2018 book, Ch1

Terminologies

Speedup (serial exec time over parallel exec time): $S = \frac{Ts}{Tp}$

Efficiency (speedup per processor): $E = \frac{S}{P} = \frac{(\frac{Ts}{Tp})}{P} = \frac{Ts}{(Tp*P)}$

Linear speedup means 100% efficiency

Cost: C = Tp * P //ideal case (linear speedup), C = Ts

Scalability: efficiency on varying number of processors and data size

- strongly scalable: regardless of input data size(N), increasing P shows fixed E
- weakly scalable: increasing P and N with same factor shows fixed E
  ex) doubling P and also doubling N shows fixed E

Computation-to-communication ratio:

the higher the better (for both speedup and efficiency) – means, less communication is better

Other issues:

Communication: collaboration among processes/threads to solve a problem

Synchronization: appropriate order/alignment when needed

Load balancing: to reduce the total execution time (Tp)

Run time analysis

global sum example for checking speedup, efficiency and scalability

Array A with n elements;

Compute $\sum_{i=0}^{n} A[i]$

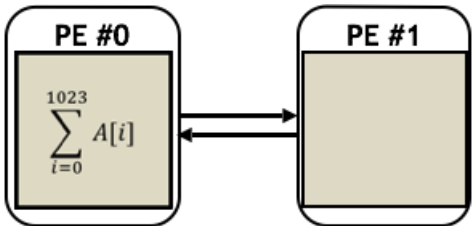
Assumptions:

each PE adds two numbers in the local mem. in 1 time unit;
PE sends data (any size) from local mem. to other PE's local mem. in 3 units;
Input/output – array A is stored in PE_0, result is gathered in PE_0;
Synchronization – all PEs operate in lock-step manner, i.e., either compute, communicate, or idle;

serial time: Ts (1, n) = n-1

with P=2, parallel time: Tp (2, n) = 3 + ($\frac{n}{2}$ - 1) + 3 + 1 = $\frac{n}{2}$ + 6

PE_0 sends half of A to PE_1 (3 units)
PE_0 and PE_1 compute sum of $\frac{n}{2}$ ($\frac{n}{2}$ -1 units)
PE_1 sends partial sum to PE_0 (3 units)
PE_0 adds two partial sums (1 unit)



With n=1024 and P=2, Ts (1, 1024) = 1023;

Tp (2, 1024) = 3+511+3+1 = 518

S = $\frac{\text{Ts}}{\text{Tp}}$ = $\frac{1023}{518}$ = 1.975

E = $\frac{\text{S}}{\text{P}}$ = $\frac{1.975}{2}$ = 98.75%

With n=1024 and P=4, Tp (4, 1024) = 3 + 3 + 255 + 3 + 1 + 3 + 1 = 269

PE_0 sends half of A to PE_1 (3 units)

PE_0 and PE1 sends half of A/2 to PE_2 and PE_3, respectively (3 units)

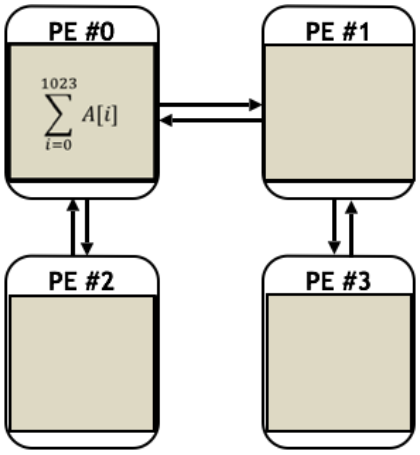
$PE_{0,1,2,3}$ computes local sum of 256 elements each (255 units)

PE_2 and PE_3 send local sums to PE_0 and PE_1, respectively (3 units)

PE_0 and PE_1 add local sums (1 unit)

PE_1 sends partial sum to PE_0 (3 units)

PE_0 adds partial sums (1 unit)



