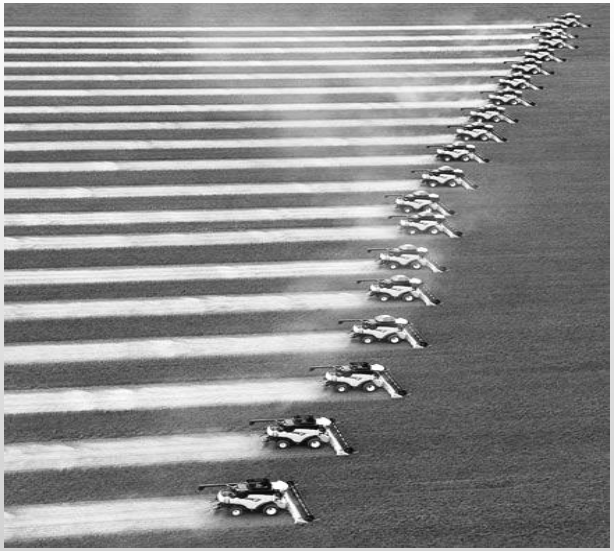
Concurrent Computing

Lecturers: Prof. Majid Mirmehdi majid@cs.bris.ac.uk
Dr. Tilo Burghardt tilo@cs.bris.ac.uk
Dr. Daniel Page page@cs.bris.ac.uk

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001



LECTURE MM5

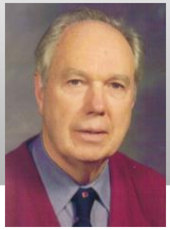
PARADIGMS OF PARALLELISM

Lecture Overview

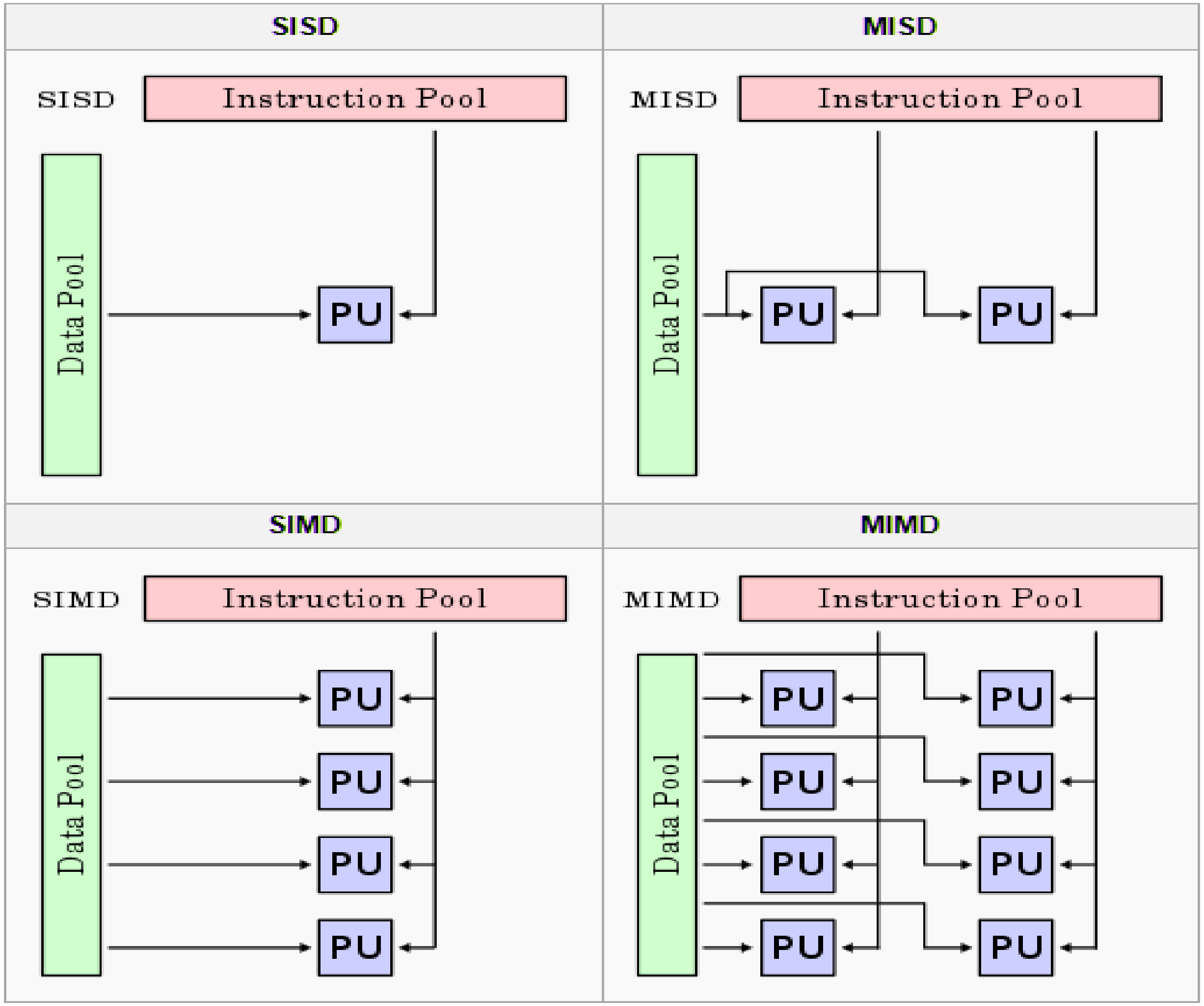
- **Paradigms of Hardware Parallelism**
  - Flynn’s Taxonomy
  - SISD, SIMD, MISD, MIMD
- **Paradigms of Logical Parallelism**
  - Independence: Bernstein’s Condition
  - Geometric Parallelisation
  - Parallelisation via Farming
  - Algorithmic Parallelisation
- **Process Networks**
- **Performance issues**

