphores

- Can be associated with a shared recourse and serve as a resource counter:
  - Initialized to some integer value – the total amount of recourse units available;
  - Takes any nonnegative integer value – current amount of resource units available
  - Used for condition synchronization, e.g. wait until a unit of resource is available and can be occupied
Example: Bounded Buffer Problem

- Producers and Consumers interact using a bounded buffer – a multi-slot communication buffer limited in size (no underflow, no overflow)
- Buffer of size $n$ as an array of some type $T$:
  
  ```c
  TypeT buf[n];
  int front = 0, rear = 0;
  ```

- Accessing buffer
  - `rear` points to an empty slot
  - `front` points to the head data item
  ```
  deposit:  buf[rear] = data;
          rear = (rear + 1) % n;
  ```

  ```
  fetch:  result = buf[front];
          front = (front + 1) % n;
  ```

Synchronized Access to the Bounded Buffer

- $P$ must be delayed until there is an empty slot in the buffer
- $C$ must be delayed until there is a full slot in the buffer.
- Use two general semaphores:
  ```
  sem empty = n, full = 0; /* n-2 ≤ empty+full ≤ n */
  ```

  - `empty` counts empty slots,
  - `full` counts deposited data items.
Bounded Buffer Using General Semaphores

```c
typedef struct { } T
int front = 0, rear = 0;
sem empty = n, full = 0; /* n-2 <= empty-full <= n */
process Producer {
    while (true) {
        produce message data and deposit it in the buffer;
        P(empty);
        buf[rear] = data; rear = (rear+1) % n;
        V(full);
    }
}
process Consumer {
    while (true) {
        fetch message result and consume it;
        P(full);
        result = buf[front]; front = (front+1) % n;
        V(empty);
        ...
    }
}
```

Pros and Cons of Semaphores

- Sufficiently general to solve any mutual exclusion or condition synchronization problem
- Disadvantages:
  - Complexity: Can be difficult to understand as the same operations are used for both mutual exclusion and condition synchronization.
  - Error prone: It is easy to make mistakes, such as forgetting a V operation, especially when several semaphores are used. Such error typically causes a deadlock.
  - There is a loss of modularity since the correct use of a semaphore in one process often depends on its use in another process.
  - Semaphores are too low-level.
- A combination of locks and condition variables have the same synchronization power like semaphores
Distributed Programming

• A distributed program is formed of processes distributed across nodes of a distributed memory platform
  – No shared variables, only communication channels are shared
  – Processes have to exchange messages in order to interact
• Message passing provide ability for moving data across process private memory spaces
  – Message passing primitives: send and receive
  – Distributed programs use message passing for inter-process communication
• Programming options
  – Design a special programming language or extend an existing one
  – Provide an existing (sequential) language with external library for distributed programming
Existing Mechanisms and Environments

- Communication mechanisms:
  - Asynchronous message passing
  - Synchronous message passing
  - Remote procedure call (RPC) and rendezvous
  - Remote method invocation (RMI)

- Programming environments and APIs:
  - Berkley Sockets API – substrate technology
  - PVM (Parallel Virtual Machine)
  - MPI (Message Passing Interface)
  - RPC API, Java socket API, CORBA, Java RMI

Programming with Distributed Processes

- Processes of a distributed applications are heavy-weight processes started asynchronously on different (or the same) computers
  - One process of a distributed application can start another (detached) HW-process by `fork` and `exec` or `rexec`

- **Single Program Multiple Data (SPMD) model**
  - One program is written for several processes (tasks)
  - Control (conditional) statements are used to select different part (task) for each process
  - To select a task, process needs to know: the total number of processes in the application (or in a group), its own number (rank).

- **Multiple Program Multiple Data (MPMD) model**
  - Different programs are written for different processes (tasks)
Message Passing

- **Message passing** is sending and receiving messages via communication channels shared by processes
  - Send/receive is one-way data transfer from a memory of a source area to a destination area
- Message passing involves synchronization - a message cannot be received until it has been sent.
- A channel can be implemented as:
  - a shared memory location - on a single processor or a shared memory multiprocessor
  - a communication link (communication HW assist) - on a distributed memory multiprocessor or network of workstations

Massage Passing Models

- **Synchronous** – blocking semantics of both, send and receive
  - Send completes after matching receive is posted and data are sent
  - Receive completes after data has been received from the matching send
  - Channel can be implemented without buffering
- **Asynchronous** – non-blocking send and blocking receive – send is asynchronous w.r.t. receive
  - Send completes after send buffer may be reused
  - Channel is an unbounded FIFO queue of messages
Pros and Cons of Asynchronous Message Passing

- More convenient for programming
  - More concurrency, less deadlock situations
- But: it assumes an unbounded amount of storage outside usual process memory
  - More complicated to implement
  - The amount of memory space for communication channels is unpredictable (and so must be allocated dynamically)

