Parallel Computing - Others

OpenMP, MPI, R, and GPU

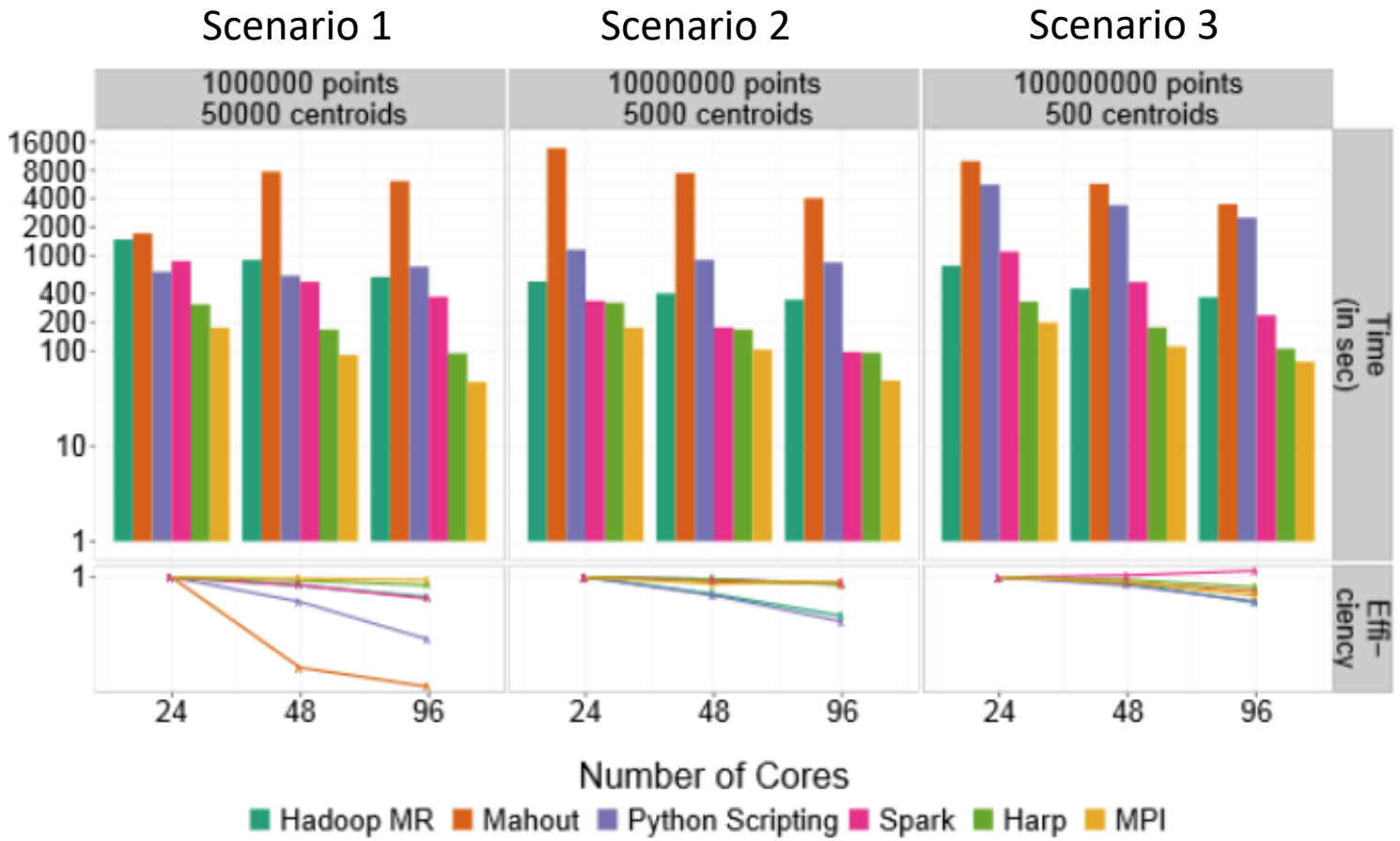
Automation levels

Parallel Computing

Explicit control: Fast but hard **Implicit control: Restrictive but easy**

Threads Processes MPI ZeroMQ	OpenMP	Parallel Computing Libraries on Python , R, and others	Hadoop Spark	SQL: Hive Pig Impala

HPC and Big Data K-Means Examples



Running Time (log_{10})

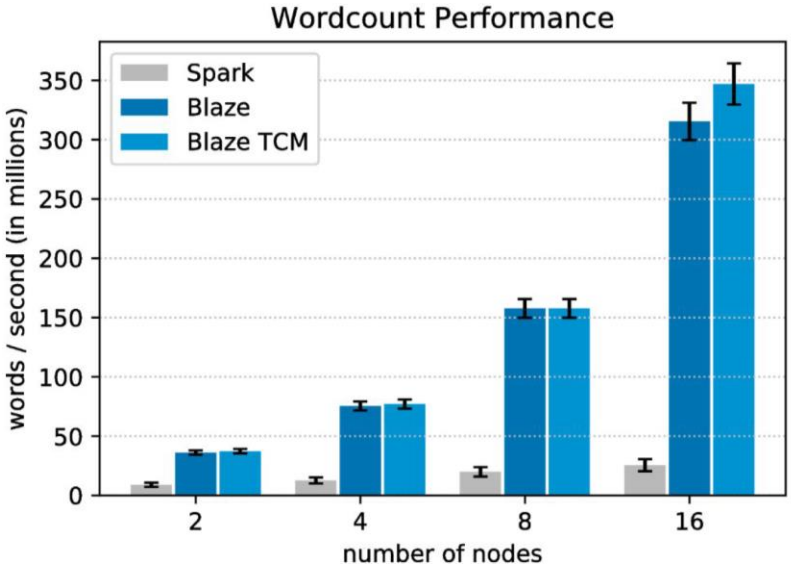
Dr. Geoffrey Fox, Indiana University. (http://arxiv.org/pdf/1403.1528.pdf)

Comparing Spark vs MPI/OpenMP On Word Count MapReduce

Junhao Li
Department of Physics, Cornell University
https://arxiv.org/abs/1811.04875

Spark provides an in-memory implementation of MapReduce that is widely used in the big data industry. MPI/OpenMP is a popular framework for high performance parallel computing. This paper presents a high performance MapReduce design in MPI/OpenMP and uses that to compare with Spark on the classic word count MapReduce task. My result shows that the MPI/OpenMP MapReduce is an order of magnitude faster than Apache Spark.