Blue Crystal, Bristol

Flynn’s Taxonomy



Michael J. Flynn



Flynn identifies **four classes** based on the number of concurrent instruction (or control) and data streams available in an architecture.

Paradigms of **Hardware Parallelism**

(What fundamental types of parallelisation exist?)

SISD

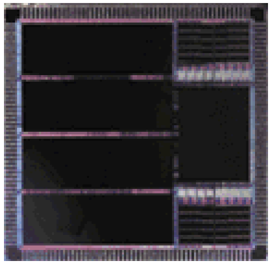
…sequential computer, no parallelism. Single control unit fetches single instruction stream from memory. Instruction manipulates single data item at a time., e.g. *1990s Pentium*

SIMD

…exploits multiple data streams against single instruction stream, e.g. *MMX technology for multimedia data types on Pentiums*

MISD

…multiple instructions operate on a single data stream; often used for fault tolerance., e.g. Space Shuttle flight controllers

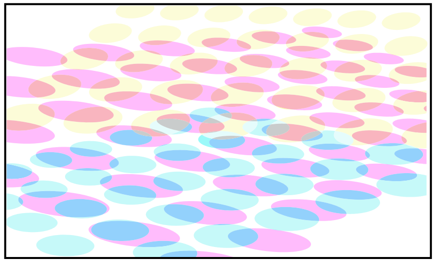


MIMD

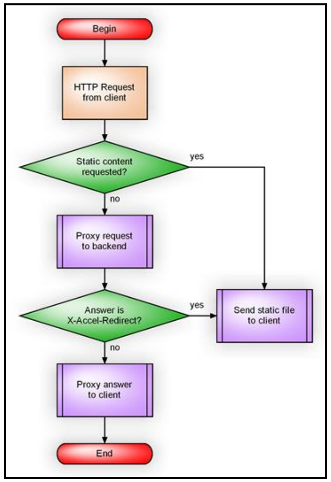
…multiple autonomous processors simultaneously execute different instructions on different data, e.g. *Intel i7, XMOS XC-1A*

Paradigms of **Logical Parallelism**

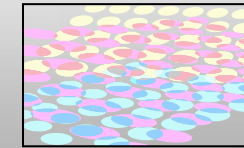
(How can it be parallelised?)



- **Geometric**: i.e. data is divided up
  - processes are quasi-independent but identical
  - each process operates on a portion of data
  - interaction with neighbours (according to geometry)
- **Farming**: i.e. data is divided up
  - single source (farmer) distributes work (data) to workers
  - work is done independently (at own pace) by worker(s)
  - workers send results to harvester
- **Algorithmic**: i.e. algorithm is parallelised
  - quasi-independent processes execute sections of algorithm
  - processes may be non-identical
  - data is passed between processes according to algorithm
- (and mixtures of the above)



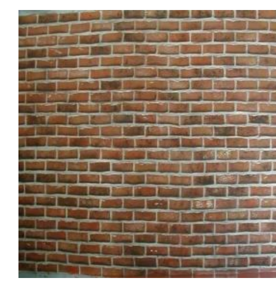
Geometric Parallelism (static data division)



- **Geometric: data is proportioned equally**

Useful when there is a natural way to decompose the data set into smaller "chunks“ and allocate them to individual processors. Replication can often serve as the enabling programming concept.

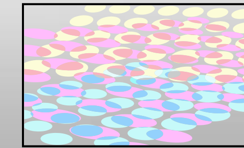
Each processor executes the same code but clearly operates on a subset of the total data.



Wall building analogy

- Distribute bricks amongst a number of bricklayers
- Each bricklayer responsible for a vertical section of wall.
- Communication required with those building neighbouring sections.

