CA3404: Winter 2018
Distributed and Parallel Algorithms

Week - 2 : Abstract models of parallel computing, PRAM (Parallel Random Access Machine)

Course email-id: dispalgo2018@gmail.com

Dt: 19-01-2018, 24-01-2018

Parallel Programming Models: Overview

There are several parallel programming models in common use:

- Shared Memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Parallel Programming Models: Overview

Parallel programming models exist as an abstraction above hardware and memory architectures.

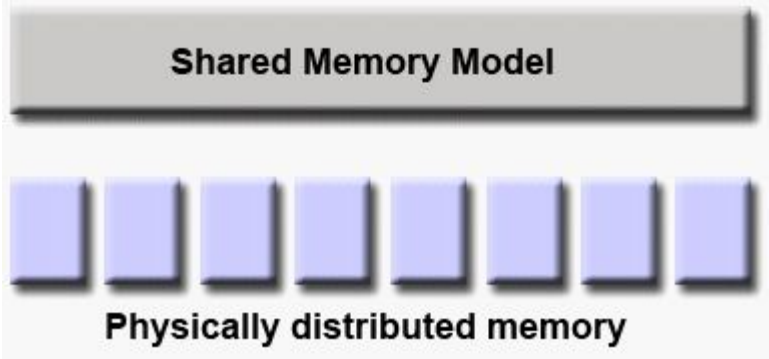
Although it might not seem apparent, these models are **NOT** specific to a particular type of machine or memory architecture.

In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples are discussed on the next slides:

Parallel Programming Models: Overview

SHARED memory model on a DISTRIBUTED memory machine:

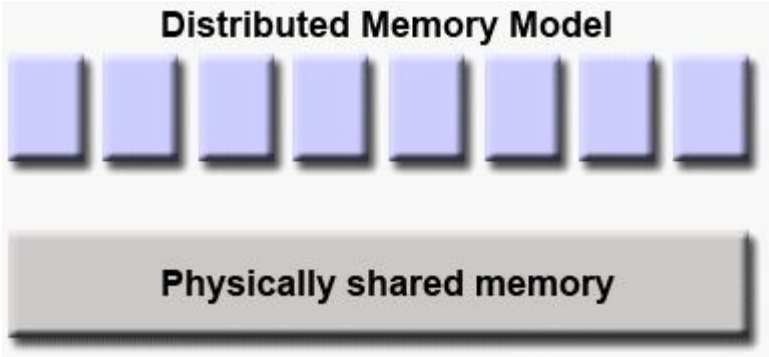
Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across networked machines, but appeared to the user as a single shared memory global address space. Generically, this approach is referred to as "virtual shared memory".



Parallel Programming Models: Overview

DISTRIBUTED memory model on a SHARED memory machine:

Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of shared memory architecture, where every task has direct access to global address space spread across all machines. However, the ability to send and receive messages using MPI, as is commonly done over a network of distributed memory machines, was implemented and commonly used.