Speed



Outline

Classical Paradigm

- MPI
- OpenMP

Modern Paradigm

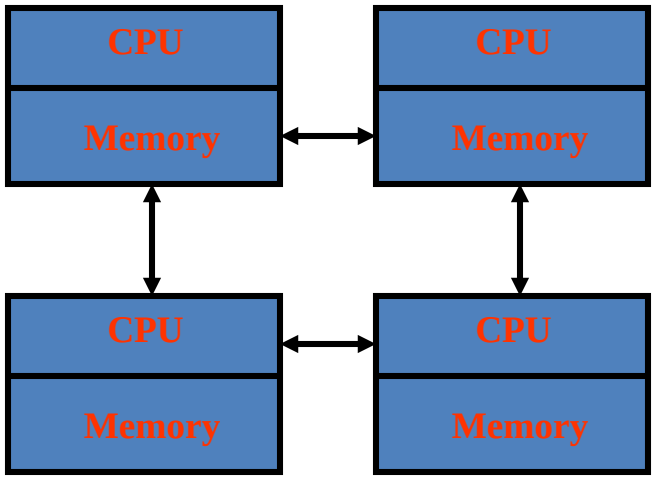
- ~~Python~~
  - (already covered in lab)
- R

GPU Acceleration

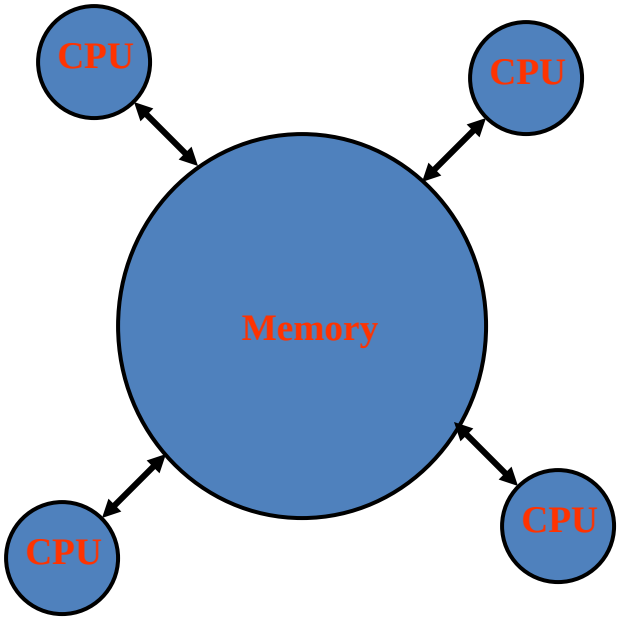
MPI

OpenMP

Distributed

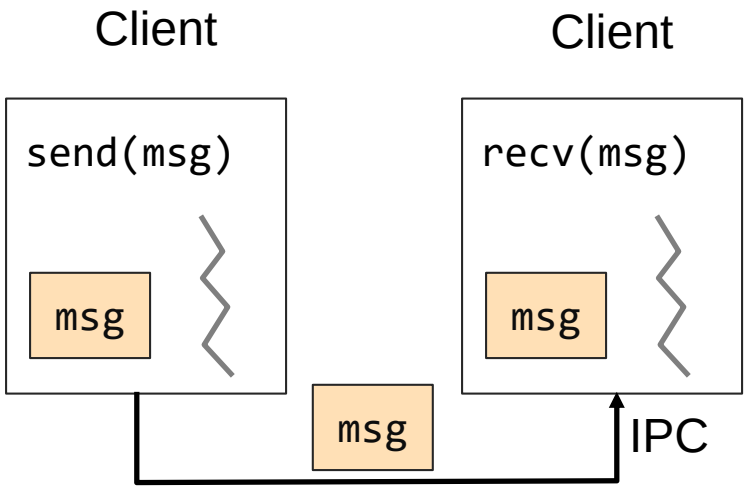


Shared



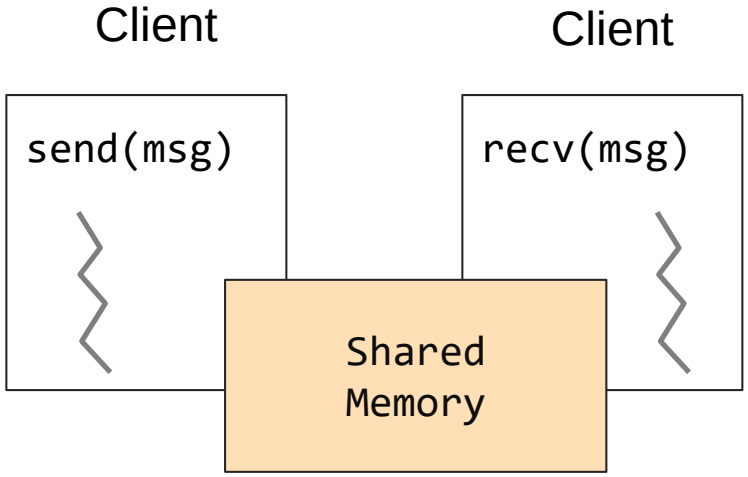
Message Passing vs. Shared Memory

MPI



- Message Passing model: exchange data explicitly via inter-process communication (IPC)
- Application developers define protocol and exchanging format, number of participants, and each exchange