Geometric Parallelism (static data division)



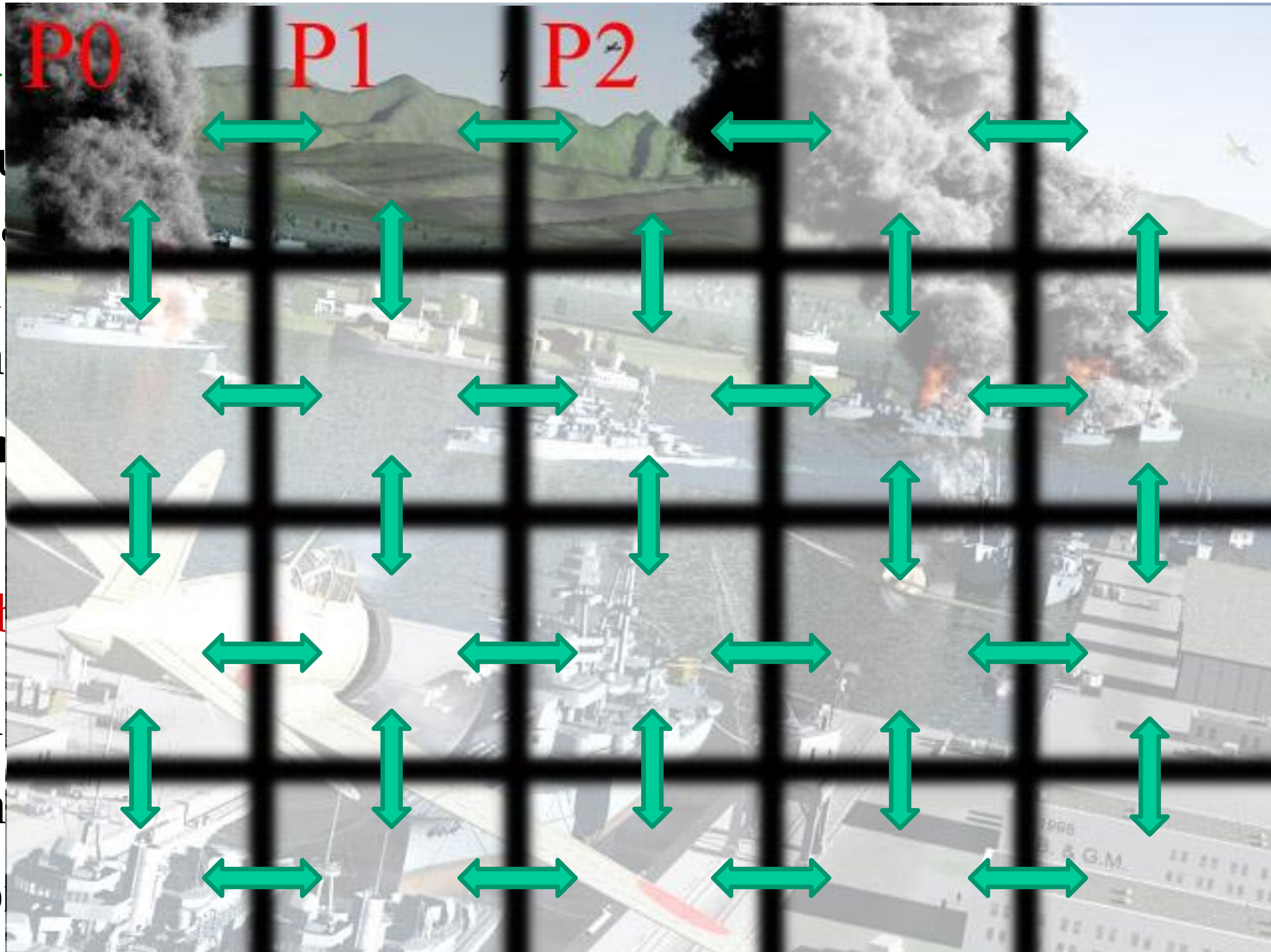
- G

Usefu
the da
them
often

**Each
on a**

Wall b

- Di
- Ea
- Co
  sections.



Case Study: Geometric Parallelism

Matrix Multiplication

matrix $\boldsymbol{A}$ of order $\boldsymbol{m} \times \boldsymbol{p}$ with elements $\boldsymbol{a_{ik}}$, and
matrix $\boldsymbol{B}$ of order $\boldsymbol{p} \times \boldsymbol{n}$ with elements $\boldsymbol{b_{ik}}$.

