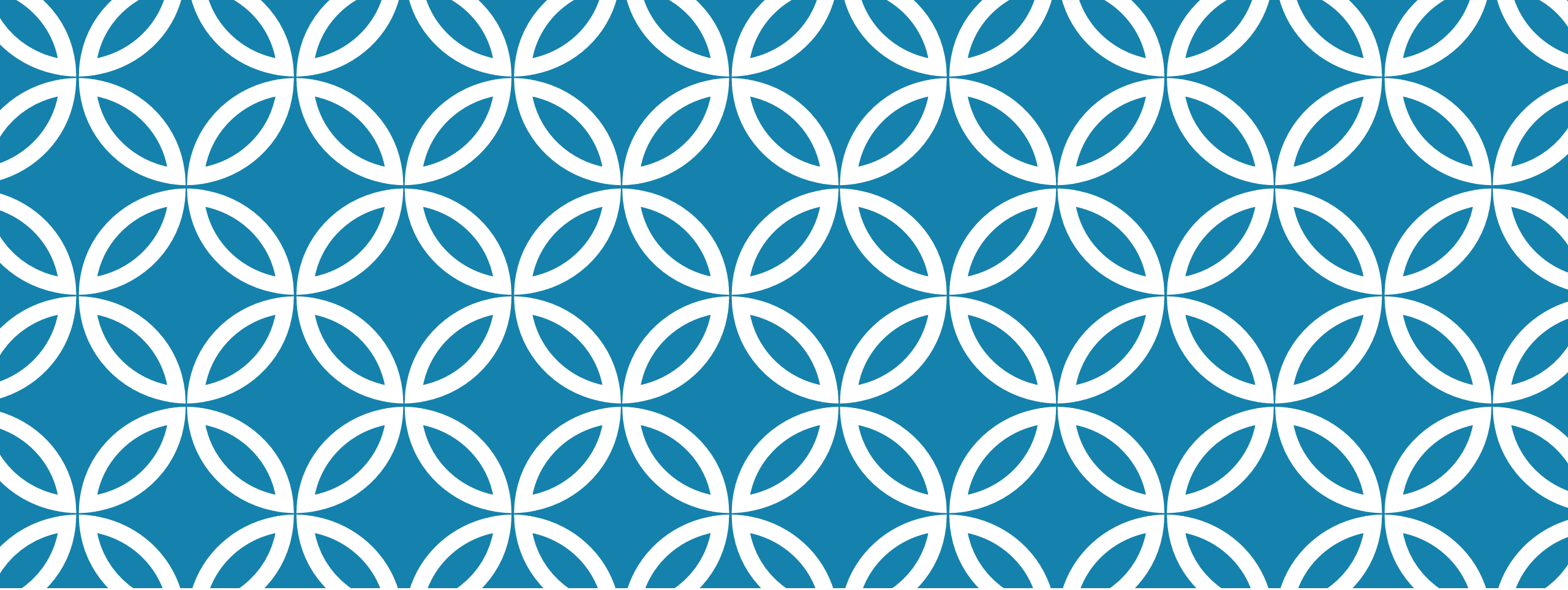


PROGRAMMING NEURAL NETWORK ON MULTI/CORE/PROCESSOR COMPUTERS

Paulus Sheetekela

CONTENT

⇒Multiprocessor computers

⇒Architecture of Multiprocessors Systems

⇒Architecture of Multicore Processors Systems

⇒Artificial Intelligence

⇒Types of Neural Networks

⇒Activation Functions

CONTENT

⇒Neural Networks Algorithms

⇒Programming Neural Networks

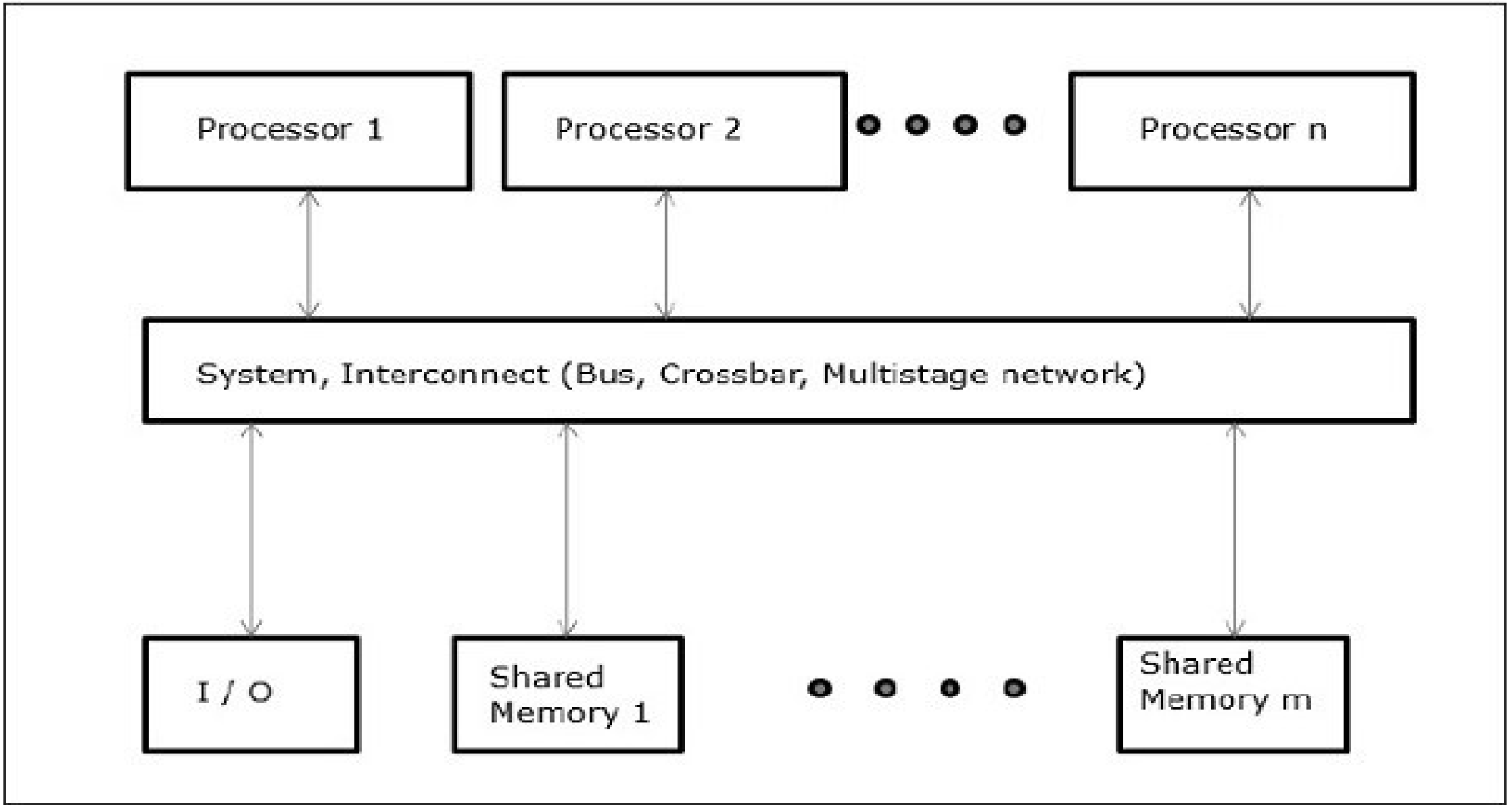
⇒Coding for Multiprocessors computers

⇒Coding for Multicore processors Computers

⇒GPU Frameworks /Libraries for NN

⇒Conclusion

ARCHITECTURE OF MULTIPROCESSOR COMPUTERS



PROGRAMMING ON MULTIPROCESSOR SYSTEMS

⇒Uniprocessor system, threads execute one after another in a time-sliced manner.

⇒Multiprocessor system, several threads execute at the same time, one on each available processor.

⇒Overall performance is improved by running different process threads on different processors.

