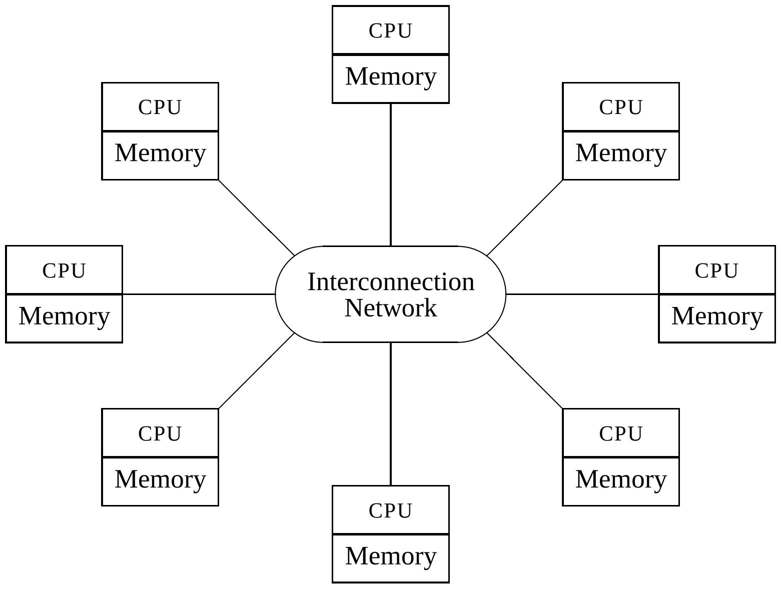
Message-Passing Programming

Introduction

- MPI – Message Passing Interface standard
  - Most popular message-passing specification to support parallel programming
  - Standardized and portable to function on a wide variety of parallel computers
  - Allowed for the development of portable and scalable large-scale parallel applications

Message-passing model

- Similar to task/channel model

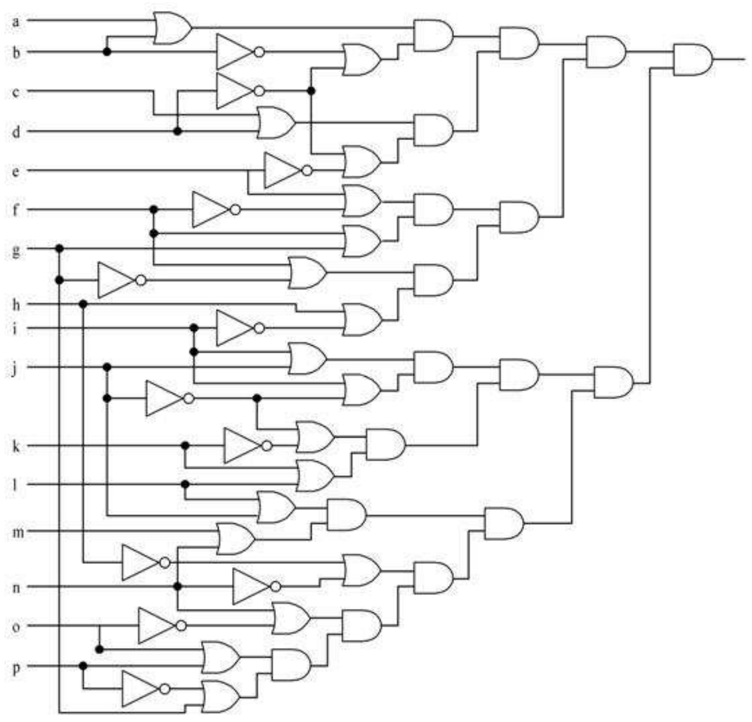


- Underlying hardware a collection of processors, each with its own local memory
  - Processor can access only its own instructions and data in its local memory
  - Message passing between processors supported by an interconnection network
  - Local data sent by PE A to PE B giving B indirect access to those values
- Implicit channel between every pair of processors
  - Use the network design strategies to minimize the communications overhead
- User specifies the number of concurrent processes when the program begins
  - Typically, the number of active processes remains constant throughout the execution of program
  - Processes are independent and may perform different functions
  - Process alternately performs computations on local variables and communicates with other processes/I/O devices
- Processes pass messages to communicate and synchronize with each other
- Advantages of message passing model over other parallel programming models
  - Runs well on a wide variety of MIMD architectures

    * Allows programmers to manage memory hierarchy
    * Natural fit for multicomputers that do not share global address space
    * Possible to execute message-passing programs using shared variables as message buffers
  - Encourages the use of local memory in the design of algorithms
    * Maximize local computation and minimize communications
    * Remote memory entails communications overhead
    * High cache-hit-rates on multicomputers for good performance
  - Portable to many architectures
  - Debugging message-passing programs is simpler than debugging shared-variable programs
    * Processes cannot accidentally overwrite a variable controlled by another process
    * Easier to create deterministic programs