Product is matrix $\boldsymbol{C}$ of order $\boldsymbol{m} \times \boldsymbol{n}$ with elements $\boldsymbol{c_{ik}}$ where…

$$c_{ik} = \sum_{j=1}^{p} a_{ij} b_{jk}$$

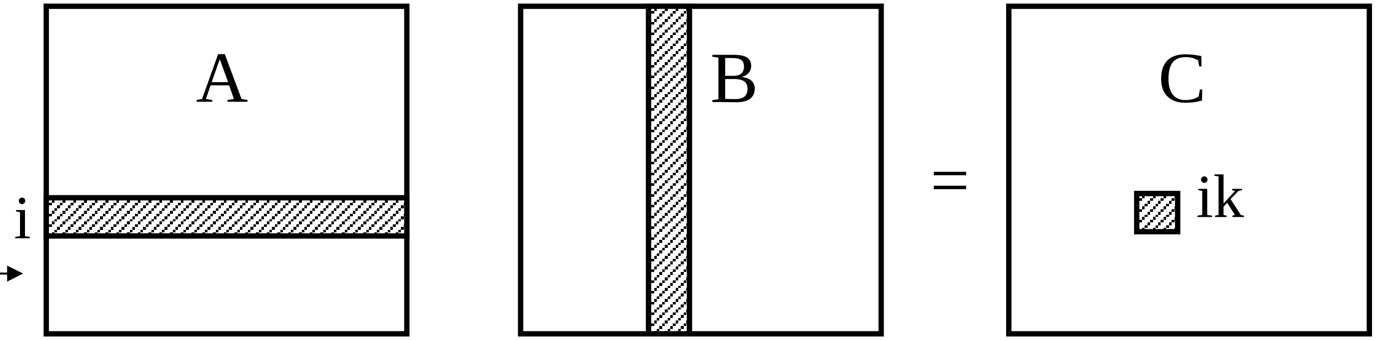
Each element $\boldsymbol{c_{ik}}$ is product of $\boldsymbol{i^{th}}$ row of $\boldsymbol{A}$ with $\boldsymbol{k^{th}}$ column of $\boldsymbol{B}$.

Matrix Multiplication (sequential)

```
// sequential solution, fragment
  ...
  for (int i = 1; i < m; i++)
    for (int k = 1; k < m; k++) {
      int tot = 0;
      for (int j = 1; j < m; j++)
        tot := tot + a[i,j]*b[j,k];
      c[i,k] = tot;
    }
  ...
```

$$c_{ik} = \sum_{j=1}^{m} a_{ij} b_{jk}$$

m × m
matrices



Matrix Multiplication: Cell Evaluation

Each cell communicates with neighbours via vertical and horizontal input and output channels.

```
// CODE FRAGMENT: cell evaluation

void multCell(chanend left, chanend top, chanend right, chanend bottom) {

  int result = 0;
  int a, b;
  for (int p=0; p<3; p++) {
    par {
      top :> a;
      left :> b;
    }
    par {
      result = result + (a * b);
      bottom <: a;
      right <: b;
    }
  }
}
```

Matrix Multiplication: Main Harness

```
// CODE FRAGMENT: Harness for setting up the multiplier cells in grid

// dimensions of M x N matrix
#define M 3  // no of rows
#define N 3  // no of columns

// main harness
int main(void) {
  chan hor[M][N+1]; // horizontal channels
  chan ver[N][M+1]; // vertical channels
  // add code here to feed grid with values
  par (int i=0;i<M;i++) {     // go through rows
    par (int j=0;j<N;j++) {   // go through columns
      multCell(hor[i][j],ver[j][i],hor[i][j+1],ver[j][i+1]);
    }
  }
  // add code here to bleed grid with output results
  return 0;
}
```

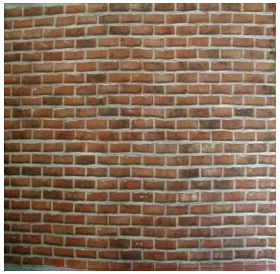
Farming (dynamic data division)

- **Farming: processors work on data subsets when ready**

Involves a farmer distributing work.

- Workers receive work from farmer,
- process the work or pass it on to next worker,
- result is returned to harvester (e.g. farmer).

Used for computation intensive calculations, e.g. ray tracing.