⇒But individual program cannot take advantage of multiprocessing, unless it has multiple threads.

⇒Multiprocessing is not apparent to most users - it is handled completely by the operating system and the programs it runs.

⇒Binding processes (force them to run on a certain processor); - not required, nor recommended for ordinary use.

⇒Programmers, take advantage of multiprocessing simply using multiple threads.

⇒Kernel programmers have to deal with several issues when porting or creating code for multiprocessor systems.

PROGRAMMING ON MULTIPROCESSOR SYSTEMS

⇒**Identifying processors**
Symmetric multiprocessor (SMP) machines have one or more CPU boards, each of which can accommodate two processors.

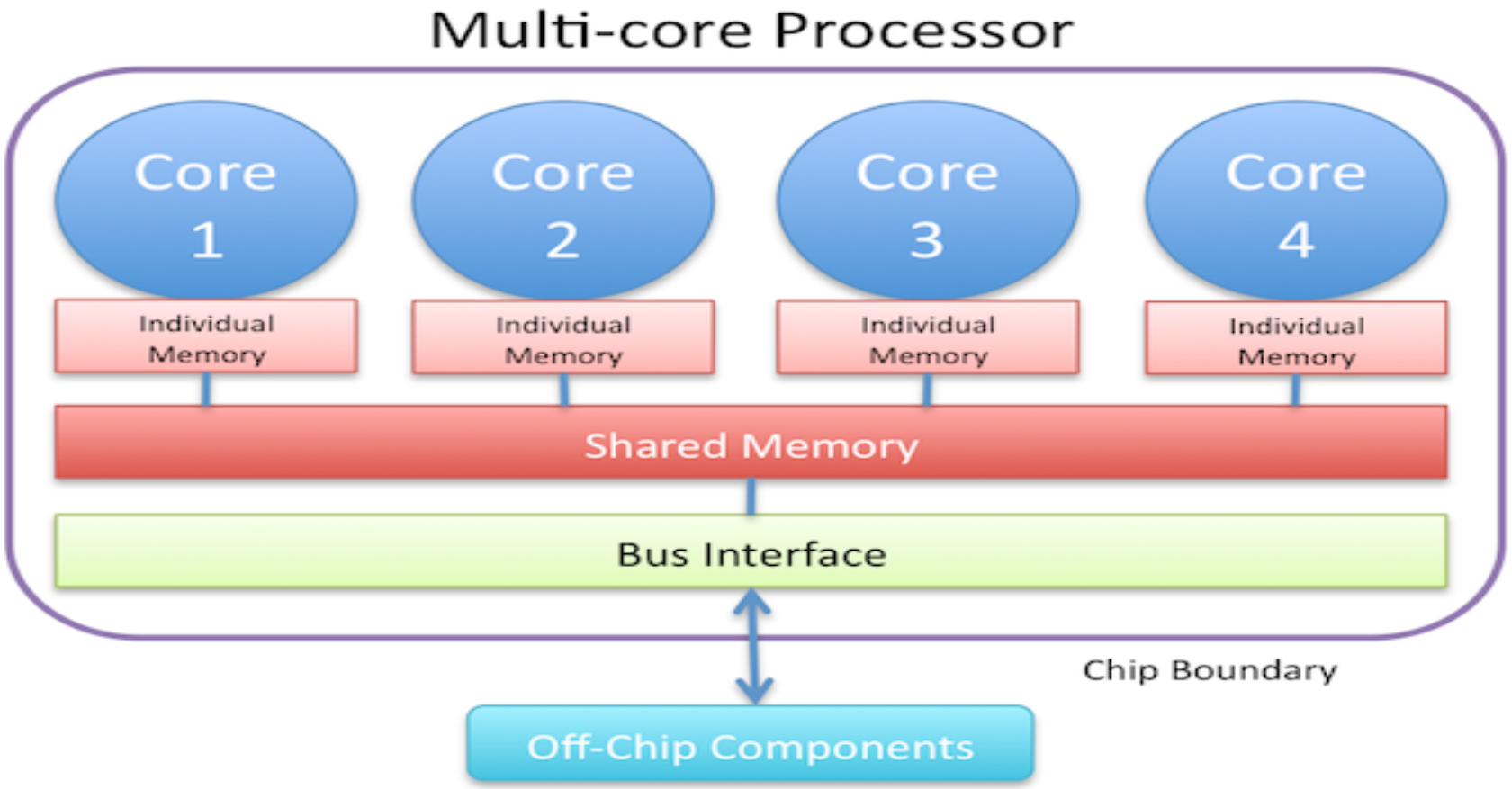
⇒**Controlling processor use**
how to control the use of processors on the multiprocessor system.