Pros and Cons of Synchronous Message Passing

- Channels can be implemented without buffering
  - Data are transferred from the sender’s memory space to the receiver’s memory space without intermediate buffering and extra copying
- But:
  - Limits concurrency
  - Deadlock prone
Paradigms for Process Interaction in Distributed Programs

- Basic interaction patterns in distributed programs:
  1. Producer/consumer (filters) – one way
  2. Client/Server – two ways as master/slave
  3. Interacting peers – two ways as equals

- Larger paradigms (model) for process interactions
  1. Manager/Workers (a.k.a. “bag of tasks” or “work farm”)
  2. Heartbeat algorithms
  3. Pipeline algorithms
  4. Probe/echo algorithms
  5. Broadcast algorithms
  6. Token-passing algorithms
     - The first three paradigms are commonly used in parallel computations; the others arise in distributed systems

1. Manager/Workers (Bag of Tasks)

- Represents distributed bag of tasks (or “work farm”) model
  - Manager maintains a bag of independent tasks and collects results
  - Each worker removes a task from the bag, executes it, and possibly generates new tasks that it puts in the bag.
  - Manager acts as a server; workers – as clients.
  - How can the manager detect termination? every worker is waiting to get a new task and the bag is empty.

- Advantages:
  - Scalability: Easy to vary the number of workers, and granularity of tasks.
  - Load balancing: Easy to ensure that each worker does about the same amount of work.
2. Heartbeat Algorithms

- Divide work (evenly).
- Processes periodically exchange information using a send (expand) and then receive (contract) interaction.
  - Typically each proc exchanges data with its neighbors
  - The exchanges provide a “fuzzy” barrier among the workers
- Used in many iterative applications with data parallelism
  - E.g. exchange edges in Jacobi iteration

A Typical Heartbeat Algorithm

```c
process Worker[i = 1 to numWorkers] {
    declarations of local variables;
    initialize local variables;
    while (not done) {
        send values to neighbors;
        receive values from neighbors;
        update local values;
    }
}
```

- Examples: Jacobi iteration, region labeling
3. Pipeline Algorithms

• What: divide work evenly, compute and circulate data among workers
• When: used when workers need all the data, not just edges from neighbors.
• Pipeline structures – circular or open (or closed)

\[ W_1 \rightarrow \cdots \rightarrow W_n \]

(a) open

\[ W_1 \rightarrow \cdots \rightarrow W_n \rightarrow \text{Coordinator} \]

(b) closed

\[ W_1 \rightarrow \cdots \rightarrow W_n \]

(c) circular

4. Probe/Echo Algorithms

• Used to disseminate and/or to gather information
  – Probes – to disseminate request or information
  – Echoes – to collect information or to acknowledge
  – Each probe should be echoed
• For example:
  – Broadcast a message
    • Using a spanning tree – when knows a global topology
    • Using neighbor sets – when knows neighbors
  – Network topology problem: collect all local topologies and build their union, i.e. the entire network topology
  – Web "crawlers"
5. Broadcast Algorithms

• Used
  – For dissemination of information
  – For making decentralized decisions
    • Each process must participate in every decision
  – For solving many distributed synchronization problems (e.g. distributed semaphores, distributed mutual exclusion)

• Use logical clocks to order communication events:
  – A logical clock (\(lc\)) is a private integer counter that a proc increases after every communication event
  – Proc attaches a timestamp (\(ts\)) to each message it sends:
    \[ ts = lc++ \; ; \; \text{send}(m, \; ts) \]
  – Proc checks and corrects its \(lc\) when it receives a message with \(ts\):
    \[ \text{Receive}(m, \; ts) \; ; \; lc = \max(lc, \; ts+1) \; ; \; lc++ \]

6. Token Passing Algorithms

• Used to convey permission (e.g. distributed mutual exclusion) or to detect termination

• Example: Mutual exclusion with a token ring
  – A lock is a token that is passed in the ring
  – A proc that needs the locks grabs it when the lock passes the proc
Parallelism in Scientific Computing

Scientific Computing

- Scientific computing is a way to examine physical phenomena by *computational modeling*
  - Weather forecast, modeling of ocean basin, CFD (Computational Fluid Dynamics), modeling of nuclear reactions, modeling the evolution of galaxies, etc.
- High Performance Computing (HPC): improve performance by any means
  - Traditional demand of scientific computing
Parallel Scientific Computing

- Goal: speedup on LARGE problems (or solve an even larger problem)
- Speedup: $T_1 / T_n$ for $n$ processors
  - start with a good algorithm and optimized sequential code
- Parallelization:
  - Simple way to identify concurrency is to look at loop iterations
    - dependence analysis; if not enough concurrency, then look further
  - Not much concurrency here at this level (all loops sequential)
  - Examine fundamental dependences, ignoring loop structure
- The challenge for a parallel program is to minimize overheads:
  - Create processes once
  - Have good load balancing
  - Minimize the need for synchronization and use efficient algorithms for critical sections and barriers