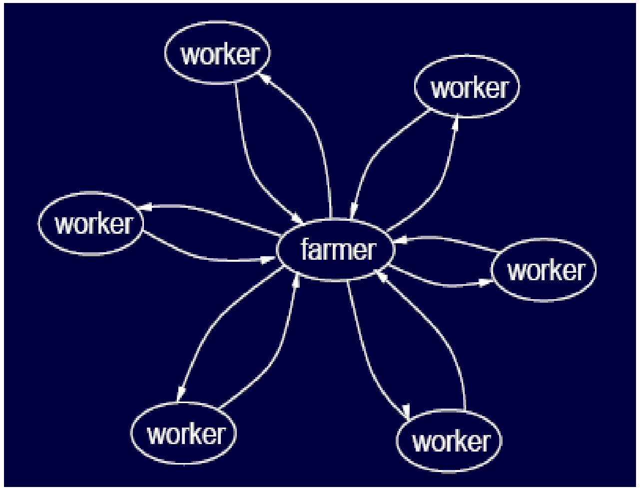


Wall building analogy

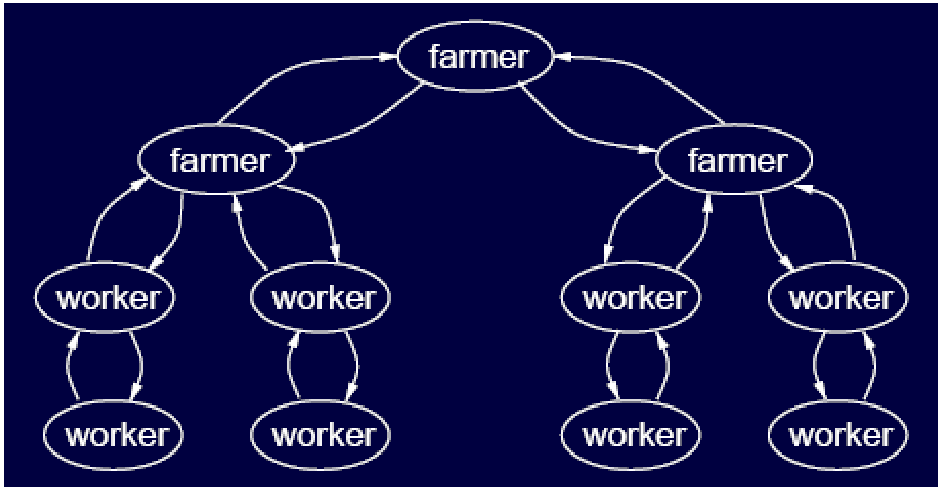
- Each bricklayer lays a brick at a time.
- Bricks and mortar are taken from a common pool.
- Communication of interim results not particularly necessary.

Farming (dynamic data division)

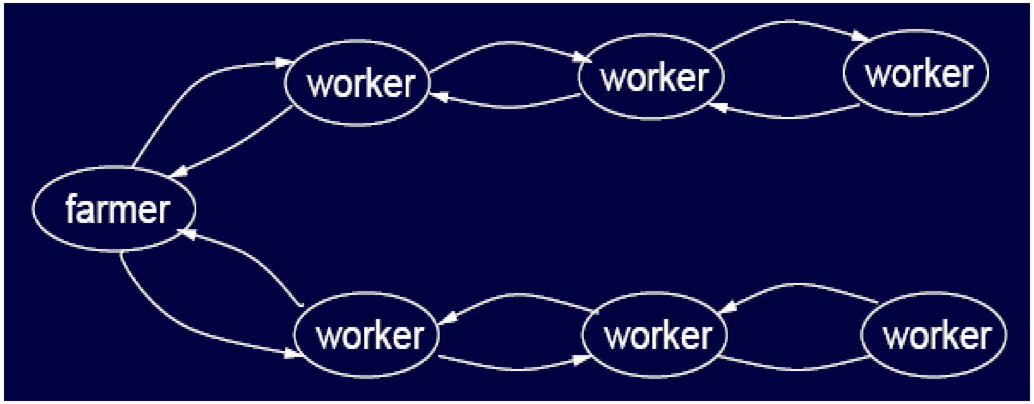
Common Topologies of Processor Farms



Simple Star



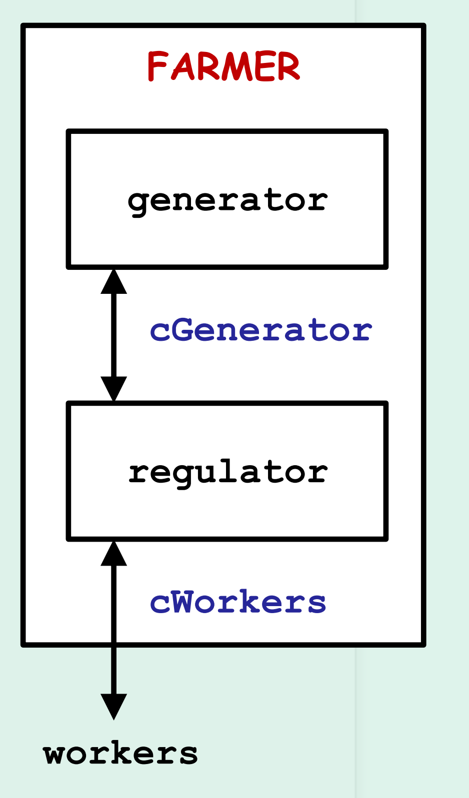
Hierarchical Tree (with co-worker pairs)



Deep Stumps (star of worker chains)

XC Regulator Fragment for Processor Farm

```
// fragment code for farm regulator
void regulator(chanend cWorkers, chanend cGenerator, int workerNum) {
  ...
  int idle = workerNum;
  while (farmerRunning) {
    if (idle > 0)
      select { //some workers free, results or data allowed
        case cWorkers :> result:
          cGenerator <: result;
          idle++;
          break;
        case cGenerator :> dataPacket:
          cWorkers <: dataPacket;
          idle--;
          break;
          ...
      }
    else
      select { //all workers busy, only results allowed
        case cWorkers :> result:
          cGenerator <: result;
          idle++;
          break;
        ...
      }
  }
  ...
}
```



The Price Tag: Communication Overheads

For example in farming…

- Adding more workers to a farm can enhance performance.