OpenMP

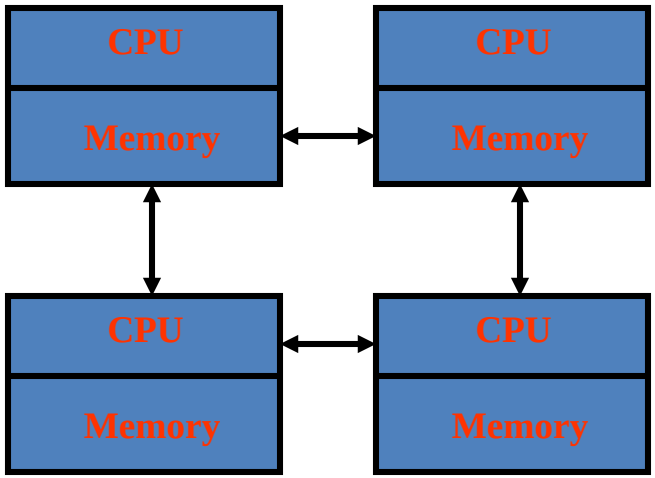


- Shared Memory model: all multiple processes to share data via memory
- Applications must locate and and map shared memory regions to exchange data

`message (msg)`

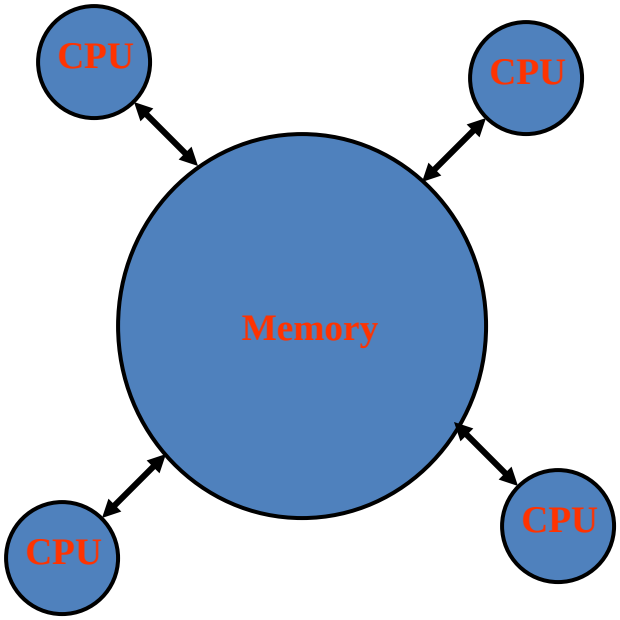
MPI

Distributed



OpenMP

Shared



MPI

- MPI - Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, and parallel programmers
  - Used to create parallel programs based on message passing
  - https://www.mpi-forum.org/docs/
- Portable: one standard, many implementations
  - Available on almost all parallel machines in C and Fortran
  - *De facto* platform for the HPC community