Parallel Simulation of Phenomena

- Parallelization – the idea of domain decomposition: divide data into partitions; assign a worker to each (or use bag of tasks)
- Typical parallel simulation algorithm:
  - (with shared memory)
    start with a model;
    // step through time
    for [ t = start to finish ] {
      Compute;
      BARRIER;
      Update;
      BARRIER;
    }
  - (with message passing)
    start with a model;
    // step through time
    for [ t = start to finish ] {
      Compute;
      EXCHANGE;
      Update;
      EXCHANGE;
    }
Techniques in Scientific Computing

• **Grid computations**
  – To solve PDEs – to approximate the solution at a finite number of points using iterative numerical methods
  – PDEs are used to model continues systems and processes (e.g. airflow over a wing)
• **Particle computations**
  – To model (discrete) systems of particles/bodies that exert influence on each other (e.g. stars)
• **Matrix computations**
  – Linear equations
  – Arise in many application domains like optimization problems, e.g., modeling the stock market or the economy, image processing, etc.

1. Grid Computations

• A continues-time continues-space process is model as a 3D (2D) grid of points in discrete time
  – The finer spatial and temporal steps are the greater accuracy can be achieved
  – Many different computations per time step
  – Applications: weather, fluid (air) flow, plasma physics, etc.
• PDE solvers
• Parallelization – the idea of domain decomposition (iterative data parallelism)
  – divide area into blocks or strips of points; assign a worker to each
2. Particle Computations

- To model a discrete system consisting of interacting particles/bodies
  - $n$ particles requires $O(n^2)$ calculations on a time step
  - By using approximations, can be improved to $O(n \log_2 n)$
- Typically irregular iterative computations
- Examples of applications:
  - Particle interactions due to chemical bonding,
  - Gravity (evolution of galaxies)
  - Electrical charge, fluid flow, etc.

Example: Gravitational N-Body Problem

- N bodies. Each body:
  - Position $p(x, y, z)$
  - Velocity $v(x, y, z)$
  - Mass $m$ – constant
  - Force $F(x, y, z)$
- Gravity causes the bodies to accelerate and to move.
- The motion of the bodies is simulated by stepping through discrete instants of time:
  initialize bodies;
  for time step {
    calculate forces; // read $p, m$; compute $F$
    move bodies;    // read $F, m$; compute new $p$ and $v$
  }
Calculation Forces. Moving Bodies

- Force on a body is the vector sum of the forces from all other bodies
  - Magnitude of the gravitational force between bodies i and j:
    \[ F_{i,j} = \frac{G \times m_i \times m_j}{r_{i,j}^2} \]
  - magnitude: symmetric ("equal and opposite")
  - direction: vector from one body to the other
- Changes in velocity and position (moving a body) – leapfrog scheme:
  - Acceleration: \( a = \frac{F}{m} \)
  - Change in velocity: \( dv = a \times DT \)
  - Change in position:
    \[ dp = v \times DT + (a/2) \times DT^2 = (v + dv/2) \times DT \]
    - Here \( DT \) – the length of a time step

Parallelization of the N-Body Problem

- Assume, shared memory programming model
- Parallelization – divide bodies among workers use a barrier after each phase:
  initialize bodies;
  for time step {
    calculate forces;
    BARRIER;
    move bodies;
    BARRIER;
  }
Assignment of Bodies to Workers

- Assume, 2 workers (B – “Black” and W – “White”) and 8 bodies
- For each body $i$ assigned to a worker, the worker computes forces between the body $i$ and bodies $i + 1, \ldots, n$

<table>
<thead>
<tr>
<th>Pattern</th>
<th>1 2 3 4 5 6 7 8</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks</td>
<td>B B B B W W W W</td>
<td>$B = 22, W = 6$</td>
</tr>
<tr>
<td>stripes</td>
<td>B W B W B W B W</td>
<td>$B = 16, W = 12$</td>
</tr>
<tr>
<td>reverse stripes</td>
<td>B W W B B W W B</td>
<td>$B = 14, W = 14$</td>
</tr>
</tbody>
</table>

- The pattern of stripes leads to a fairly well-balanced workload that is easy to program.