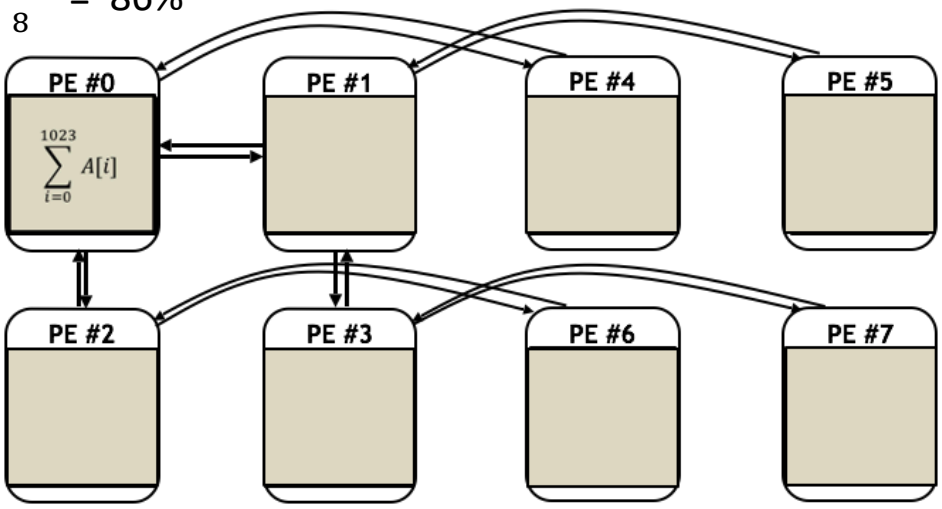
$$S = \frac{Ts}{Tp} = \frac{1023}{269} = 3.803$$

$$E = \frac{S}{P} = \frac{3.803}{4} = 95.07\%$$

With n=1024 and P=8, Tp(8, 1024) = 3+3+3+127+3+1+3+1+3+1 = 148

$$S = \frac{Ts}{Tp} = \frac{1023}{148} = 6.91$$

$$E = \frac{S}{P} = \frac{6.91}{8} = 86\%$$



As the number of processors increases, efficiency decreases, due to the communication overheads.

Generic formula for $P=2^q$ PEs and $n = 2^k$ (with the given assumptions):

Data distribution: $q * 3$ time units (where, $q = \log_2 P$)
Computing local sums: $\frac{n}{P} - 1 = 2^{k-q} - 1$
Collecting partial sums: $q * 3$
Adding partial sums: q

→ $Tp(P, n) = Tp(2^q, 2^k) = 3q + 2^{k-q} - 1 + 3q + q = \underline{(2^{k-q} - 1) + 7q}$

where, computation time = $2^{k-q} - 1 = \frac{n}{P} - 1$
communication time = $7q = 7 * \log P$

Scalability checking:

Definitions:

Increasing P regardless of n yields same efficiency → strongly scalable

Increasing P and n with same factor yields same efficiency → weakly scalable

Given solution program is weakly scalable.

Q: Prove that the given solution program is weakly scalable.
Hint: start from $Ts = n-1$, $Tp = \frac{n}{P} - 1 + 7 * \log P$, compute E