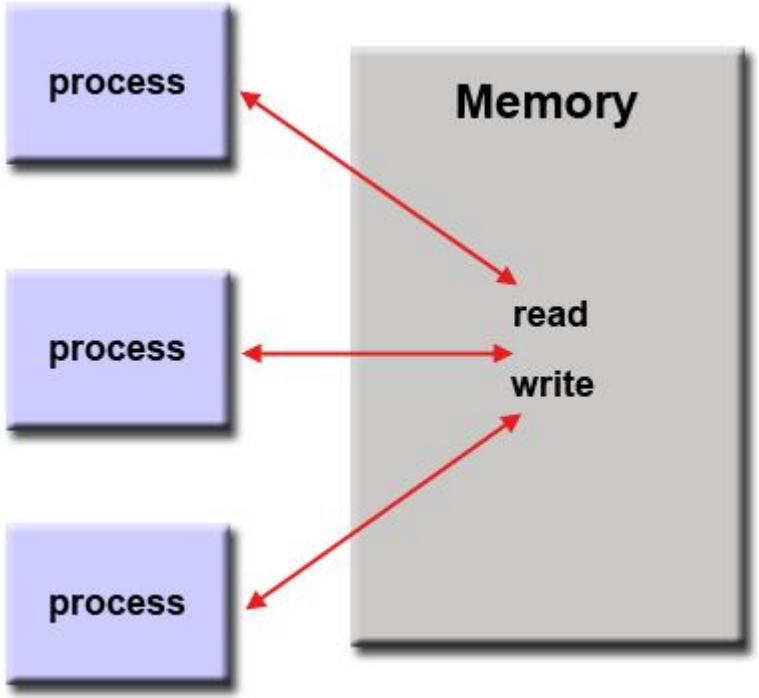


- **Which model to use?** This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.

- The following sections on the next slides describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model (without threads)

- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.



- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.

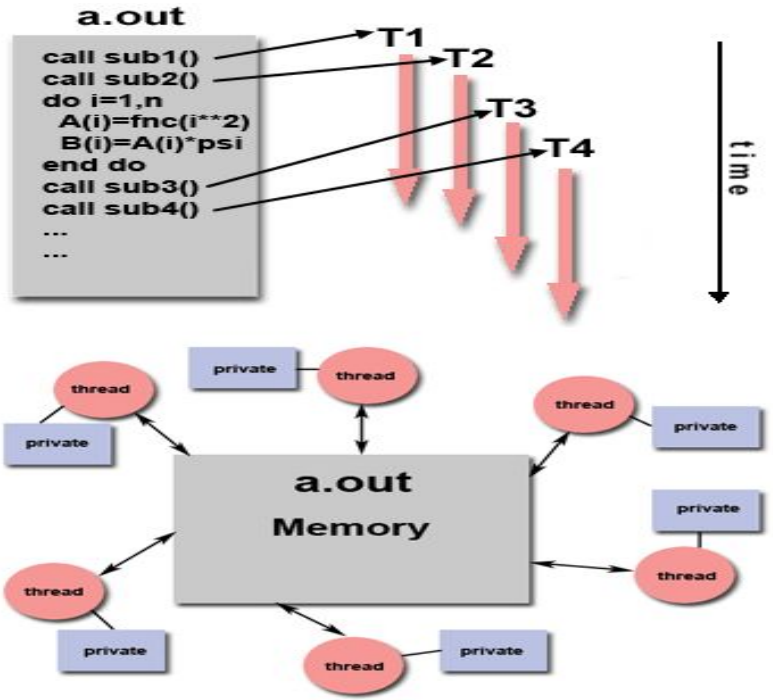
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data **locality**:

  -> Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.

  -> Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

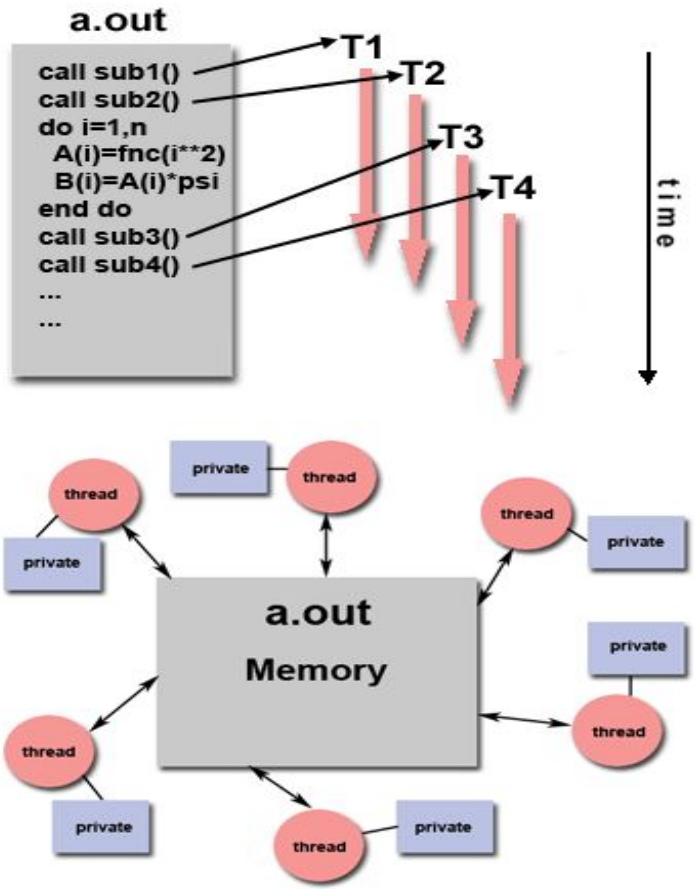
Shared Memory Model (with threads)