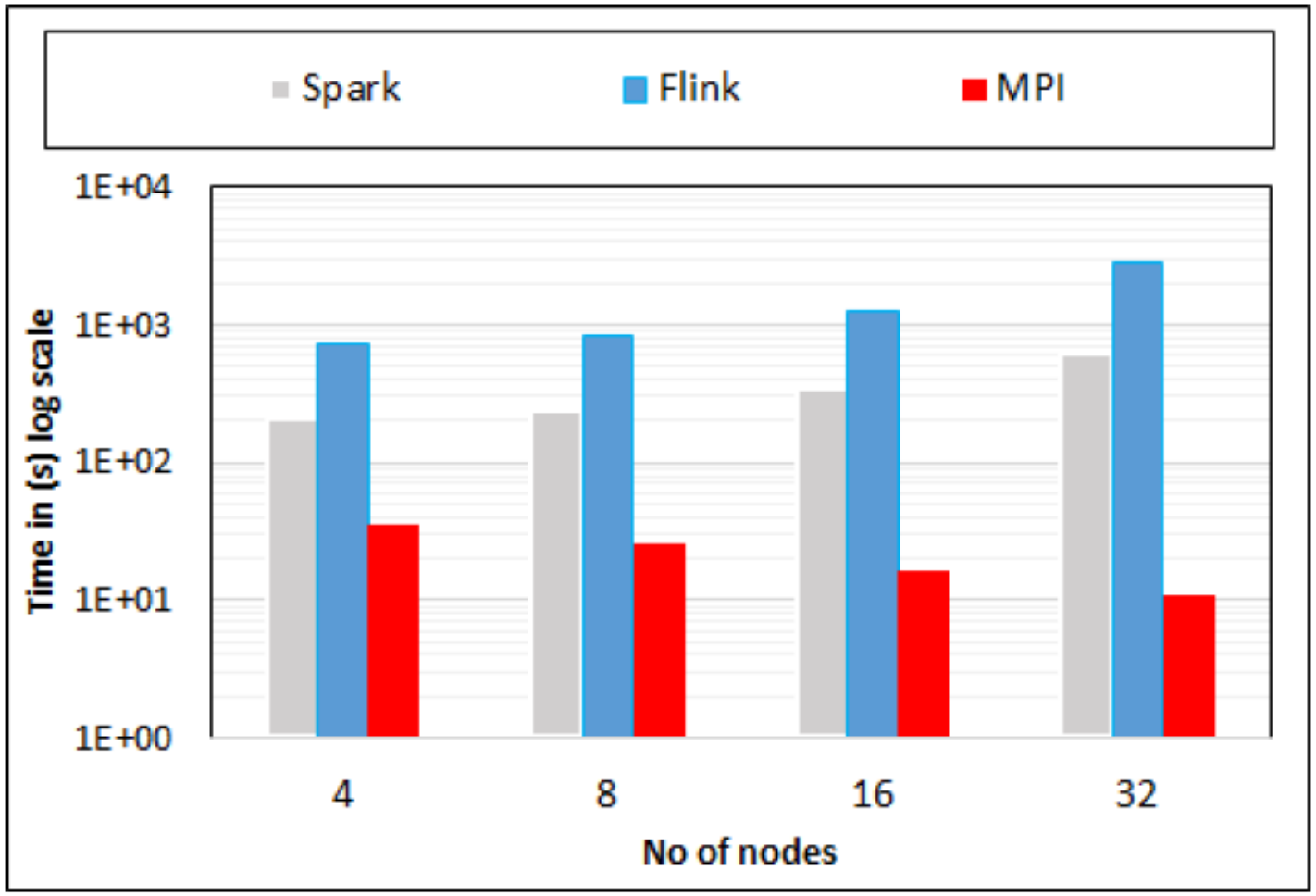
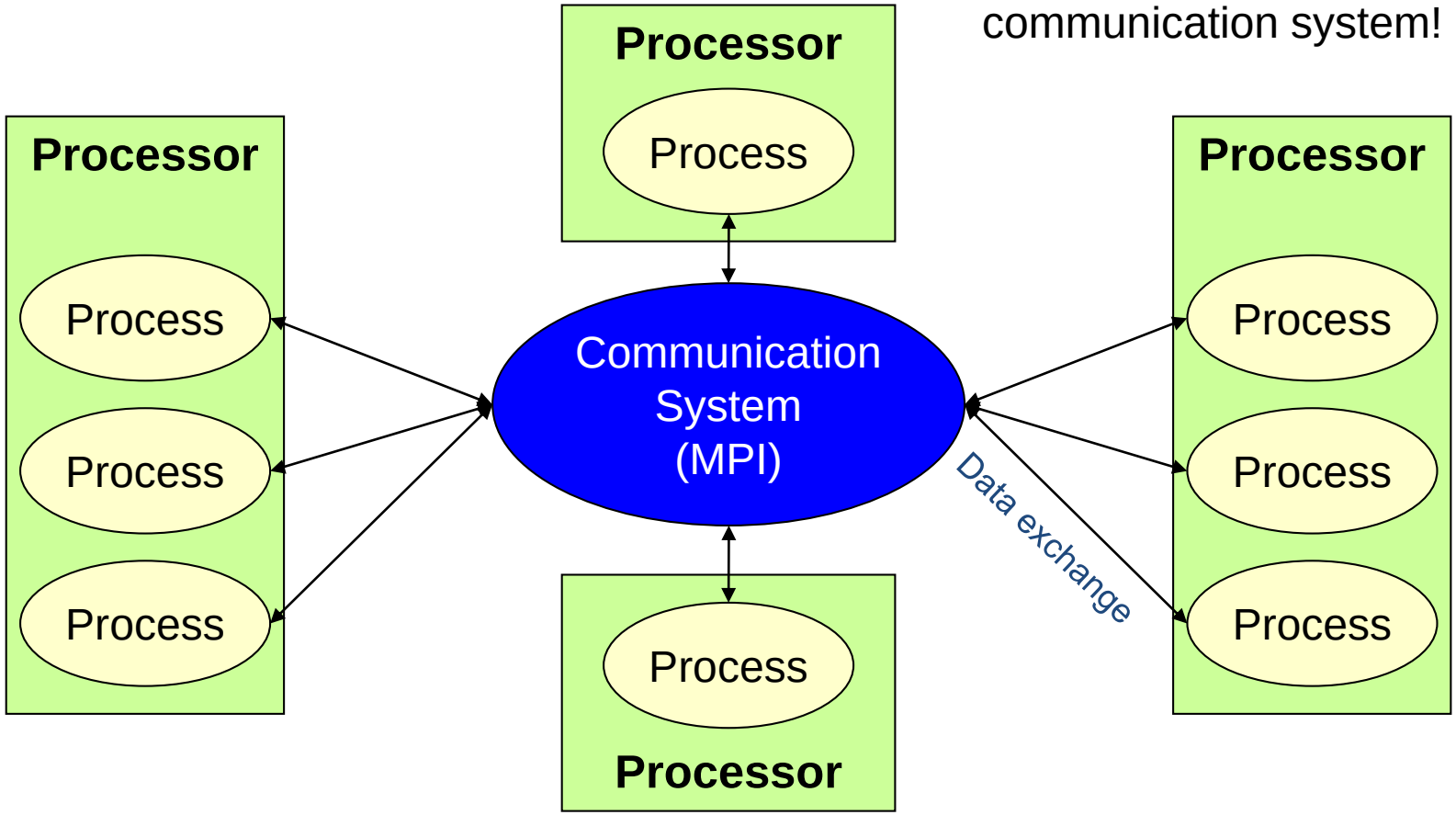


Fig. 9. MDS execution time with 32000 points on varying number of nodes. Each node runs 20 parallel tasks.

Kamburugamuve et al. “Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink”

MPI

Library functions as the only interface to the communication system!



® Class 6 Message Passing Interface (MPI) - HKBU

MPI

- Many parallel programs can be written using just these six functions, only two of which are non-trivial;
    - MPI_INIT
    - MPI_FINALIZE
    - MPI_COMM_SIZE
    - MPI_COMM_RANK
    - MPI_SEND
    - MPI_RECV

MPI

```
#include “mpi.h”
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf(“Hello, world!\n”);
  MPI_Finalize();
  return 0;
}
```

`#include “mpi.h”` provides basic MPI definitions and types.
MPI_Init starts MPI
MPI_Finalize exits MPI
Notes:

- Non-MPI routines are local; this “printf” run on each process
- MPI functions return error codes or MPI_SUCCESS

Point-to-Point Communication

Four types of point-to-point send operations, each of them available in a blocking and a non-blocking variant

	blocking	Non-blocking
Standard	`MPI_Send`	`MPI_Isend`
Buffered	`MPI_Bsend`	`MPI_Ibsend`
Synchronous	`MPI_Ssend`	`MPI_Issend`
Ready	`MPI_Rsend`	`MPI_Irsend`