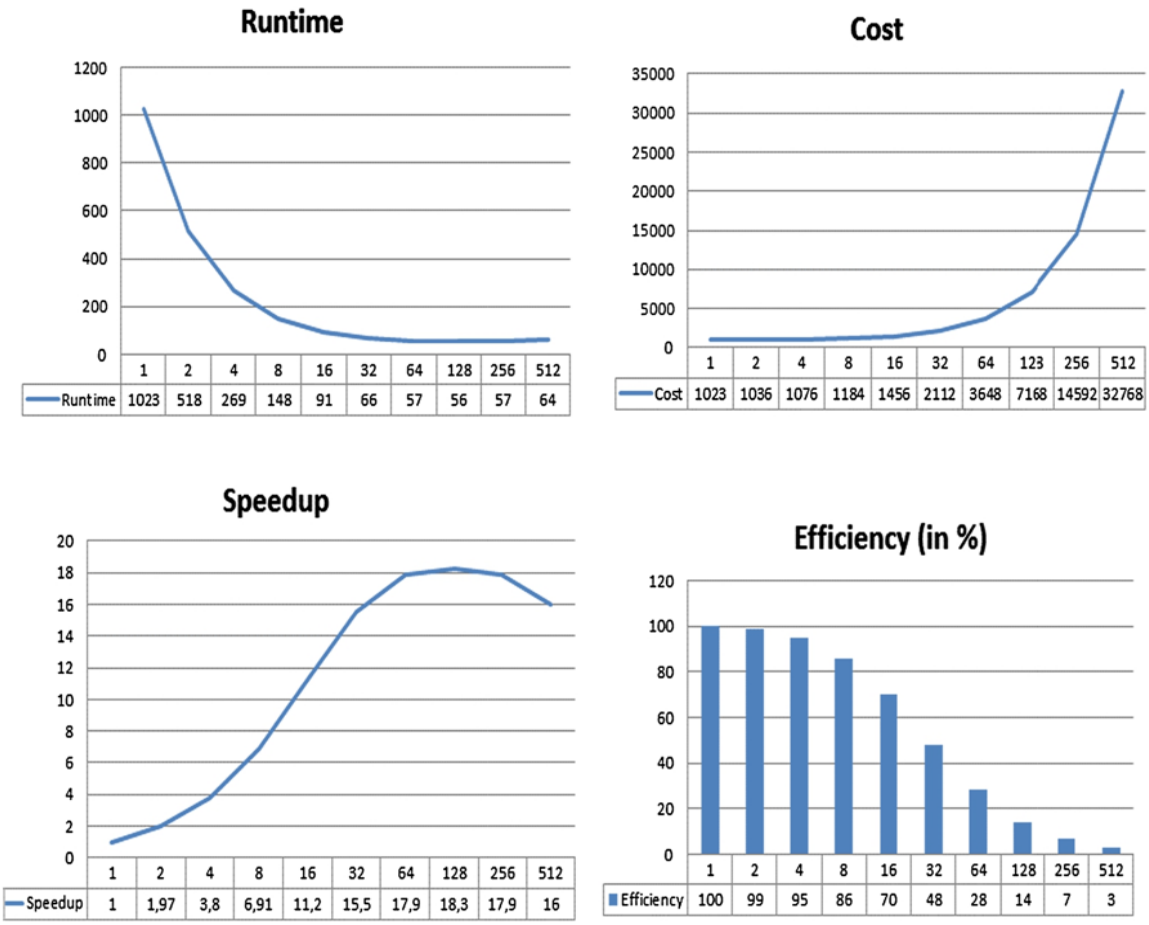


FIGURE 1.4 Strong scalability analysis: runtime, speedup, cost, and efficiency of our parallel summation algorithm for adding $n = 1024$ numbers on a varying number of PEs (ranging from 1 to 512).