Synchronization

- Barriers after each stage
- Critical sections in calculate forces phase:
  - Access to the same body by different workers must be synchronized (mutual exclusion)
  - Approaches:
    - One global lock – very inefficient
    - A lock per a body – too much synchronization
    - Eliminate critical sections at all
- To eliminate critical sections, replicate shared data:
  - Change the force vector into a force matrix: a private force vector per worker.
  - The result force vector: sums in columns – can be calculated in parallel in the moving phase
Introduction to Pthreads

Pthreads: POSIX Threads

- **Pthreads** is a standard set of C library routines for multithreaded programming with shared variables
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Allows to create and synchronize multiple threads in a heavy-weight process
  - Threads share a common address space (thru common variables)
  - Each thread has a private stack for local variables
- Goal in developing the Pthreads API:
  - “To give programmers the ability to write concurrent applications that run on both uniprocessor and multiprocessor machines transparently, taking advantage of the additional processors if any.”
Pthreads APIs

- Thread Management (29 functions)
  - Create, init, detach, delay, destroy, exit, join, get/set attributes, …
- Thread Cancellation (9)
  - Cancel, test for cancellation, ...
- Mutex Synchronization (19)
  - Init, destroy, lock, unlock, try lock, get/set attributes
- Condition Variable Synchronization (11)
  - Init, destroy, wait, timed wait, signal, broadcast, attributes
- Read/Write Lock (13)
  - Init, destroy, write lock/unlock, read lock/unlock, attributes: timeouts, ...

Pthreads APIs (cont’d)

- Thread Specific Storage (4)
- Signals (3)
  - Send a signal to thread, signal mask
- Unsupported extension (16)
  - Get/set scheduling policy, …
- Extension: semaphores (semaphore.h)
- The Pthreads API contains over 60 subroutines (excluding unsupported and extensions).
  - In total up to 100.
Header Files, Compiling and Linking

- Header files:
  
  ```
  #include <pthread.h>
  #include <sched.h>
  #include <semaphore.h> -- semaphore extension
  ```

- Compile with `gcc` and link to `pthreadlib`
  - for example:
    ```
    gcc -o test-hwAPI test-hwAPI.c
    hardwareAPI.o -l pthread -l socket
    ```
  - Extensions: `-lpthread` `-lsocket`

- Run as an ordinary executable code:
  ```
  ./test-hwAPI parameters
  ```

Pthread Naming Convention

- Types: `pthread[_object][_np]_t`
- Functions: `pthread[_object]_action[_np]`
- Constants and Macros: `PTHREAD_PURPOSE[_NP]`

- If the type of an object is not a thread, then:
  - `object` represents the type of object, e.g. attribute
  - `action` is an operation to be performed on the object
  - `np` or `NP` indicates that the name or symbol is a non
    portable extension to the API set,
  - `PURPOSE` indicates the use or purpose of the symbol.
**Primitive Data Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_attr_t</td>
<td>Thread creation attribute</td>
</tr>
<tr>
<td>pthread_cleanup_entry_np_t</td>
<td>Cancelation cleanup handler entry</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Condition variable creation attribute</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Condition variable synchronization primitive</td>
</tr>
<tr>
<td>pthread_jmpoption_np_t</td>
<td>Options structure for extensions to pthread_join()</td>
</tr>
<tr>
<td>pthread_key_t</td>
<td>Thread local storage key</td>
</tr>
<tr>
<td>pthread_mutexattr_t</td>
<td>Mutex creation attribute</td>
</tr>
<tr>
<td>pthread_mutex_t</td>
<td>Mutual exclusion synchronization primitive</td>
</tr>
<tr>
<td>pthread_once_t</td>
<td>Once time initialization control variable</td>
</tr>
<tr>
<td>pthread_opton_np_t</td>
<td>Thread run-time options structure</td>
</tr>
<tr>
<td>pthread_rwlock_t</td>
<td>Read/Write lock attribute</td>
</tr>
<tr>
<td>pthread_rwlock_t</td>
<td>Read/Write synchronization primitive</td>
</tr>
<tr>
<td>pthread_t</td>
<td>Thread handle</td>
</tr>
<tr>
<td>pthread_id_np_t</td>
<td>Thread ID. For use as an integral type.</td>
</tr>
<tr>
<td>struct sched_param</td>
<td>Scheduling parameters (priority and policy)</td>
</tr>
</tbody>
</table>

**Thread Declaration and Creation**

Declaration: `pthread_t tid; /* thread */`

Creation:

```c
int pthread_create(pthread_t* tid,
                    const pthread_attr_t* attr,
                    void* (*start_routine)(void *),
                    void *arg);
```

- Analogous to a combined `fork` and `exec` routine
- The routine creates a thread with the specified thread attributes.
- When `attr` is `NULL` the default thread attributes are used.
- Returns a thread id in `tid`.
- The new thread begins execution by calling `start_routine` with a single argument `arg`.