⇒**Using Dynamic Processor Deallocation**
the hardware of all systems with more than two processors can detect correctable errors, which are gathered by the firmware. This prediction is made by the firmware based-on-failure rates and threshold analysis.

⇒**Dynamic memory guarding**
AIX systems are designed to be resilient in regards to memory errors. Memory error resilience is the result of both hardware and operating system-level recoveries.

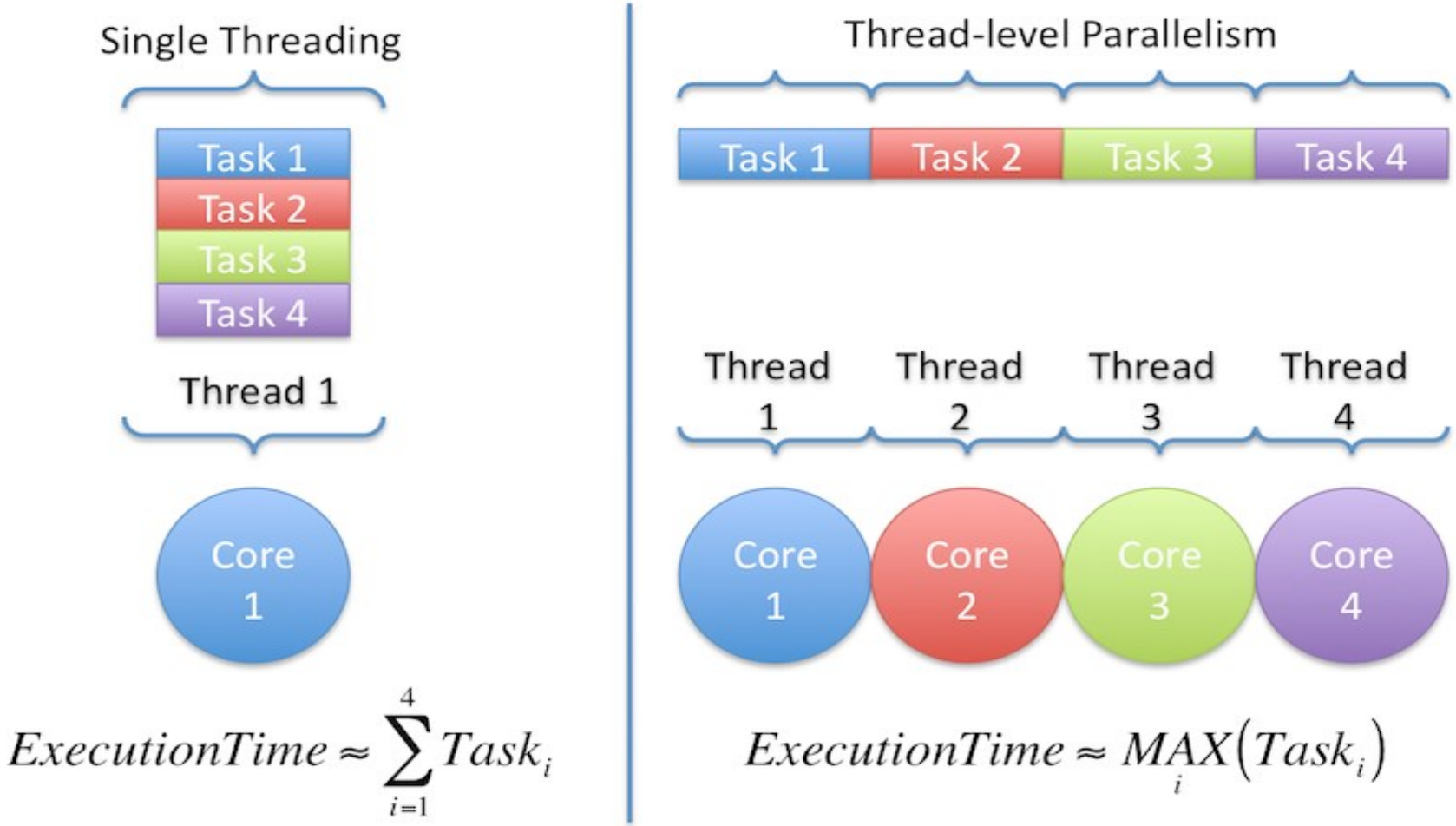
⇒**Creating locking services**
programmers may want to implement their own high-level locking services instead of using the standard locking services (mutexes) provided in the threads library.