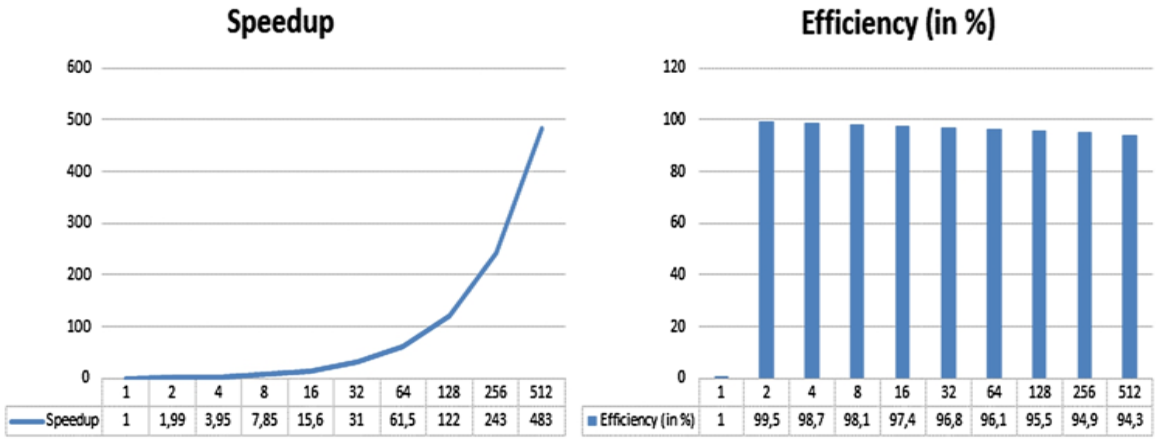
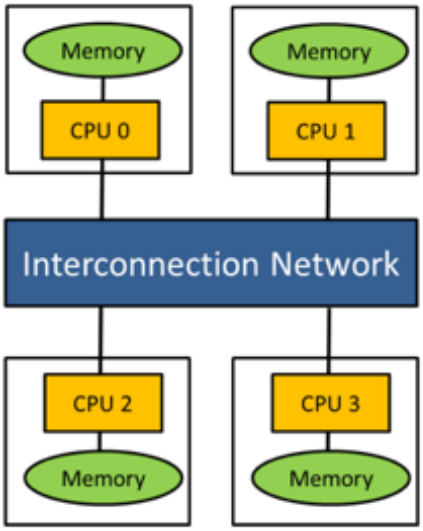
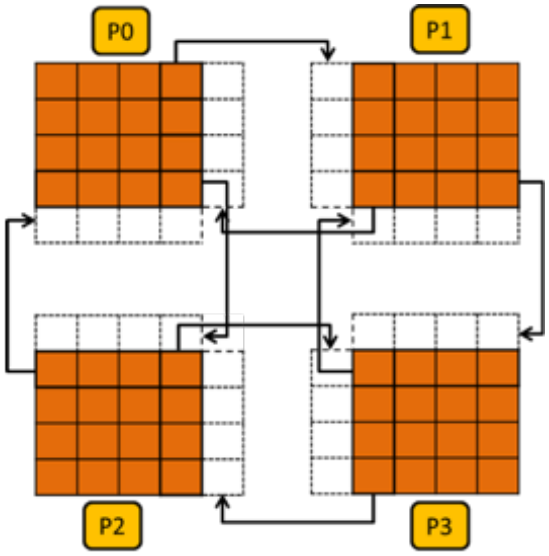


FIGURE 1.5 Weak scalability analysis: speedup and efficiency of our parallel summation algorithm for adding $n = 1024 \times p$ numbers on p PEs (p ranging from 1 to 512).

Distributed memory systems



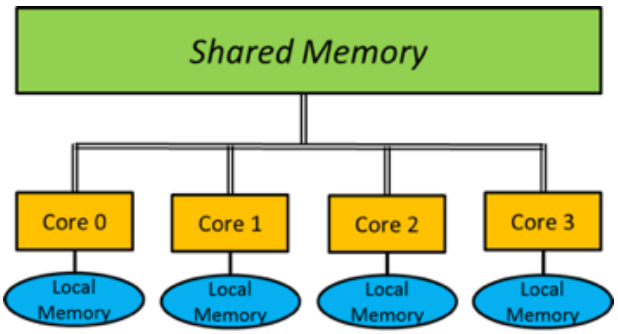


ex) partitioned 8x8 matrix for 5-point stencil code

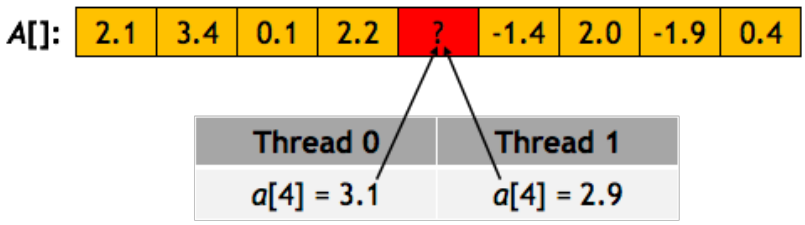
- Each node has its own, private memory. Processors communicate explicitly by sending messages across a network
    - Most popular language: MPI (e.g., MPI_Send, MPI_Recv, MPI_Bcast, MPI_Reduce)