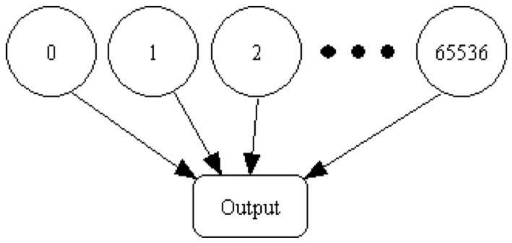
Circuit satisfiability

- Implement a program to compute whether a circuit is satisfiable (yields 1 for some combination of inputs)
  - Important for the design and verification of logical devices
  - NP-complete
  - Consider the following circuit:



$$
\begin{array}{r}
((((a \vee b) \wedge (\bar{b} \vee \bar{d})) \wedge ((c \vee d) \wedge (\bar{d} \vee \bar{e}))) \wedge \\
((e \vee \bar{f}) \wedge (f \vee g)) \wedge ((f \vee \bar{g}) \wedge (h \vee \bar{i}))) \wedge \\
(((i \vee j) \wedge (i \vee \bar{j})) \wedge ((\bar{j} \vee \bar{k}) \wedge (\bar{k} \vee l))) \wedge \\
((j \vee l) \wedge (m \vee n)) \wedge (\bar{h} \vee \bar{n}) \wedge (n \vee \bar{o}) \wedge \\
((o \vee p) \wedge (g \vee \bar{p}))
\end{array}
$$

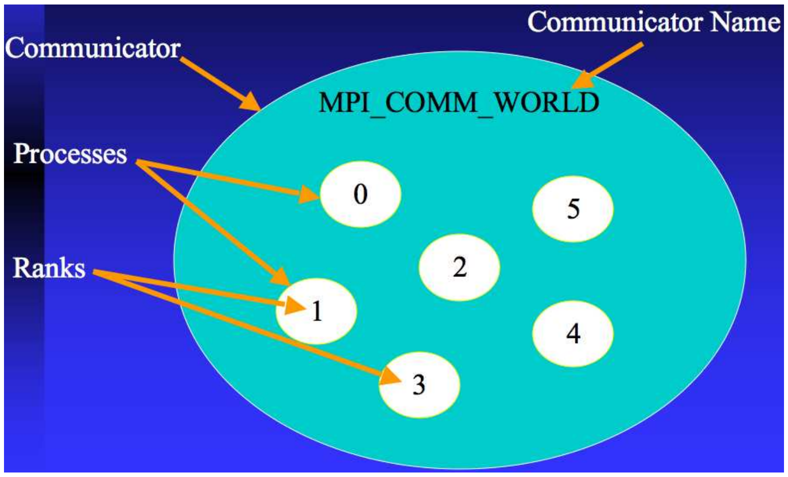
- Solve the problem by trying every combination
  - For a circuit with n inputs, you have to try 2^n combinations
- Solve by partitioning, or functional decomposition
  - Associate one task with each combination of inputs
  - If a task finds that its combination of inputs causes the circuit to return the value 1, it prints the combination
  - Independent tasks imply that satisfiability checks may be performed in parallel
- No channels between tasks



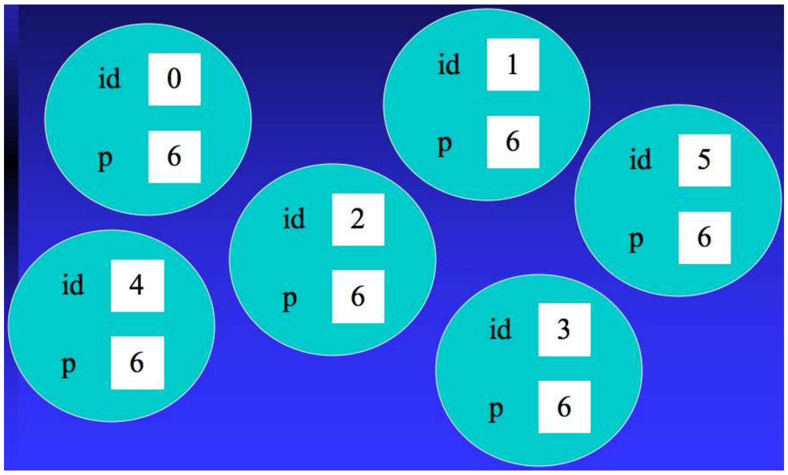
  - Embarrasingly parallel
  - Any of the tasks may produce an output
    - A channel from each task to output device
- Agglomeration and mapping
  - Fixed number of tasks with no communication between tasks
  - Variable time for each task to complete
    - Most of the tasks represent bit combinations for which the circuit is not satisfiable
    - Some tasks may give up quickly; other tasks may take longer
  - Map tasks to processors in a cyclic fashion to balance computational load
  - Minimize process creation time
    - One process per processor
    - n tasks for p processors
    - Cyclic/interleaved allocation
      - Assign each process pth task in round robin fashion
      - Distribution with $n = 20$ and $p = 6$
      - Task k is assigned to process $k\%p$
  - Code in `csat/csat1.c` on `stovokor`
    - Each active process executes its own copy of this program
    - Each MPI process has its own copy of all the active variables declared in the program
- Function `MPI_Init`
  - First MPI function call made by every MPI process; must be called before any other MPI function
    - The only exception is the function `MPI_Initialized` to check if MPI has been initialized
  - Do any set up needed for further calls to MPI library
  - All MPI identifiers, including function identifiers, begin with prefix `MPI_`, followed by a capital letter and a series of lowercase letters and underscores
  - All MPI constants are strings of capital letters and underscores beginning with `MPI_`

```
int MPI_Init ( &argc, &argv );
```