ARCHITECTURE OF MULTICORE PROCESSOR SYSTEMS



A basic block diagram of a generic multi-core processor: Source

THREAD-LEVEL PARALLELISM



A conceptual visualization of thread-level parallelism

AMDAHL'S LAW – SPEEDUP ON PARALLELISM

$$Speedup_{parallel}(f,n) = \frac{1}{(1-f) + \left(f / n\right)}$$

$$Speedup_{enhanced}(f,S) = \frac{1}{(1-f) + \left(f / S\right)}$$

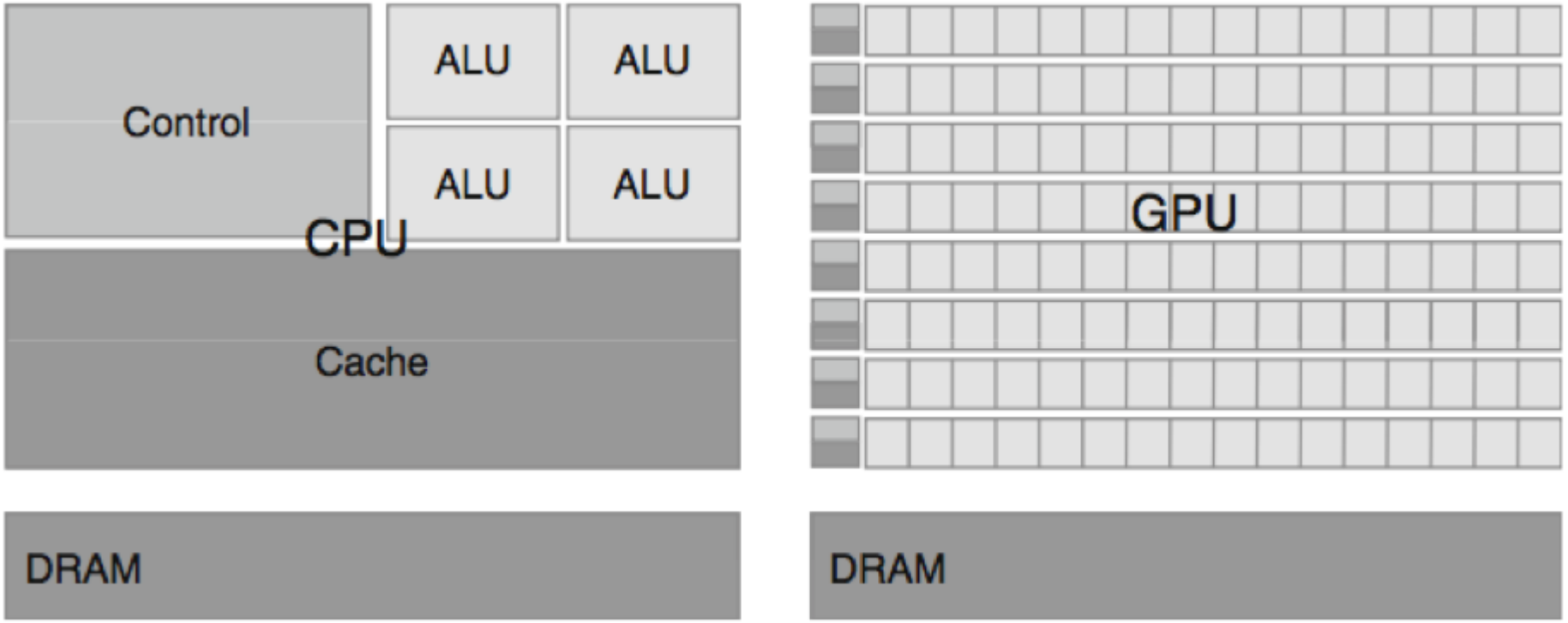
Amdahl's law - equations for speedup achieved by parallelization

MULTI-CORE/PROCESSOR SYSTEM CHALLENGES

Evenly spreading the workload among the CPUs

Eliminating false sharing and other types of memory contention between CPUs

Making sure that the data used by each CPU are located in a memory near that CPU's node



TWO WAYS TO OBTAIN THE USE OF MULTIPLE CPUS

1. Write your source code using explicit parallelism
   ⇒ showing in the source code which parts of the program are to execute asynchronously and how the parts are to coordinate with each other.

```
for (int i = 0 ; i < 1000000; i ++)
do work ( i ) ;
```

1. Apply a precompiler to it to find the parallelism that is implicit in the code,
   ⇒ insert the source directives for parallel execution for you

```
#pragma omp parallel for
for ( int i = 0 ; i < 1000000; i ++)
do work ( i ) ;
```

ARTIFICIAL INTELLIGENCE

Function of neuron can be expressed as follows:

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right) \quad n \in \mathbb{N}, w_i \in \mathbb{R}$$

where y is the output of neuron, f is the transfer function, x_i is one particular input of neuron and w_i is particular weight associated to specific input.