In the threads model of parallel programming, a single "heavy-weight" process can have multiple "light-weight", concurrent execution paths.For example

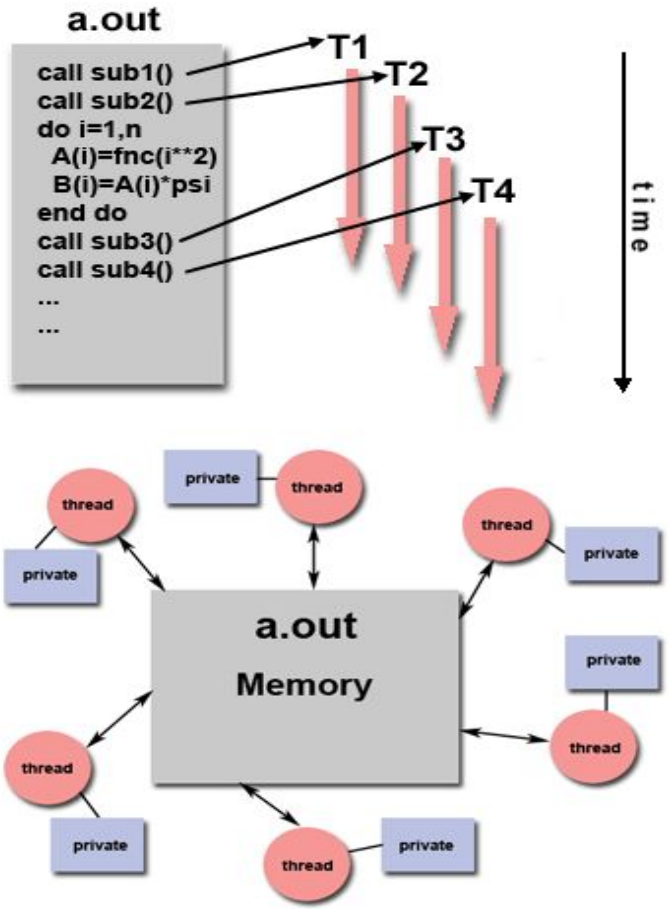


- The main program a.out is scheduled to run by the native operating system.
  a.out loads and acquires all of the necessa system and user resources to run.
  This is the "heavy-weight" process.
- a.out performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.

- Each thread has local data, but also, shares the entire resources of a.out. This saves the overhead associated with replicating a program resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of a.out.
- A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.

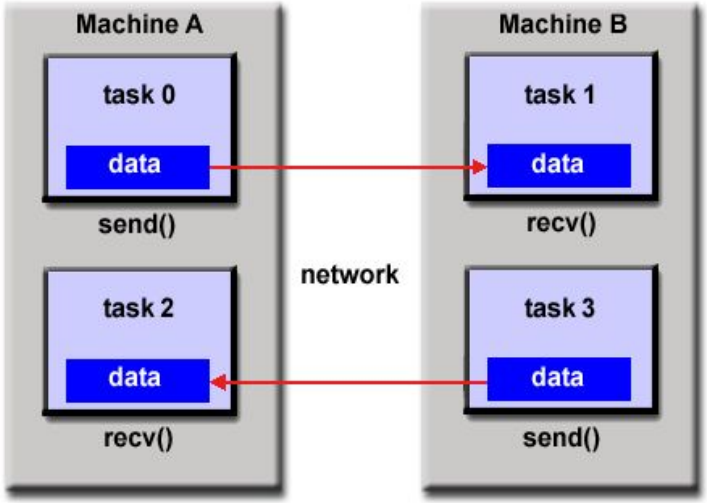


- Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
- Threads can come and go, but a.out remain present to provide the necessary shared resources until the application has completed.