However,
… the farm can become swamped with overheads of routing data to/from workers.

- This takes processing time away from the real work! Serious performance degradation if too many workers are added.
→ Communication bottleneck!

- Finding the right balance between communication overheads and time spent processing can be very hard!

...can also occur in geometric parallelism…

Algorithm Parallelisation

- **Structure of algorithm is decomposed/parallelised**

Stream of data pass through a pipeline.

Each stage of the pipeline performs some operation.

Video processing pipeline with 4 stages

Example

1. Smoothing a raw image
2. Feature extraction from smoothed image
3. Object recognition from feature descriptions
4. Graphics display of recognized objects

Wall building analogy

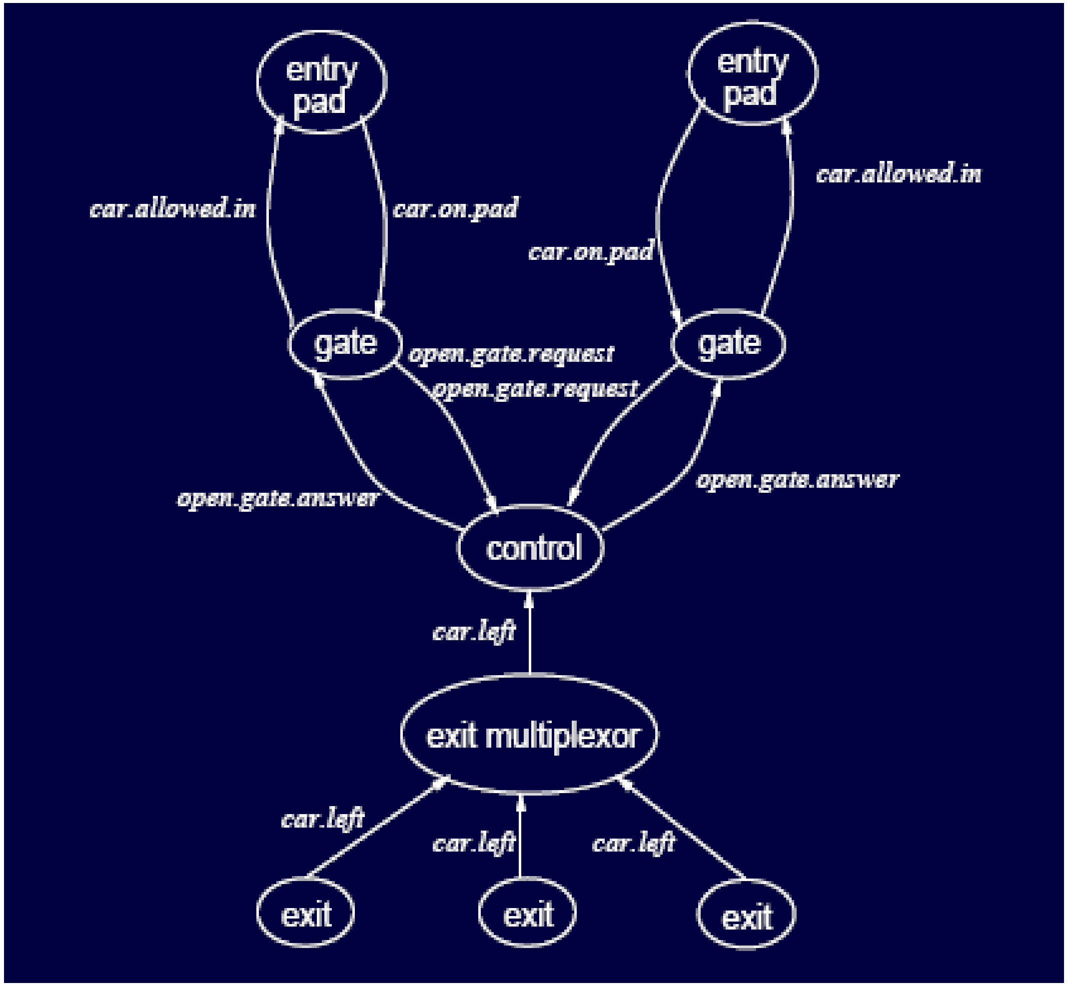
- Each bricklayer lays a row of bricks.
- Work is done simultaneously with others, but offset in time.
- Work at pace of slowest worker, some workers idle at beginning/end.