TYPES OF NEURAL NETWORKS

Feedforward Neural Network – Artificial Neuron

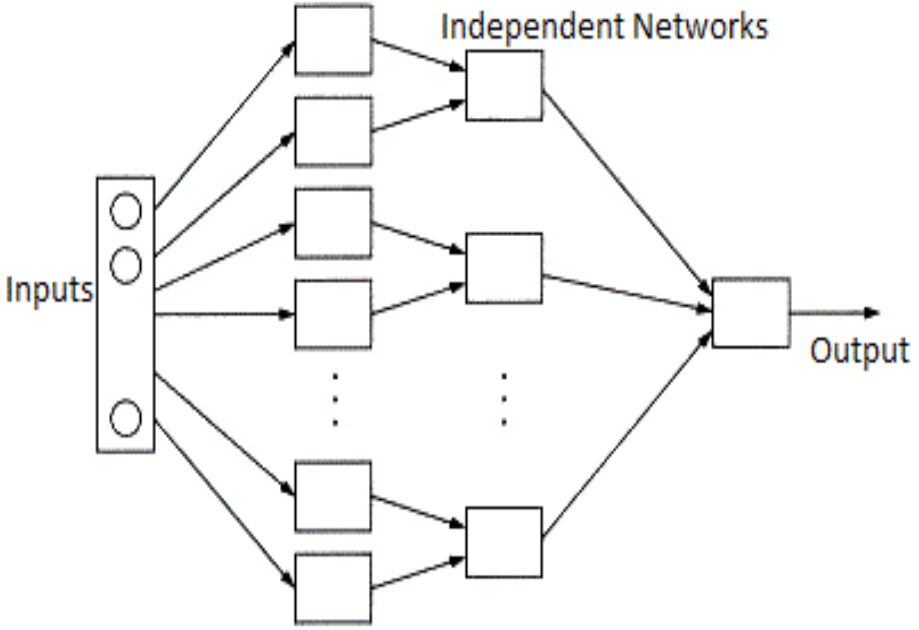
Radial basis function Neural Network

Kohonen Self Organizing Neural Network

Recurrent Neural Network(RNN) – Long Short Term Memory

Convolutional Neural Network

Modular Neural Network



ACTIVATION FUNCTIONS

The commonly used activation functions are

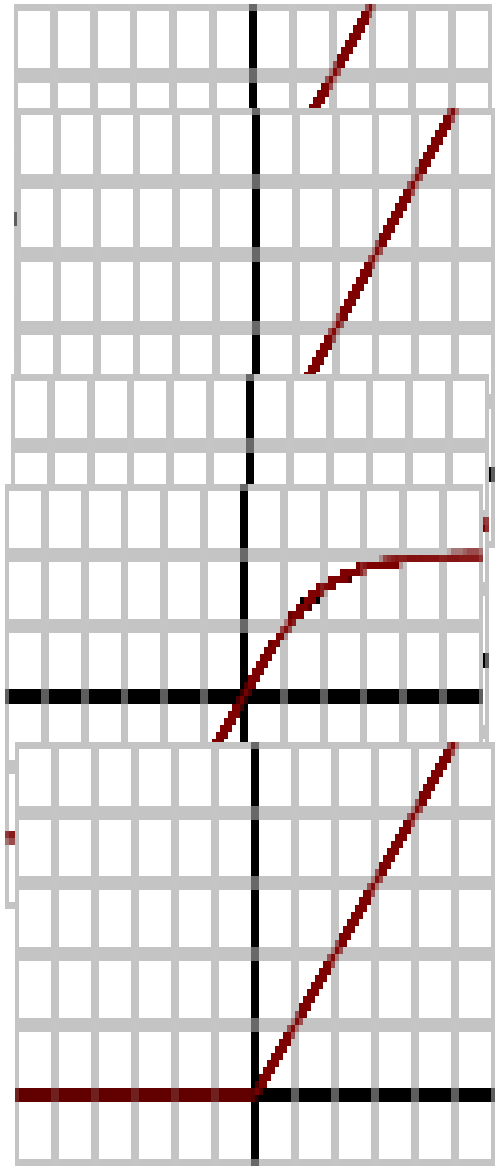
⇒linear, $f(x) = ax$

⇒step, $f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geq 0 \end{cases}$

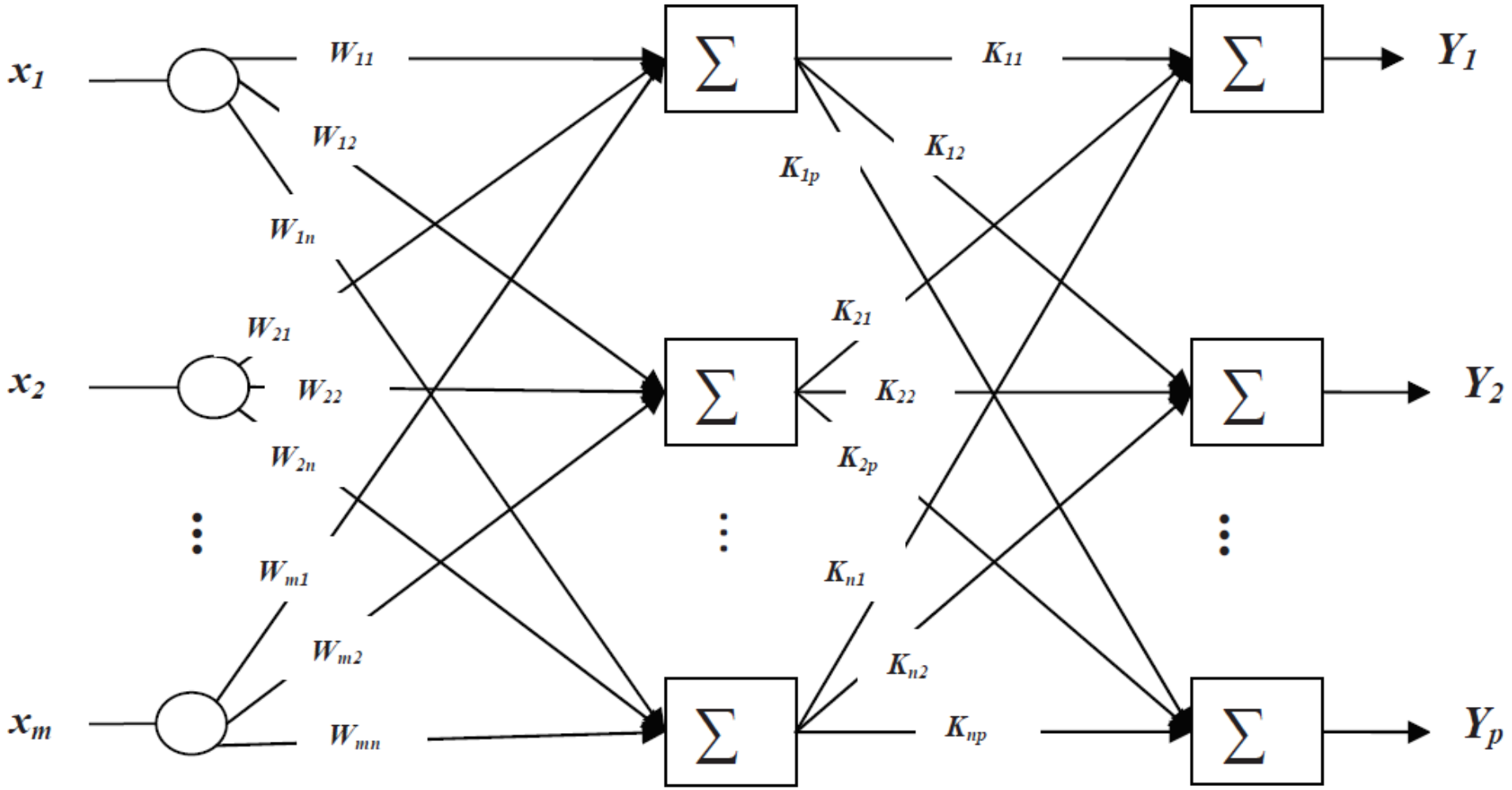
⇒Sigmoid or logistic, $f(x) = \frac{1}{1+e^{-x}}$

⇒tanh, $f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$

⇒rectified linear unit (ReLu), $f(x) = \begin{cases} 0\ for\ x < 0 \\ x\ for\ x \geq 0 \end{cases}$.