- Function `MPI_Comm_rank` and `MPI_Comm_size`
  - After initialization, every active process is a member of a communicator called `MPI_COMM_WORLD`



  - Communicator
    * Opaque object to provide the environment for message passing among processes
    * `MPI_COMM_WORLD` is the default communicator though you can also create your own communicators
  - Rank
    * Processes within a communicator are ordered, with rank providing their position in overall order
    * For p processes, the rank is given by a unique number between 0 and $p - 1$
    * Process uses its rank to determine its identity and to determine its portion of computation/dataset
    * Process identifies its own rank by
      `int MPI_Comm_rank ( MPI_COMM_WORLD, int * id );`
    * Total number of processes in a communicator is determined by
      `int MPI_Comm_size ( MPI_COMM_WORLD, int * num_procs );`



- Function `MPI_Finalize`
  - After a process has finished all MPI library calls, it calls `MPI_Finalize` to release all resources allocated to MPI, such as memory

```
int MPI_Finalize();
```

- Compiling MPI programs
  - Use the command `mpicc` as

    ```
    mpicc -o csat csat1.c
    ```
- Running MPI programs
  - Use the command `mpirun`

    ```
    mpirun -np 10 csat
    ```

Introducing collective communication

- Count the number of solutions found
  - Keep a count of solutions for each process
  - Compute the global sum of those values
  - Processors need to cooperate with each other to compute global sums
- Collective communication
  - Group of processes work together to distribute/gather a set of one or more values
  - Reduction operation
  - New code in `csat/csat2.c`
- Function `MPI_Reduce`
  - Performs one or more reduction operations on values submitted by all processes in communicator

    ```
    int MPI_Reduce ( void * operand, void * result, int count, MPI_Datatype type,
            MPI_Op operator, int root, MPI_Comm communicator );
    ```
  - `operand` is location of first element for reduction
  - `count` is the number of reductions to be performed
    - Each process submits `count` values
    - Each of submitted values is a list element for a different reduction
    - If `count` > 1, list elements for all reductions occupy a contiguous block of memory
  - `type` designates the type of elements being reduced
  - `operator` indicates the type of reduction to perform
  - `root` gives the rank of process that will have result of all reductions
  - `result` points to location of first reduction result
    - Is meaningful only for `root` process
    - Only a single process gets the global result; every process must call `MPI_Reduce`
    - If not every process participates, the program will hang

Benchmarking parallel performance

- Functions `MPI_Wtime` and `MPI_Wtick`
  - Look at wall clock time
  - Better results by ignoring the overheads like initiating MPI processes, establishing communications sockets, performing I/O on sequential device

  - Concentrate on the *middle area* between reading dataset and printing results – the actual computation time
  - `MPI_Wtime` returns the number of seconds elapsed since some point
  - `MPI_Wtick` returns the precision of the result returned by `MPI_Wtime`
  - Headers are:

    ```
    double MPI_Wtime();
    double MPI_Wtick();
    ```

  - Benchmark by enclosing the code between a pair of calls to `MPI_Wtime`, and taking the difference between the two times
  - Caveats
    * Technically, every MPI process does not start to execute at exactly the same time
    * This can throw off timing significantly
    * If there is a need to synchronize, such as `MPI_Reduce`, no process may complete until all processes have reached this point
    * Some processes may report significantly longer computation time than the latecomers

- Function `MPI_Barrier`
  - Barrier synchronization before first call to `MPI_Wtime`
  - No process can proceed past a barrier until all processes have reached it
  - Barrier ensures that all processes get into the measured section of the code at the same time

    ```
    int MPI_Barrier ( MPI_Comm comm );
    ```

  - See `csat3.c` for code
  - Run `csat3` with different number of processors to benchmark