Generic Topologies: Process Networks

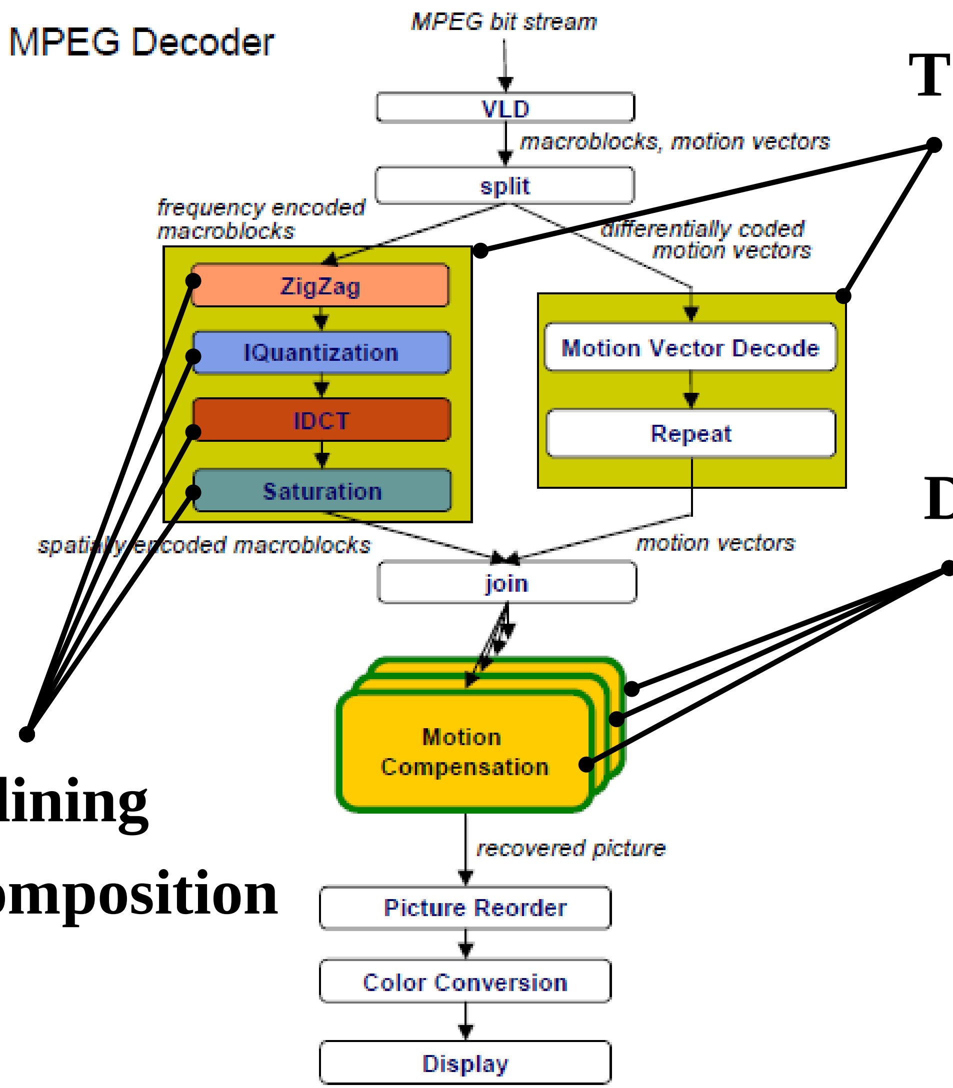
Networks provide the most flexible process topology.

Example

Car park with two entry pads (sensors) controlling a gate each, and three exit gates.