* For example:

```c
pthread_create (&tid, &attr, (void *) tf, (void *) &i);
```
Thread Identifiers

- When created, a thread is assigned a unique thread identifier (handle) that is used to reference the thread
  
  \[
  \text{pthread_t tid;} \\
  \quad \ldots \\
  \text{pthread_create(&tid, ...) } \\
  \quad \ldots \\
  \text{pthread_join(tid, NULL);} \\
  \]

- A thread can get its own handle:
  
  \[
  \text{pthread_t mytid = pthread_self()} \\
  \]
  
  - returns the handle of the calling thread

- To compare two threads, use
  
  \[
  \text{pthread_equal(tid1, tid2)} \\
  \]
  
  - returns 0 if different
  - Should not use == to compare thread IDs.

Thread Termination

A thread terminates execution in two ways

- Explicitly by executing
  
  \[
  \text{void pthread_exit(void* return_value);} \\
  \]
  
  - \text{return_value} – a single return value or NULL
  - Analogous to \text{exit}
  - If the current thread is the last thread then the process terminates
  - The \text{exit} routine kills all threads and exits the process

- Implicitly by return from the thread routine
  
  \[
  \text{return(status)} \\
  \]
  
  - Return from the thread routine is equivalent to calling \text{pthread_exit}
  - Return from the initial thread main is the equivalent to calling \text{exit}
Join a Thread

- A parent thread can wait for a child to terminate, i.e. join the child by executing:

  ```c
  int pthread_join(pthread_t tid, void** value);
  ```

  - `tid` – child’s descriptor
  - `value` – the address of the return value
  - Analogous to wait
  - Must specify thread. There is no wait any.
  - Current thread blocks until thread terminates
  - The return value of thread is returned in `value`
  - All threads must be either detached or joinable.

Detached Threads

- Threads that are not joinable are called detached threads.
  - A parent does not need to join a child.
  - When a detached thread terminates, it is destroyed.
Detached Threads (cont’d)

- A thread can be created as "detached"
  ```c
  pthread_attr_setdetachstate(&attr,
      PTHREAD_CREATE_DETACHED)
  pthread_create(&tid, &attr, ...)
  ```
- A thread can be “detached” explicitly by calling
  ```c
  int pthread_detach(pthread_t tid);
  ```

Synchronization Mechanisms in Pthreads

- **Mutexes** – mutual exclusion locks
- **Condition variables** – queues of thread (used for condition synchronization)
- **Reader/Writer locks** – shared read and exclusive write locks
- **Semaphores** – an extension (POSIX 1b) to Pthreads API implemented on top of mutexes and condition variables:
  ```c
  #include <semaphore.h>
  Link -lposix4
  ```
Mutexes

- **Mutex** is an object of the `pthread_mutex_t` data type used to protect critical sections of code, i.e. for mutual exclusion
  - Acts like a lock protecting access to shared data within a critical section.
  - Before used, must be initialized.
  - Set/get mutex attributes.
  - Lock, unlock and trylock operations.
  - Should be destroyed when no longer needed.
- Declaration:
  ```c
  pthread_mutex_t mutex;
  ```

Functions on Mutex

- `pthread_mutex_init(&mutex, &attr)`
  - Create and initialize a new mutex object, set its attributes.
  - The mutex is initially unlocked.
- `pthread_mutex_destroy(mutex)`
  - Destroy the mutex when no longer needed
- `pthread_mutex_lock(&mutex)`
  - Lock the mutex. If it is already locked, the call blocks the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock(&mutex)`
  - Attempt to lock a mutex. Returns non-zero value if the mutex is already locked, otherwise returns 0.
- `pthread_mutex_unlock(&mutex)`
  - Unlock the mutex (if called by the owning thread)
Typical Usage

- Create and initialize mutex, ..., spawn threads, ..., lock the mutex, critical section, unlock the mutex, ... destroy the mutex
- For example:
  ```c
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex, NULL);
  ...
  pthread_mutex_lock(&mutex)
  critical section;
  pthread_mutex_unlock(&mutex)
  ```

Condition Variables

- A condition variable (a.k.a. queue variable) is an object of the `pthread_cond_t` data type used for blocking and resuming threads holding mutexes
  - For condition synchronization.
  - A condition variable is always used in conjunction with a mutex lock.
- Declaration
  ```c
  pthread_cond_t cond;
  ```
- Operations on condition variables:
  - wait, signal, broadcast
  - Signal-and-continue signaling discipline
Functions on Condition Variables

- `pthread_cond_init(&cond, &attr)`
  - Create and initialize a new condition variable.
- `pthread_cond_destroy(&cond)`
  - Free the condition variable that is no longer needed.
- `pthread_cond_wait(&cond, &mutex)`
  - Wait on the condition variable, i.e. release the mutex (if owner) and place the calling thread to the tail on the cond var queue.
- `pthread_cond_signal(&cond)`
  - Signal the condition variable, i.e. move a waiting thread (if any) from the head of the cond var queue to the mutex queue.
- `pthread_cond_broadcast(&cond)`
  - Signal all: Awaken all threads waiting on the condition variable.