Distributed Memory / Message Passing Model

This model demonstrates the following characteristics:



- ➔ A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
- ➔ Tasks exchange data through communications by sending and receiving messages.
- ➔ Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

More Information:

HomeWork:

Go through, and include into your notes

MPI tutorial: computing.llnl.gov/tutorials/mpi

A message-passing system implemented using two functions.

- **void send(int dst_pid, int data)**
  Sends a message containing the integer data to the processor whose ID is dst_pid. Note that the function's return may be delayed until the receiving processor requests to receive the data — though the message might instead be buffered so that the function can return immediately.
- **int receive(int src_pid)**
  Waits until the processor whose ID is src_pid sends a message and returns the integer in that message. This is called a blocking receive. Some systems also support a non-blocking receive, which returns immediately if the processor hasn't yet sent a message. Another variation is a receive that allows a program to receive the first message sent by any processor.

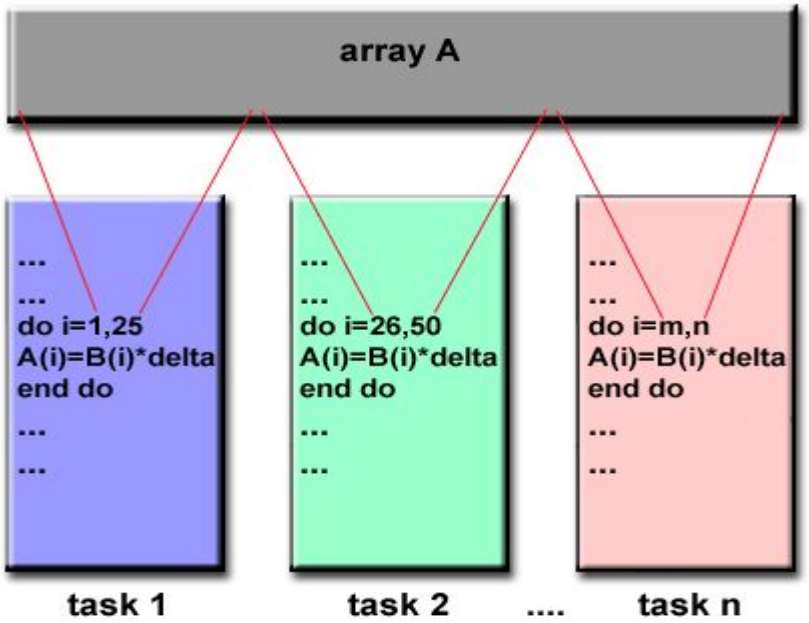
Data Parallel Model

May also be referred to as the Partitioned Global Address Space (PGAS) model.The data parallel model demonstrates the following characteristics:

- ➔ Address space is treated globally
- ➔ Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- ➔ A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- ➔ Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".

Data Parallel Model

- ➔ On shared memory architectures, all tasks may have access to the data structure through global memory.
- ➔ On distributed memory architectures, the global data structure can be split up logically and/or physically across tasks.



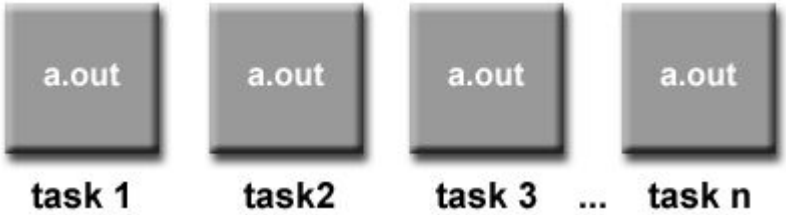
SPMD and MPMD

Single Program Multiple Data (SPMD):

-> "high level" programming model
-> built upon any combination of the previously mentioned parallel programming models
SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel.

MULTIPLE DATA: All tasks may use different data

Single Program Multiple Data contd..