Example: Decomposition of MPEG Decoder

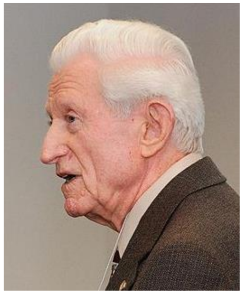


Task Decomposition:
blocks of independent computations

Data Decomposition:
same computations over different, independent data items

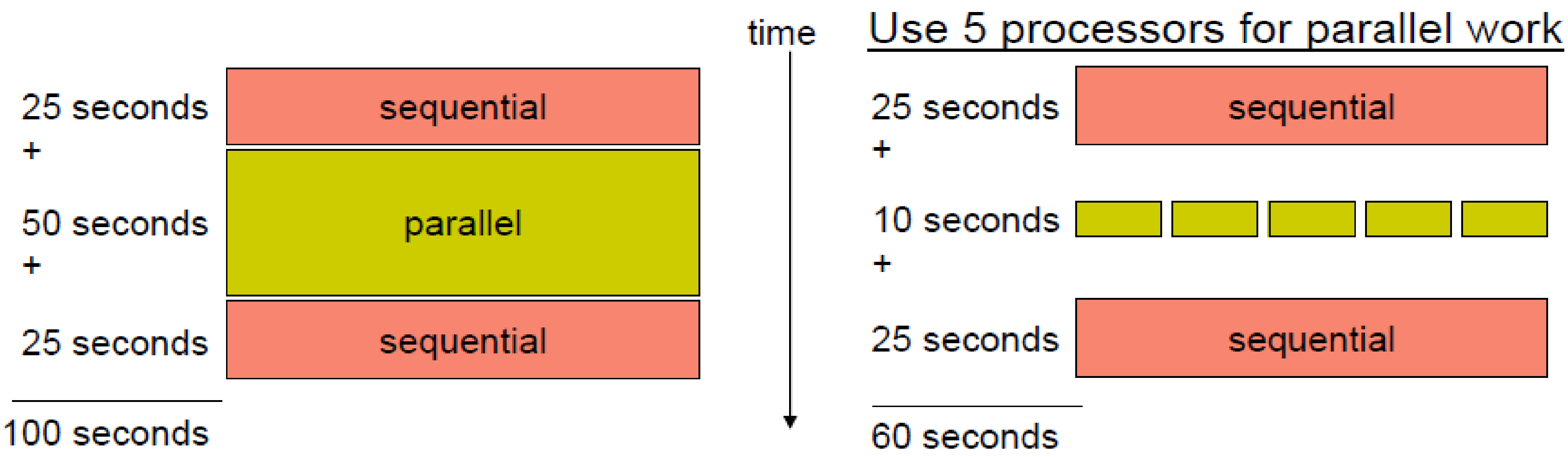
Pipelining Decomposition

Speedup Limit: Amdahl's Law

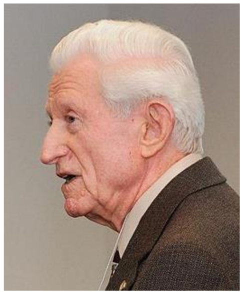


Gene Amdahl

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.



Speedup Limit: Amdahl's Law



Gene Amdahl

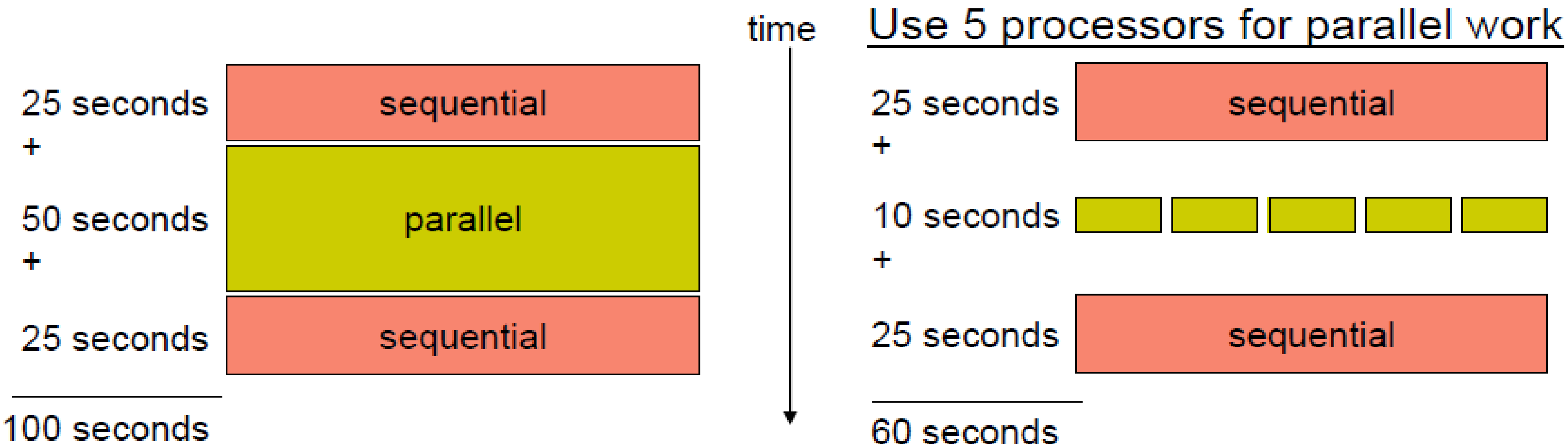
p = fraction of work that can be parallelized

n = the number of processor

$$speedup = \frac{\text{old running time}}{\text{new running time}} = \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work



Factors that Impact on Performance…

- The ***number*** and ***throughput*** of the processing nodes.
- The ***bandwidth*** between the processing nodes. Measured in Mbytes/sec (or Gbytes/sec etc.) this determines the rate at which data can be sent from one node to another.
- The ***latency*** in the node-to-node connections measures (in seconds) their delays in transmission - influenced by the bit rate, the time taken to route data, and the transmission protocol used.
- The ***Flow Control*** strategy used influences performance - nodes must be prevented from flooding each other, or the routing fabric, with traffic.

Factors that Impact on Performance…

- The amount of ***Local Memory***, e.g. in MB, often limits local computation without communication
- The ***Granularity*** of the system describes how coarsely/finely a computation and its data is partitioned into concurrently computing processes – this directly impacts on communication and load balancing
- The ***Speedup*** *factor* for a parallel computer can be defined as:

$$S(N) = \frac{\text{Execution time using one processor}}{\text{Execution time using } N \text{ processors}}$$

- The ***Efficiency*** is then defined as:

$$E(N) = \frac{S(N)}{N}$$