Typical Usage of A Condition Variable

- Take an action when the counter, `x`, is zero

```c
action() {
    ... // ...
    pthread_mutex_lock(&m);
    while (x != 0)
        pthread_cond_wait(&cv, &m);
    take_action();
    pthread_mutex_unlock(&m);
    ... // ...
}
```

```c
counter() {
    ... // ...
    pthread_mutex_lock(&m);
    x--; // ...
    if (x==0)
        pthread_cond_signal(&cv);
    pthread_mutex_unlock(&m);
    ... // ...
}
```
Example: Summing Matrix Elements

```c
#include <pthread.h>
#include <stdio.h>
#define MAXSIZE 2000 /* maximum size */
#define MAXWORKERS 4 /* maximum number of workers */

pthread_mutex_t barrier; /* lock for the barrier */
pthread_cond_t go; /* condition variable */
int numWorkers; /* number of worker threads */
int numArrived = 0; /* number who have arrived */

/* a reusable counter barrier */
void barrier() {
    pthread_mutex_lock(&barrier);
    numArrived++;
    if (numArrived < numWorkers)
        pthread_cond_wait(&go, &barrier);
    else {
        numArrived = 0; /* last worker awakens others */
        pthread_cond_broadcast(&go);
    }
    pthread_mutex_unlock(&barrier);
}

void *Worker(void *); /* size == stripSize*numWorkers */
int sum[MAXSIZE]; /* sum computed by each worker */
int matrix[MAXSIZE][MAXSIZE];
```

### Summing Matrix Elements (cont’d)

/* read command line, initialize, and create threads */
int main(int argc, char *argv[]) {  
    int i, j;
    pthread_attr_t attr;
    pthread_t workerid[MAXWORKERS];
    /* set global thread attributes */
    pthread_attr_init(&attr);
    pthread_attr_setscope(attr, PTHREAD_SCOPE_SYSTEM);
    /* initialize mutex and condition variable */
    pthread_mutex_init(&barrier, NULL);
    pthread_cond_init(&go, NULL);
    /* read command line */
    size = atoi(argv[1]);
    numWorkers = atoi(argv[2]);
    stripSize = size/numWorkers;
    /* initialize the matrix */
    for (i = 0; i < size; i++)
        for (j = 0; j < size; j++)
            matrix[i][j] = 1;
    /* create the workers, then exit main thread */
    for (i = 0; i < numWorkers; i++)
        pthread_create(&workerid[i], attr,
                        Worker, (void *) i);
    pthread_exit(NULL);
}
Summing Matrix Elements (cont’d)

```c
/* Each worker sums the values in one strip. */
void *Worker(void *arg) {
  int myid = (int) arg;
  int total, i, j, first, last;
  /* determine first and last rows of my strip */
  first = myid*stripSize;
  last = first + stripSize - 1;
  /* sum values in my strip */
  total = 0;
  for (i = first; i <= last; i++)
    for (j = 0; j < size; j++)
      total += matrix[i][j];
  sums[myid] = total;
  Barrier();
  if (myid == 0) { /* worker 0 computes the total */
    total = 0;
    for (i = 0; i < numWorkers; i++)
      total += sums[i];
    printf("the total is %d\n", total);
  }
}
```

Semaphore Extension

- Implemented with mutexes and condition variables
  
  ```c
  #include <semaphore.h>
  ```
  - Defines semaphore types and functions
- A semaphore is an object of the `sem_t` type that can take any nonnegative value and can be altered by P (decrement) and C (increment) operations
  - If the semaphore is zero, the P operation blocks the calling thread until the semaphore is positive.
Semaphores (cont’d)

- Declaration: `sem_t sem;

- Functions
  - `set_wait(&sem)` – `P(sem)` – decrements the semaphore if it’s positive, otherwise blocks the process until the semaphore is positive
  - `sem_post(sem)` – `V(sem)` – increments the semaphore

- For example:
  ```c
  sem_init(&sem, SHARED, 1); /* sem = 1; */
  ...
  sem_wait(&sem); /* P(sem) */
  Critical section;
  sem_post(&sem); /* V(sem) */
  ```

Thread-Safe (Reentrant) Functions

- A multithreaded program must use thread-safe (reentrant) versions of standard functions.
  - A re-entrant function can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other.
  - A thread-safe function is either re-entrant or protected from multiple simultaneous execution by some form of mutual exclusion.