- Example: Compute Clusters
    - Collection of commodity systems such as CPUs connected by an interconnection network (e.g., Infiniband)

Shared memory systems



- All cores can access a common memory space through a shared bus or crossbar switch
    - e.g. multi-core CPU-based workstations in which all cores share main memory
- In addition to the shared main memory each core can also contains a smaller local memory (e.g. L1-cache) in order to reduce expensive accesses to main memory (*von-Neumann bottleneck*)
    - Modern multi-core CPU systems support cache coherence
    - *ccNUMA*: cache coherent non-uniform access architectures
- Popular languages: C++11 multithreading, OpenMP, CUDA
- Parallelism created by starting threads running concurrently on the sys.
- Exchange of data implemented by threads reading from and writing to shared memory locations.
- **Race condition:** occurs when two threads access a shared variable simultaneously



    - Corresponding programming techniques:

        mutexes, *condition variables*, *atomics*

- Thread creation is more lightweight/faster compared to process creation:

    ex) CreateProcess(): 12.76 ms

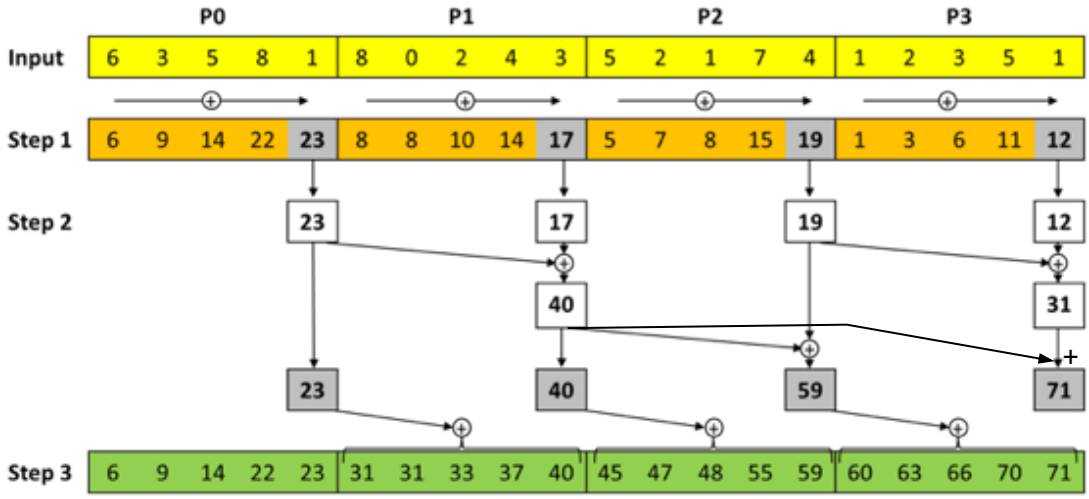
    CreateThread(): 0.038 ms

Parallel Program Design

- Partitioning:
  - Given problem needs to be decomposed into pieces; e.g. data parallelism, task parallelism, model parallelism
- Communication:
  - Chosen partitioning scheme determines the amount and types of required communication
- Synchronization:
  - In order to cooperate in an appropriate way, threads or processes may need to be synchronized
- Load Balancing:
  - Work needs to be equally divided among threads or processes in order to balance the load and minimize idle times