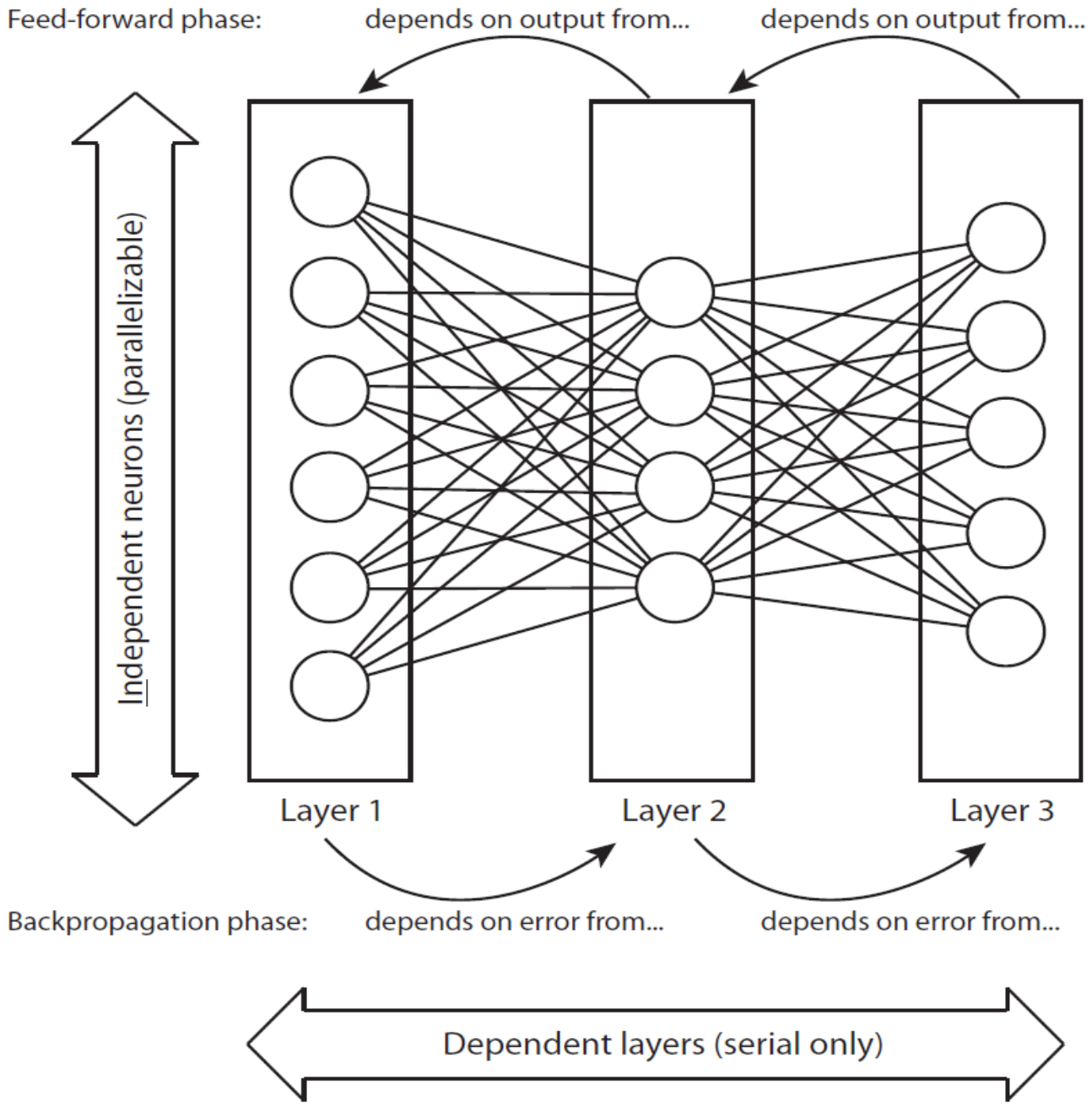


NEURAL NETWORKS ALGORITHMS (NNA)



PARALLELIZATION OF NNA

Parallelization issues with conventional, layered neural networks.



PROGRAMMING NEURAL NETWORKS

Pseudocode example of programming multilayer perceptron

```
int N;
int M;
int K;
double w;
float
for i=0; i<N-1; i++; //N = number of layers in a network;
    for j=0;j<M-1; j++; // M = number of neuron in a layer [i];
      for k=0;k<K-1; k++; //K = number of inputs in a neuron [j];
      sum += weight_of_neuron_j[k]*inputs_neuron_j[k];
      next k;
      Output_Neuron_j=Activation_Function(sum);
next j;
next i;
```

LEARNING METHODS

The neural networks learn or trained in the following three ways:

⇒Supervised learning methods - the correct to each input signal, and the weights are adjusted so as to minimize the error.

⇒Unsupervised or self-learning - distribute samples in some categories, due to the disclosure of the internal structure and nature of the input data

⇒Mixed learning – mixing two first ways of training.

⇒There is a the number of learning algorithms oriented toward solution of different tasks. Among them is the algorithm of back propagation of the error [8], which is one of the most successful algorithms used in learning.