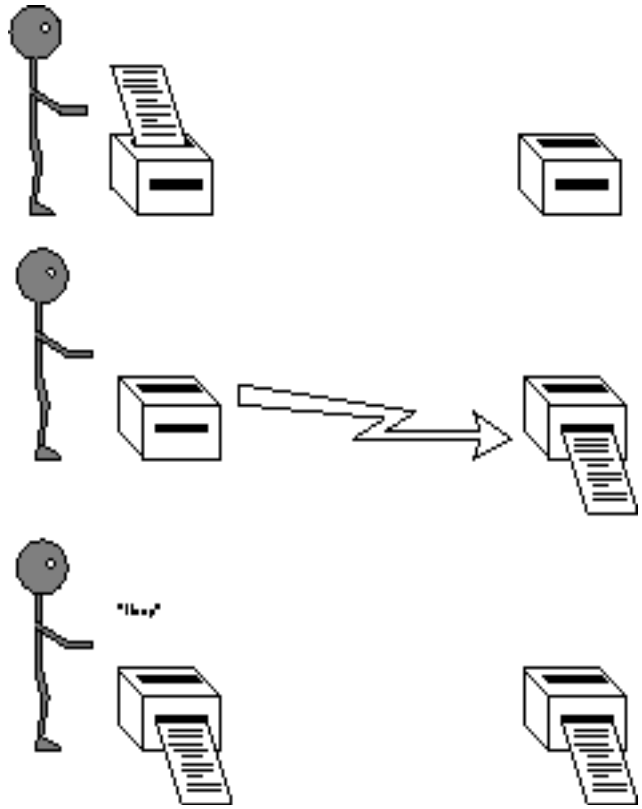
Standard (regular) send: Asynchronous; the system decides whether or not to buffer messages to be sent

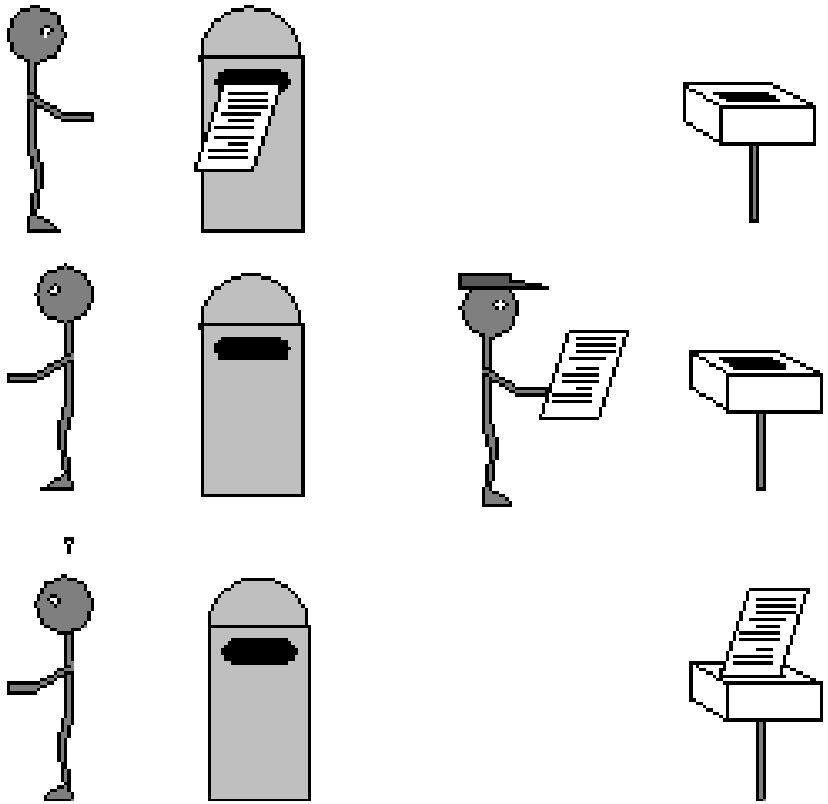
Buffered send: Asynchronous, but buffering of messages to be sent by the system is enforced

Synchronous send: Synchronous, i.e. the send operation is not completed before the receiver has started to receive the message

Ready send: Immediate send is enforced: if no corresponding receive operation is available, the result is undefined

Synchronous Vs. Asynchronous





Point-to-Point Communication

- Meaning of blocking or non-blocking communication (variants with 'I'):

  Blocking: the program will not return from the subroutine call until the copy to/from the system buffer has finished.

  Non-blocking: the program immediately returns from the subroutine call. It is not assured that the copy to/from the system buffer has completed so that user has to make sure of the completion of the copy.

  In MPI,

  - Blocking variant: **`MPI_Recv`**
    - Receive operation is completed when the message has been completely written into the receive buffer
  - Non-blocking variant: **`MPI_Irecv`**
    - Continuation immediately after the receiving has begun
  - Can be combined with each of the four send modes
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

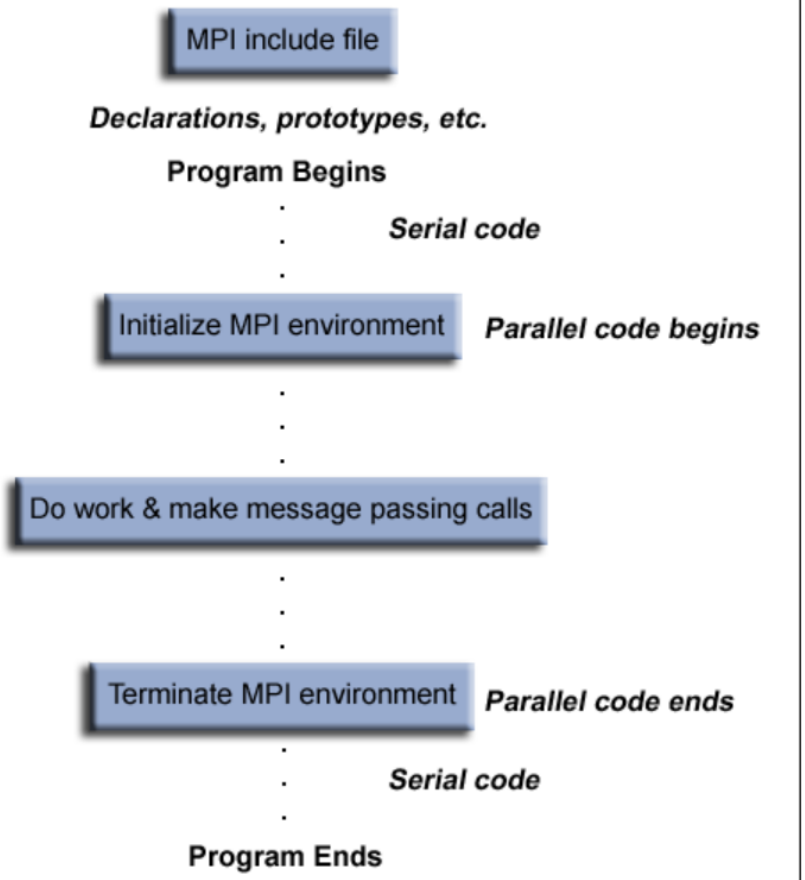
Point-to-Point Communication

- Syntax:

  `MPI_Send(buf,count,datatype,dest,tag,comm)`

  `MPI_Recv(buf,count,datatype,source,tag,comm,status)`
- where
  - int *buf — pointer to the buffer's begin
  - int count — number of data objects
  - int source — process ID of the sending process
  - int dest — process ID of the destination process
  - int tag — ID of the message
  - MPI_Datatype datatype — type of the data objects
  - MPI_Comm comm — communicator space
  - MPI_Status *status — object containing message information
- In the non-blocking versions, there's one additional argument request for checking the completion of the communication.