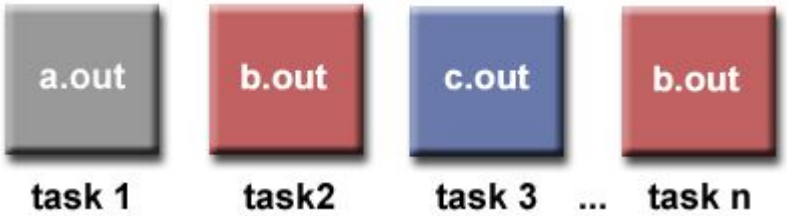


★ SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.

★ The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.

Multiple Program Multiple Data (MPMD):

- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel.



- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition (discussed later under Partioning).

RAM: A Model of Serial Computation

Random Access Machine is a favorite model of a sequential computer. Its main features are:

1. Computation unit with a user defined program.
2. Read-only input tape and write-only output tape.
3. Unbounded number of local memory cells.
4. Each memory cell is capable of holding an integer of unbounded size.
5. Instruction set includes operations for moving data between memory cells, comparisons and conditional branches, and simple arithmetic operations.
6. Execution starts with the first instruction and ends when a HALT instruction is executed.

RAM Model

7. All operations take unit time regardless of the lengths of operands.
8. Time complexity = the number of instructions executed.
9. Space complexity = the number of memory cells accessed.

PRAM Model

Parallel Random Access Machine is an idealized model of a shared memory SIMD(Single Instruction Multiple Data) machine. Its main features are:

1. Unbounded collection of numbered RAM processors P0, P1, P2,... (without tapes).
2. Unbounded collection of shared memory cells M[0], M[1], M[2],....
3. Each Pi has its own (unbounded) local memory (registers) and knows its index i.
4. Each processor can access any shared memory cell (unless there is an access conflict, see further) in unit time.
5. Input af a PRAM algorithm consists of n items stored in (usually the first) n shared memory cells.

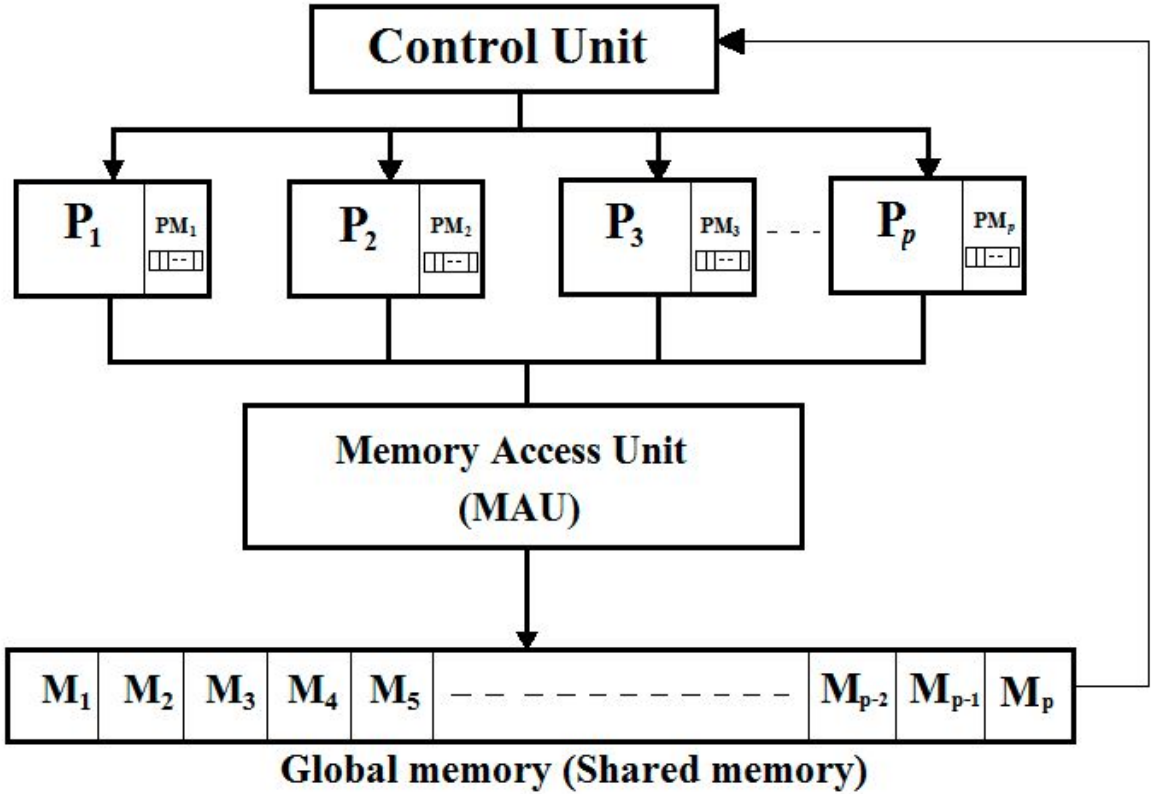
6. Output of a PRAM algorithm consists of n' items stored in n' shared memory cells.