LEARNING METHODS

batch gradient descent (BGD)	stochastic gradient descent (SGD)	mini-batch gradient descent (mini-BGD)

for i in range (nb_epoches):
params_grad = evaluate_gradient(loss_function, data, params)
params = params - learning_rate * params_grad
	for i in range(np_epochs):
	np.random.shuffle(data)
	for example in data:
	params_grad = evaluate_gradient(loss_function, example, params)
	params = params - learning_rate * params_grad
		for i in range(np_epochs):
		np.random.shuffle(data)
		for batch in get_batches(data, batch_size=50):
		params_grad = evaluate_gradient(loss_function, batch, params)
		params = params - learning_rate * params_grad

CODING FOR MULTIPROCESSORS COMPUTERS

```
1 typedef struct {
2         double* data ; /*mat r ix data */
3         double** av ; /*row ac c e s s v e c t o r */
4         double* dataLim ; /* conta ins f i r s t addr e s s a f t e r data ar ray */
5         int rws ; /*number of rows */
6         int cls ; /*number of columns */
7         long elements ; /*number of elements in matrix */
8 } t2dMatrix ;
9
```

Single thread matrix multiplication on CPU: continue next slide

CODING FOR MULTIPROCESSORS COMPUTERS

```
10 void cpuMatrixMulKernel (void) {
11      double* pA;
12      double* pB;
13      double* pC = matrix_c.data ;
14      double** row = &matrix_a.av [0] ; /* row p o int e r */
15      double** rwl = &matrix_a.av [1] ; /* row limit */
16      double* col = matrix_b.av[0] ; /* column pointer */
17      double* cll = matrix_b.av [1] ; /* column limit */
18
19      while(pC < matrix_c.dataLim) {
```

Single thread matrix multiplication on CPU: continue next slide

CODING FOR MULTIPROCESSORS COMPUTERS

```
20          while ( col<cll) {
21                  *pC = gBeta*(pC) ; /*gBeta →beta scalar */
22                  pA = *row ;
23                  pB = col ;
24                  while (pA < *rwl ) {
25                          *pC+= ( gAlpha *(*pA)*(*pB)) ; /*gAlpha →alpha scalar */
26                          pA++;
27                          pB += matrix_b.cls ;
28                  }
```

Single thread matrix multiplication on CPU: continue next slide

CODING FOR MULTIPROCESSORS COMPUTERS

```
29                      col++;
30                      pC++;
31                  }
32                  col = matrix_b.av[0] ;
33                  row++;
34                  rwl++;
35          }
36 }
```

Single thread matrix multiplication on CPU: End!

CODING FOR MULTIPROCESSORS COMPUTERS

```
1 /*thread body of multithread matrix multiplication*/
2 void* cpuThreadMatrixMulKernel (void  *arg) {
3           unsigned rowIdx ;
4           unsigned colIdx ;
5           /* initial index of thread ( element in matrix C) */
6           long long idx = (long long)arg ;
7           double* row ; /*row pointer */
8           double* rwl ; /*row limit */
9           double* col ; /*column pointer */
10
11          while (idx< matrix_c.elements ) {
12                      /* determine row for element in matrix C */
13                      rowIdx = idx/matrix_b.rws ;
14                      /* determine column for element in matrix C */
15                      colIdx = idx % matrix_a . cls ;
16
```

CODING FOR MULTIPROCESSORS COMPUTERS

```
17                row = matrix_a.av[rowIdx ] ;
18                rwl = matrix_a.av[rowIdx+1] ;
19                c o l = &matrix_b.av[0] [col] ;
20                matrix_c.data [idx] =gBeta*matrix_c .data[idx] ;
21
22                while ( row<rwl ) {
23                        matrix_c . data[idx] += gAlpha *((*row) *( *col)) ;
24                        row++;
25                        col+= matrix_b . c l s ;
26                }
27
28        /*gThreads → number of threads */
29        idx += gThreads ; /*new index of element in matrix C*/
30  }
31  pthread_exi t (NULL) ;
32 }
```

Multithread matrix multiplication on CPU:end!

CODING FOR MULTICORE PROCESSOR COMPUTERS

NVIDIA GPU CUDA

```
void gpuMatrixMulKernel( double* matrix_a, double* matrix_b, double* matrix_c,
  const int aRws, const int aClsbRws, const int bCls,const double alpha, const double beta ) {
                    /* calculate global index of element in matrix C */
                    int gid = blockIdx.x* blockDim.x + threadIdx.x ;
                    int row = ( gid/aRws )*aRws ; /* determine row for element in C */
                     int col = (gid%bCls) ; /* column row for element in C */
                    if(gid<(aRws*bCls)) {
                              matrix_c[gid] = beta*matrix_c[gid] ;
                              for ( int i= 0 ; i<aClsbRws ; i++){
                                        matrix_c[gid]+=alpha*matrix_a[row] * matrix_b [ c o l ] ;
                                        row++;
                                        col+=bCls ;
                               }
                    }
          }
```

CODING FOR PARALLEL COMPUTERS USING DIRECTIVES

POSIX

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NTHREADS 5
void *myFun(void *x) {
        int tid;
        tid = *((int *) x);
        printf("Hi from thread %d!\n", tid);
        return NULL;
}
int main(int argc, char *argv[]) {
        pthread_t threads[NTHREADS];
        int thread_args[NTHREADS];
        int rc, i; /* spawn the threads */
        for (i=0; i<NTHREADS; ++i) {
                thread_args[i] = i;
                printf("spawning thread %d\n", i);
                rc = pthread_create(&threads[i], NULL, myFun, (void *) &thread_args[i]);
        } /* wait for threads to finish */
        for (i=0; i<NTHREADS; ++i) {
                rc = pthread_join(threads[i], NULL);
        }
return 1;
}
```

CODING FOR PARALLEL COMPUTERS USING DIRECTIVES

OpenMP directives for loops

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char **argv)
{ int i, thread_id, nloops;
#pragma omp parallel private(thread_id, nloops)
      { nloops = 0;
      #pragma omp for
      for (i=0; i<1000; ++i)
            { ++nloops; }
      thread_id = omp_get_thread_num();
      printf("Thread %d performed %d iterations of the loop.\n",
      thread_id, nloops );
      }
return 0;
}
```