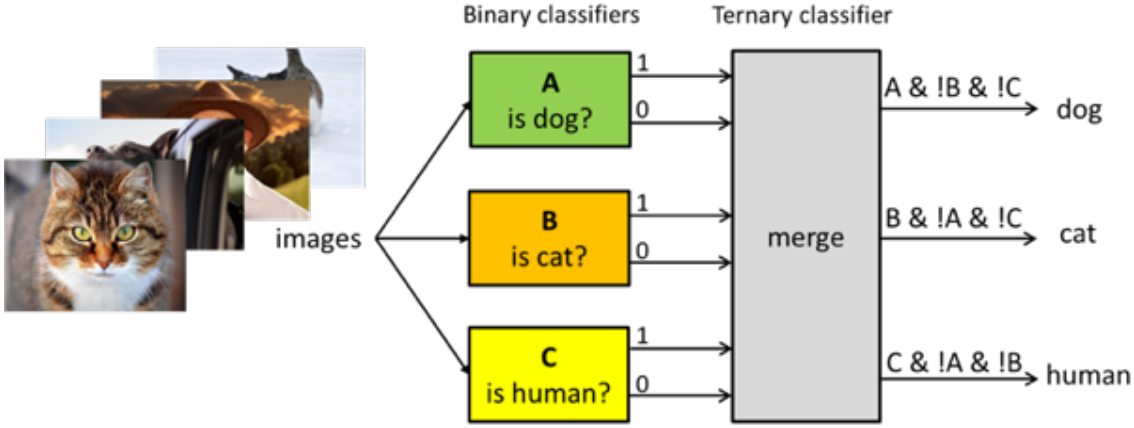
Discovering parallelism: prefix sum example

```
for (i=1; i<n; i++) A[i]=A[i]+A[i-1];
```



- Step 1: local summation within each processor
- Step 2: Prefix sum computation using only the rightmost value of each local array
- Step 3: Addition of the value computed in Step2 from the left neighbor to each local array element

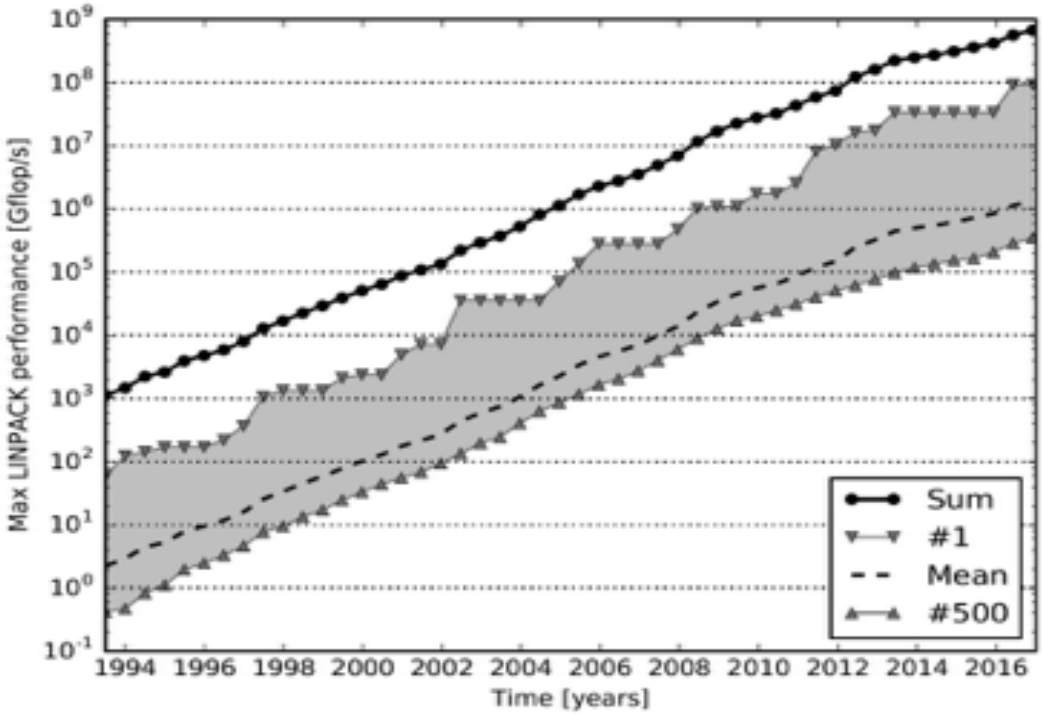
Partitioning strategies: image classifier example