MPI Full Example 1




```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

MPI Full Example 2

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
  const int tag = 42;         /* Message tag */
  int id, ntasks, source_id, dest_id, err, i;
  MPI_Status status;
  int msg[2]; /* Message array */

  err = MPI_Init(&argc, &argv); /* Initialize
MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
/* Get nr of tasks */
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /*
Get id of this process */
  if (ntasks < 2) {
    printf("You have to use at least 2
processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only
one processor */
    exit(0);
  }
```

```
if (id == 0) {  /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
      err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE,
            tag, MPI_COMM_WORLD, \
                        &status);          /* Receive a
message */
      source_id = status.MPI_SOURCE; /* Get id of sender
*/
      printf("Received message %d %d from process %d\n",
            msg[0], msg[1], \
              source_id);
    }
}
else {    /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Put own identifier in the message */
    msg[1] = ntasks;        /* and total number of
processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag,
            MPI_COMM_WORLD);
  }

  err = MPI_Finalize();         /* Terminate MPI */
  if (id==0) printf("Ready\n");
  exit(0);
  return 0;
}
```

MPI

- Collective operations are called by all processes in a communicator

- MPI_Bcast distributes data from one process (the root) to all others in a communicator.

  Syntax:

  ```
  MPI_Bcast(void *message, int count, MPI_Datatype
  datatype, int root, MPI_Comm comm)
  ```

- MPI_Reduce combines data from all processes in communicator or and returns it to one process

  Syntax:

  ```
  MPI_Reduce(void *message, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm
  comm)
  ```

- In many numerical algorithm, send/receive can be replaced by Bcast/Reduce, improving both simplicity and efficiency.

MPI

Function	MPI Call	Type	Purpose
Initialization	*MPI_Init*	Setup	Prepares system for message-passing functions.
Communication Rank	*MPI_Comm_rank*	Setup	Provides a unique node number for each node.
Communication Size	*MPI_Comm_size*	Setup	Provides the number of nodes in the system.
Finalize	*MPI_Finalize*	Setup	Disable communication services.
Send	*MPI_Send*	P2P	Send data to a matching receive.
Receive	*MPI_Recv*	P2P	Receive data from a matching send.
Send-Receive	*MPI_Sendrecv*	P2P	Simultaneous send and receive between two nodes i.e. send, receive using the same buffer.
Barrier Synchronization	*MPI_Barrier*	Collective	Synchronize all the nodes together.
Broadcast	*MPI_Bcast*	Collective	"Root" node sends same data to all other nodes.

Subramaniyan, R., Aggarwal, V., Jacobs, A. and George, A.D., 2006, May. FEMPI: A Lightweight Fault-tolerant MPI for Embedded Cluster Systems. In *ESA* (pp. 3-9).

MPI Forum

This website contains information about the activities of the MPI Forum, which is the standardization forum for the Message Passing Interface (MPI). You may find standard documents, information about the activities of the MPI forum, and links to comment on the MPI Document using the navigation at the top of the page.

Link to the central MPI-Forum GitHub Presence

MPI 4.0 Standard

The MPI Forum has released a new version of MPI on June 9, 2021. This version is available here:

MPI 4.0 Standard

Comments on the MPI 4.0 should be sent to the MPI Comments mailing list.

Updates

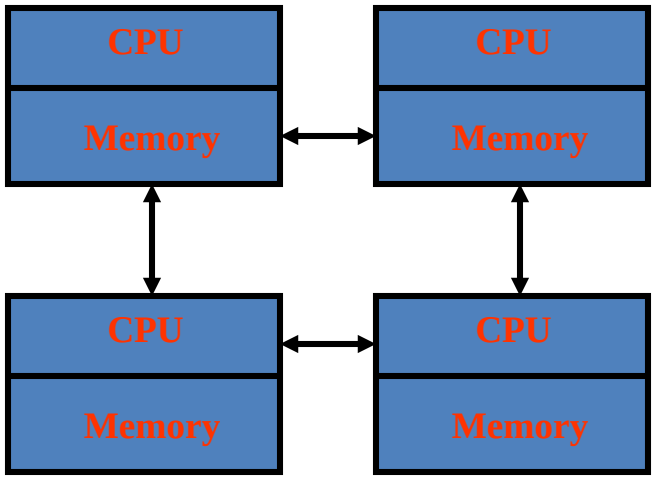
BoF at SC 20, Nov. 18, 2020

The MPI Forum BoF took place on Wednesday November 18th, 2020 at 10am Eastern US time.