CODING FOR PARALLEL COMPUTERS USING DIRECTIVES

MPI

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
      int myrank, nprocs;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
      printf("I am node %d of %d\n", myrank, nprocs);
      MPI_Finalize();
      return 0;
}
```

CUDA C++ CODE

```
__global__ void kFit(const float* X, const int X_w, const int X_h, const float* y, const int y_w, float* l1, const int l1_w, float* l_1_d, float* pred,
float* pred_d, float* W0, float* W1, float* buffer) {
        for (unsigned i = 0; i < 50; ++i) {
    dSigmoid(dDot(X, W0, l1, X_h, X_w, l1_w), l1, X_h, l1_w); //Finding the values of the hidden layer

    dSigmoid(dDot(l1, W1, pred, X_h, l1_w, y_w), pred, X_h, y_w); //Finding the matrix with predictions pred

    dMartixByMatrixElementwise(dMartixSubstractMatrix(y, pred, pred_d, X_h, y_w), dSigmoid_d(pred, buffer, X_h, y_w), pred_d, X_h, y_w );
//Determine the vector of prediction errors pred_d

    dMartixByMatrixElementwise(dDot_m1_m2T(pred_d, W1, l_1_d, X_h, y_w, l1_w), dSigmoid_d(l1, buffer, X_h, l1_w), l_1_d, X_h, l1_w);
//Back propagate the prediction errors to l_1_d

    dDot_m1T_m2( l1, pred_d, W1, X_h, l1_w, y_w ); //Update weights W1 with the result of matrix multiplication of transposed l1 and pred_d
    dDot_m1T_m2( X, l_1_d, W0, X_h, X_w, l1_w );  //Update weights W0 with the result of matrix multiplication of transposed X and l_1_d
  }
}
```

A SUBROUTINE FOR MATRIX MULTIPLICATION

```
__global__ void kDot(const float *m1, const float *m2, float *output, const int m1_rows , const int m1_columns, const int m2_columns ){
        /*  Computes the product of two matrices: m1 x m2. Inputs:  m1: array, left matrix of size m1_rows x m1_columns,    m2: array, right matrix of size m1_columns x m2_columns (the number of rows in the right matrix,  must be equal to the number of the columns in the left one), output: array, the results of the computation are to be stored here: m1 * m2, product of two arrays m1 and m2, a matrix of size m1_rows x m2_columns,    m1_rows: int, number of rows in the left matrix m1,  m1_columns: int, number of columns in the left matrix m1, m2_columns: int, number of columns in the right matrix m2 */
    const int id = blockIdx.x * blockDim.x + threadIdx.x;
    const int r = (int)id / m2_columns;
    const int c = id % m2_columns;
    float t_output = 0.f;
    for( int k = 0; k < m1_columns; ++k ) {
        t_output += m1[ r * m1_columns + k ] * m2[ k * m2_columns + c ];
    }
    output[ id ] = t_output;
}
__device__ float* dDot(const float *m1, const float *m2, float *output, const int m1_rows , const int m1_columns, const int m2_columns )
{
        kDot <<< m1_rows, m2_columns >>> (m1, m2, output, m1_rows , m1_columns, m2_columns );
        cudaDeviceSynchronize();
        return output;
}
```

SUBROUTINE FOR THE SIGMOID FUNCTION

```
__global__ void kSigmoid(float const *input, float *output) {
    /*  Computes the value of the sigmoid function f(x) = 1/(1 + e^-x).
     Inputs:
     input: array
     output: array, the results of the computation are to be stored here*/

    const int id = blockIdx.x * blockDim.x + threadIdx.x;

    output[id] = 1.0 / (1.0 + std::exp(-input[id]));
}

__device__ void dSigmoid(float const *input, float *output, const int height, const int width){
            kSigmoid <<< height, width >>> (input, output);
            cudaDeviceSynchronize();
}
```

GPU FRAMEWORKS /LIBRARIES

⇒TENSOR FLOW – open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

⇒CUDA-CONVENT – C++/CUDA implementation of convolutional (or more generally, feed-forward) neural networks.

⇒THEANO – a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. It can use GPUs and perform efficient symbolic differentiation.

⇒TORCH – scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It is easy to use and efficient, thanks to an easy and fast scripting language, LuaJIT, and an underlying C/CUDA implementation.

⇒DECAF – for deep image classification

⇒CAFFE – a deep learning framework made with expression, speed, and modularity in mind.

⇒cuDNN – a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

⇒fbfft

FOUR MAJOR PERFORMANCE STRATEGIES IN GPGPU

⇒Maximizing parallel execution

⇒Optimizing memory usage

⇒Optimizing thread execution

⇒Optimizing instruction usage

CONCLUSION/DISCUSSIONS

RESOURCES

⇒IBM AIX V6.1 documentation: https://www.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.base/kc_welcome_61.htm

⇒The Art of Multiprocessor Programming, Revised Reprint 1st Edition: by Maurice Herlihy and Nir Shavit, Morgan Kaufmann Publishers is an imprint of Elsevier, 2012, eBook ISBN: 9780123977953, Paperback ISBN: 9780123973375

⇒The **OpenCL** Specification, *Version: 1.1, Document Revision: 44,* Khronos OpenCL Working Group, *Editor*: Aaftab Munshi

⇒**Caffe:** http://caffe.berkeleyvision.org/

⇒**cuDNN Library SDK:** https://developer.nvidia.com/cudnn

⇒https://medium.com/@datamonsters/artificial-neural-networks-for-natural-language-processing-part-1-64ca9ebfa3b2

⇒GPGPU Approach on scaling and navigation when creating, modelling visualization of 3d objects in CAD/E/M systems, Paulus Sheetekela

⇒Neural network 3D reconstruction from Point clouds models for CAD systems using, Paulus Sheetekela