- Task parallelism
  - Different binary classifier assigned to a different process (P_0, P_1, P_2)
  - Every process classifies each image using the assigned classifier.
  - Binary classification results for each image are send to P_0 and merged
  - Limited parallelism (only 3 for the 3 binary classifiers) and possible load imbalance (complexity of classifiers is different)
- Data parallelism
  - Input images could be divided into a number of batches.
  - Once a process has completed the classification of its assigned batch, a scheduler dynamically assigns a new batch
- with a large number of processors, task parallel + data parallel

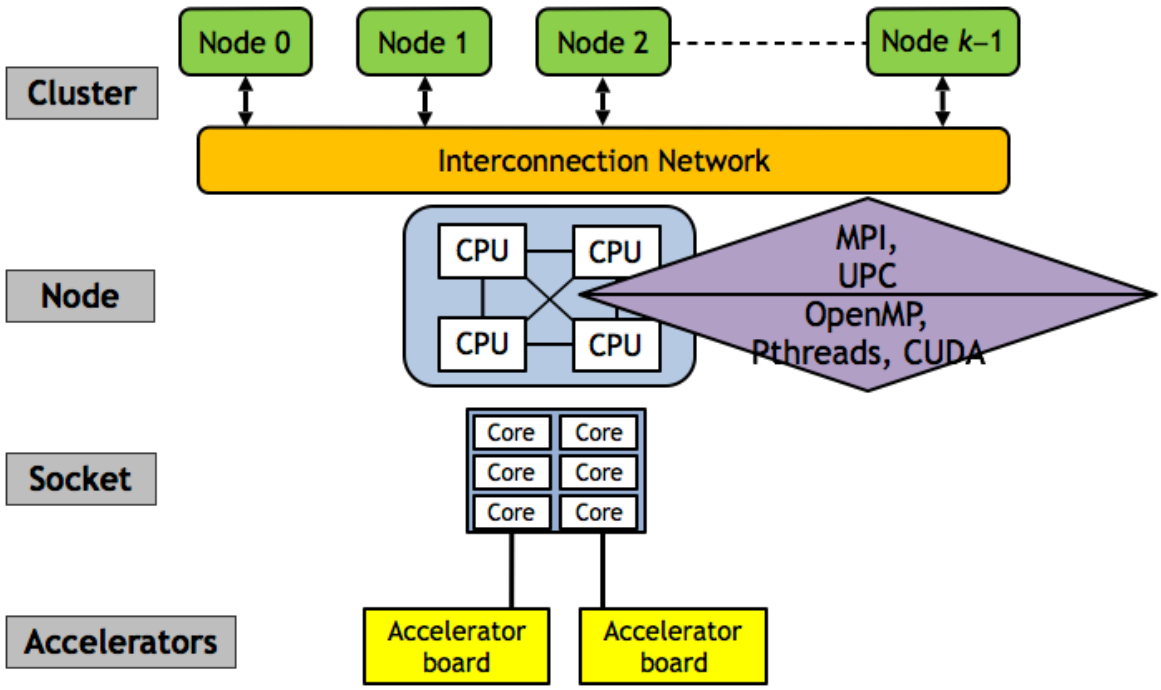
HPC trends

Top500 trends



- Top500: Supercomputers ranked according to their maximally achieved LINPACK performance in GFlop/s (top500.org)
  - Measures the performance for solving a dense system of linear equations ($A \cdot x = b$) in terms of Flop/s
  - Top-ranked system in 2017: *Sunway Taihu Light* contains over 10 million cores (hybrid system) and achieves 93 PFlop/s
- Green500: Flop/s-per-Watt of achieved LINPACK performance
  - Ranking is based on power efficiency
  - Top-ranked systems have heterogeneous nodes (having accelerators, e.g., GPU) – achieving HP with less energy
- Another well-known HPC ranking is the Graph500 project www.graph500.org

ex) Heterogeneous HPC system and associated programming languages



- Node-level parallelization: distributed memory model (e.g., MPI)
- Intra-node parallelization: within a node, shared-memory model (e.g., PThread, OpenMP) on multicore processor
- Accelerator-level parallelization: on heterogeneous system, e.g., GPU board with CUDA