7. PRAM instructions execute in 3-phase cycles.

a)Read (if any) from a shared memory cell.

b) Local computation (if any).

c)Write (if any) to a shared memory cell.



8. Processors execute these 3-phase PRAM instructions synchronously.

9. Special assumptions have to be made about R-R and W-W shared memory access conflicts.

10. The only way processors can exchange data is by writing into and reading from memory cells.

11. P0 has a special activation register specifying the maximum index of an active processor. Initially, only P0 is active, it computes the number of required active processors and loads this register, and then the other corresponding processors start executing their programs.

12. Computation proceeds until P0 halts, at which time all other active processors are halted.

13. Parallel time complexity = the time elapsed for P0's computation.

14. Space complexity = the number of shared memory cells accessed.

PRAM is an attractive and important model for designers of parallel algorithms. Why?

1. It is **natural**: the number of operations executed per one cycle on p processors is at most p.
2. It is **strong**: any processor can read or write **any** shared memory cell in unit time.
3. It is **simple**: it abstracts from any communication or synchronization overhead, which makes the complexity and correctness analysis of PRAM algorithms easier. Therefore,
4. It can be used as a **benchmark**: If a problem has no feasible/efficient solution on PRAM, it has no feasible/efficient solution on any parallel machine.
5. It is **useful**: it is an idealization of existing (and nowaday more and more abundant) shared memory parallel machines.

Constrained PRAM models

1. **Bounded** number of shared **memory cells**. This may be called a **small memory PRAM**. If the input data set exceeds the capacity of the shared memory, the input and/or output values can be distributed evenly among the processors.
2. **Bounded** size of a **machine word** and/or **memory cell**. This parameter is usually called **word size of PRAM**.
3. **Bounded** number of **processors**. This may be called a **small PRAM**. If the number of threads of execution is higher, processors may interleave several threads.
4. Constraints on simultaneous access to shared memory cells: handling **access conflicts**.

Handling shared memory access conflicts

Here, n number of processors can perform independent operations on n number of data in a particular unit of time. This may result in simultaneous access of same memory location by different processors.
To solve this problem, the following constraints have been enforced on PRAM model −

1. **Exclusive Read Exclusive Write (EREW) PRAM**: No two processors are allowed to read or write the same shared memory cell simultaneously.
2. **Concurrent Read Exclusive Write (CREW) PRAM**: Simultaneous reads of the same memory cell are allowed, but only one processor may attempt to write to an individual cell

3. **Concurrent Read Concurrent Write (CRCW) PRAM**: Both simultaneous reads and both simultaneous writes of the same memory cell are allowed.
Concurrent Read has a clear semantics, whereas Concurrent Write has to be further constrained. There exist several basic submodels:

a. **PRIORITY CRCW:** the processors are assigned fixed distinct priorities and the processor with the highest priority is allowed to complete WRITE.
b. **ARBITRARY CRCW**: one randomly chosen processor is allowed to complete WRITE. The algorithm may make no assumptions about which processor was chosen.
c. **COMMON CRCW:** all processors are allowed to complete WRITE iff all the values to be written are equal. Any algorithm for this model has to make sure that this condition is satisfied. If not, the algorithm is illegal and the machine state will be undefined.