- To get declared reentrant (thread safe) versions of standard functions, define the _REENTRANT macro before any include:
  ```c
  #ifndef _REENTRANT
  #define _REENTRANT
  #endif
  #include <stdio.h>
  #include <pthread.h>
  ...
  ```
Introduction to MPI

MPI: Message Passing Interface

- **MPI is a message-passing library specification**
  - message-passing model
  - not a compiler specification
  - not a specific product
- For multiprocessor, clusters, and heterogeneous networks
- Designed
  - to permit the development of parallel software libraries
  - to provide access to advanced parallel hardware for end users, library writers, and tool developers
An MPI Library

- MPI is a de facto standard for message passing systems
  - Version 1 most widely used
  - Other common systems: PVM, p4, Parmacs
- Language bindings
  - Fortran
  - C
    - all MPI functions and constants have prefix `MPI_`
    - an MPI data type is specified for each C data type (see `mpi.h`)

MPI Library: Large or Small?

- **MPI is large**: about 130 functions
  - extensive functionality and flexibility
    - message passing (point-to-point, collective)
    - process groups and topologies
    - sizing
- **MPI is small**: 6 functions allow writing many programs
  
  - `MPI_Init`, `MPI_Finalize`,
  - `MPI_Comm_size`, `MPI_Comm_rank`,
  - `MPI_Send`, `MPI_Recv`
  
  - or another 6-function set:
    
    - `MPI_Init`, `MPI_Finalize`,
    - `MPI_Comm_size`, `MPI_Comm_rank`,
    - `MPI_Bcast`, `MPI_Reduce`
C Data Types Mapping in MPI

<table>
<thead>
<tr>
<th>MPI data type</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

MPI Processes

• *An MPI application* consists of a static set of processes communicating by sending and receiving tagged messages in process groups.

  mpirun -np 2 hello

  • Starts two hello tasks on two processors

• *MPI process* is a smallest unit of computation, e.g. a Unix process (a program assigned to a node).
  – Uniquely identified by an index (*rank*) in a given process group.

• Processes communicate with tagged messages within process groups identified by *communicators* (communication contexts)
Programming, Building and Running an MPI Application

- Install one of the MPI implementations, e.g. MPICH
- Develop your application – one program for all processes
  - The number of processes is specified at run time (could be 1)
  - Use the number of proc and proc ranks to determine process tasks
  - In C: `#include <mpi.h>`
- Compile:
  ```c
  mpicc -o myprog myprog.c
  ```
  - For large projects, develop a Makefile (see for example, `mpich/examples/Makefile.in`)
- Run:
  ```bash
  mpirun -np 2 myprog
  ```
  - The option `-help` shows all options to mpirun.

Basic MPI Functions

- `int MPI_Init(int *argc, char **argv[])`
  - Start MPI, enroll the process in the MPI application
- `int MPI_Finalize()`
  - Stop (exit) MPI
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Determine the number of processes in the group `comm`
    - `comm` – communicator, e.g. `MPI_COMM_WORLD`
    - `size` – number of processes in group (returned)
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Determine the rank of the calling process in the group `comm`
    - `comm` – communicator, e.g. `MPI_COMM_WORLD`
    - `rank` – the rank (returned) is a number between zero and `size-1`
Example: “Hello World”

```c
#include "mpi.h"
#include <stdio.h>
int main( argc, argv )
    int argc;
    char **argv;
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello world! I’m %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Basic (Blocking) Send

```c
int MPI_Send(void *buf, int count, MPI_datatype dt,
                int dest, int tag, MPI_Comm comm)
    - Sends a message with the tag tag to the process dest in the context comm
      • buf — send buffer
      • count — number of items in buffer
      • dt — data type of items
      • dest — destination process rank
      • tag — message tag
      • comm — communicator

    - Returns integer result code as for all MPI functions, normally MPI_SUCCESS
    - datatype can be elementary, continues array of data types, stridden blocks of data types, indexed array of blocks of data types, general structure.
```
Basic (Blocking) Receive

```c
int MPI_Recv(void *buf, int count, MPI_datatype dt,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

- Receive a message with the tag tag from the process source in the context comm
  - buf: receive buffer (loaded)
  - count: max number of entries in buffer
  - dt: data type of entries
  - source: source process rank
  - tag: message tag
  - comm: communicator
  - status: status (returned).

Wildcard Sources and Tags in Receive

- “Wildcard” values are provided for tag (MPI_ANY_TAG) and source (MPI_ANY_SOURCE).
  - If `source = MPI_ANY_SOURCE`, then the receiving function will accept a message from any process that sent the message to the calling process.
  - If `tag = MPI_ANY_TAG`, then the receiving function will accept any message from the process specified by `source`. 
Inspecting Received Message

- The received message can be inspected via `status` structure that has three components `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`

  ```c
  MPI_Status status;
  MPI_Recv(..., &status);
  ... status.MPI_TAG;
  ... status.MPI_SOURCE;
  MPI_Get_count(&status, datatype, &count);
  ```

- `MPI_Get_count` may be used to determine how much data of a particular type has been received.