https://www.mpi-forum.org/

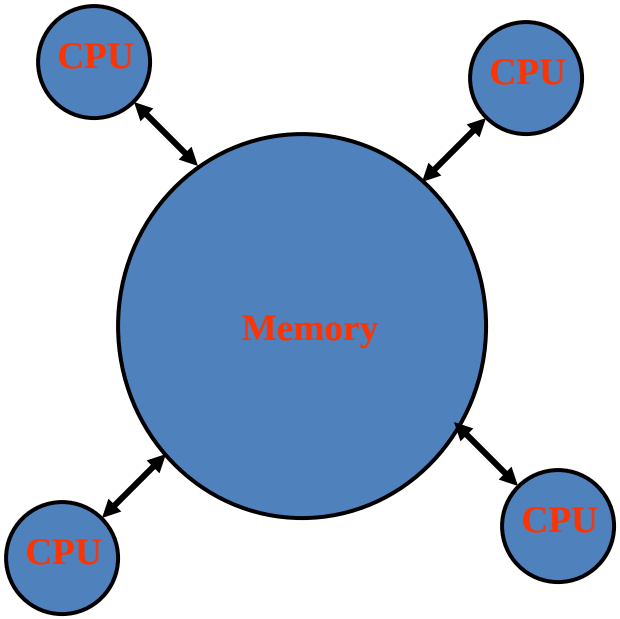
MPI

Distributed



OpenMP

Shared



OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
  - www.openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

Who's Involved

OpenMP Architecture Review Board (ARB)

- ARM
- AMD
- Fujitsu
- HP
- IBM
- Intel
- NEC
- Nvidia
- PGI
- Oracle
- Microsoft
- NASA
- Texas Instruments
- (Many Universities)
- (Many Research Labs)
- ……

https://www.openmp.org/about/members/

Why choose OpenMP ?

- Portable
  - standardized for shared memory architectures
- Simple and Quick
  - incremental parallelization
  - supports both fine-grained and coarse-grained parallelism
  - scalable algorithms without message passing
- Compact API
  - simple and limited set of directives

Traditional Threading in C

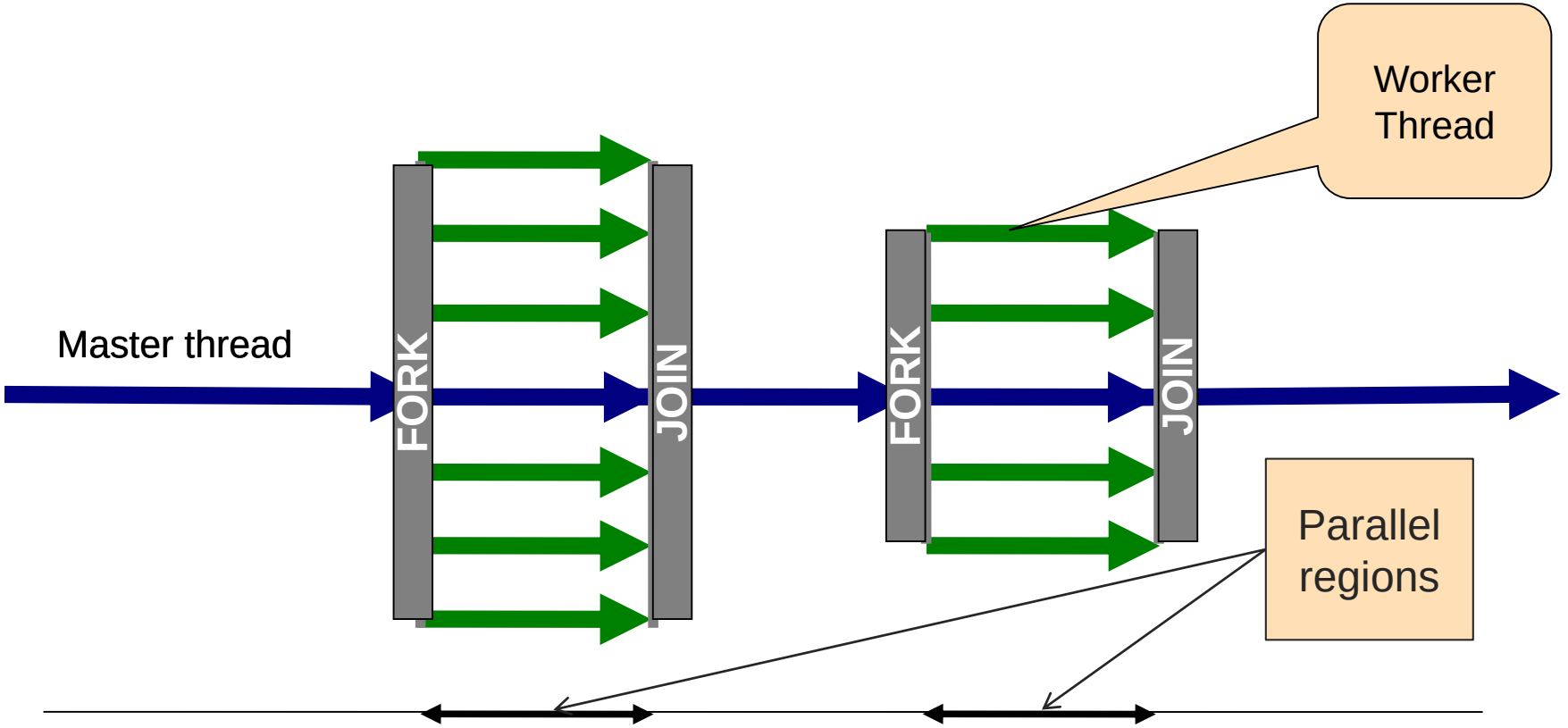
```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_attr_t attr;
  pthread_t threads[16];
  int tn;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], &attr, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

OpenMP

- Traditional Threading libraries are hard to use
  - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

OpenMP execution model

- Fork and Join: Master thread spawns a team of threads as needed



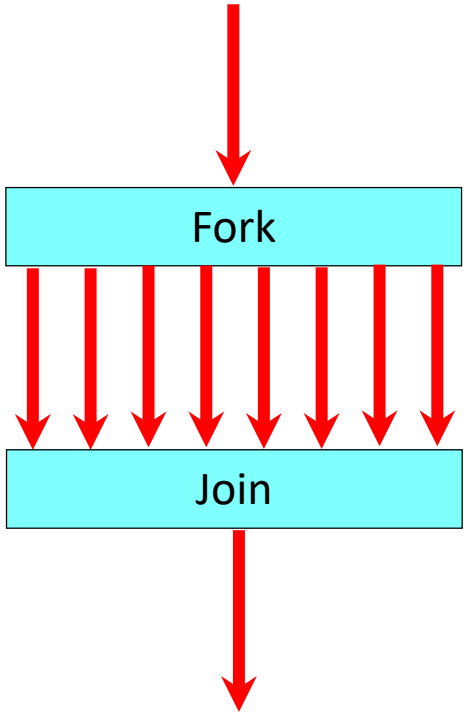
$>gcc -fopenmp foo.c -o foo

OpenMP Example in C