Non-Blocking Communication Operations

- Non-blocking calls (send and receive) initiate communication
- Non-blocking send

  ```c
  int MPI_Isend(void* buf, int count,
                 MPI_Datatype datatype, int dest, int tag,
                 MPI_Comm comm, MPI_Request *request)
  ```

- Non-blocking receive

  ```c
  int MPI_Irecv(void* buf, int count,
                MPI_Datatype datatype, int source, int tag,
                MPI_Comm comm, MPI_Request *request)
  ```

- A request object is returned in `request` to identify the operation (to query the status of communication or wait for its completion.)
Complete Communication

```c
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
    – Returns when the operation identified by request completes.

int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
    – is essentially a MPI_WAIT that returns immediately,
      with flag = true if the operation identified by request has completed.
```

Example: Non-Blocking Send/Receive

```c
MPI_Comm_rank(comm, &rank);
if (rank = 0)
    {
        MPI_Isend(a, 10, MPI_REAL, 1, tag, comm, request);
        /**** do some computation to mask latency ****/
        MPI_Wait(request, &status);
    }
if (rank = 1)
    {
        MPI_Irecv(a, 10, MPI_REAL, 0, tag, comm, request);
        /**** do some computation to mask latency ****/
        MPI_Wait(request, &status);
    }
```
Probing for Pending Messages

MPI_Iprobe(source, tag, comm, flag, status)
- polls for pending messages
MPI_Probe(source, tag, comm, status)
- returns when a message is pending

- Non-blocking/blocking check for an incoming message without receiving it

Collection Operations

- A collective operation is executed by having all processes in the group call the same communication routine with matching arguments.
  - Several collective routines have a single originating or receiving process -- the root.
  - Some arguments in the collective functions are specified as “significant only at root”, and are ignored for all participants except the root.
Collective Communication

- Global synchronization
  \[
  \text{int MPI\_Barrier(MPI\_Comm comm)}
  \]

- Broadcast from buf of root to all processes
  \[
  \text{int MPI\_Bcast(void *buf, int count, MPI\_datatype dt, int root, MPI\_Comm comm)}
  \]

- Collective data movement
  \[
  \begin{align*}
  &\text{int MPI\_Gather( void *sendbuf, int sendcount,} \\
  &\quad \text{MPI\_datatype sendtype, void *recvbuf,} \\
  &\quad \text{int recvcount, MPI\_datatype recvtype,} \\
  &\quad \text{int root, MPI\_Comm comm)} \nonumber \\
  &\text{int MPI\_Scatter(void *sendbuf, int sendcount,} \\
  &\quad \text{MPI\_datatype sendtype, void *recvbuf,} \\
  &\quad \text{int recvcount, MPI\_datatype recvtype,} \\
  &\quad \text{int root, MPI\_Comm comm)}
  \end{align*}
  \]

Collective Move Operations

In the diagram, data moves from one process to another using the MPI_Barrier, MPI_Bcast, MPI_Gather, and MPI_Scatter functions.
Collective Move Operations (cont’d)

Example

Gather 100 integers from every process in group to root.

```c
MPI_Comm comm;
    int gsize, sendarray[100];
    int root, myrank, *rbuf;
    ...
    MPI_Comm_rank(comm, &myrank);
    if (myrank == root) {
        MPI_Comm_size(comm, &gsize);
        rbuf = (int *)malloc(gsize*100*sizeof(int));
    }
    MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100,
                 MPI_INT, root, comm);
```
Global Reduction Operations

- Reduce from sendbufs into recvbuf in root using operation op
  
  ```c
  int MPI_Reduce(void *sendbuf, void *recvbuf,
                 int count, MPI_datatype dt,
                 MPI_Op op, int root, MPI_Comm comm)
  ```

- Reduce from sendbufs into recvbufs using operation op
  
  ```c
  int MPI_Allreduce(void *sendbuf, void *recvbuf,
                    int count, MPI_datatype dt,
                    MPI_Op op, int root, MPI_Comm comm)
  ```

- Available operations include:
  ```c
  MPI_MAX, MPI_MIN, MPI_SUM, MPI_LAND, MPI_BOR, ...
  ```

Timing

- `double MPI_Wtime(void)`
  - Returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The time is “local” on the host.

- `double MPI_Wtick(void)`
  - Returns the resolution of MPI_WTIME in seconds, the number of seconds between successive clock ticks.

- Example of usage:
  ```c
  {
    double starttime, endtime;
    starttime = double MPI_Wtime();
    .... stuff to be timed ...
    endtime = double MPI_Wtime();
    printf("That took %f seconds\n",
           endtime-starttime);
  }
  ```
MPI Program Example: Compute PI

```c
#include "mpi.h"
#include <math.h>
int main(argc, argv)
    int argc;
    char *argv[];
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    Compute PI (cont’d)
    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        h = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += 4.0 / (1.0 + x*x);
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
}
```