```
#include <stdio.h>
#include <omp.h>

  int main() {

    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
      printf( "Hello, World!\n" );
    }

    return 0;
  }
```

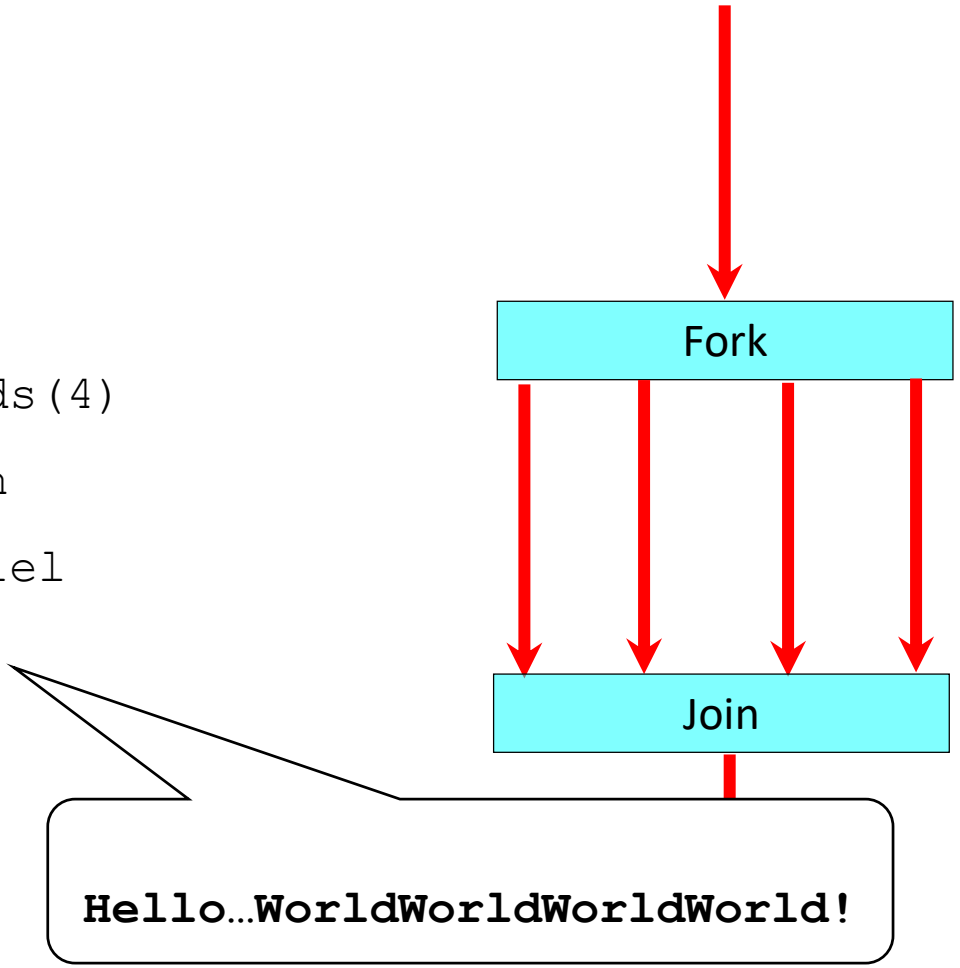
OpenMP

- **Serial regions by default, annotate to create *parallel regions***
  - **Generic parallel regions**
  - **Parallelized loops**
  - **Sectioned parallel regions**
- **Thread-like Fork/Join model**
  - **Arbitrary number of *logical* thread creation/ destruction events**



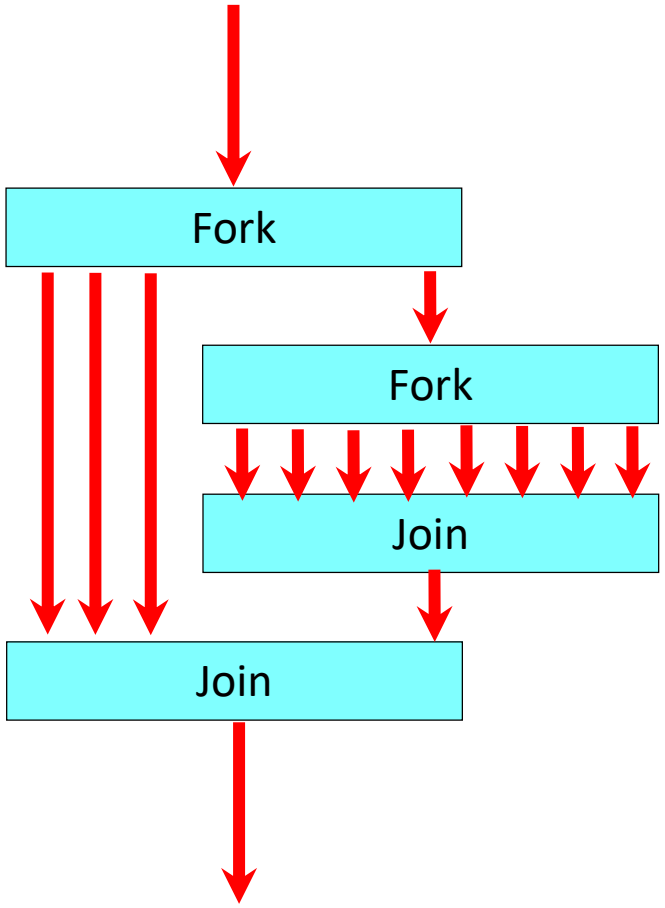
OpenMP

```
#include <stdio.h>
#include <omp.h>

int main() {
  // serial region
  printf("Hello…");
  omp_set_num_threads(4)
  // parallel region
  #pragma omp parallel
  {
    printf("World");
  }
  // serial region
  printf("!");
}
```



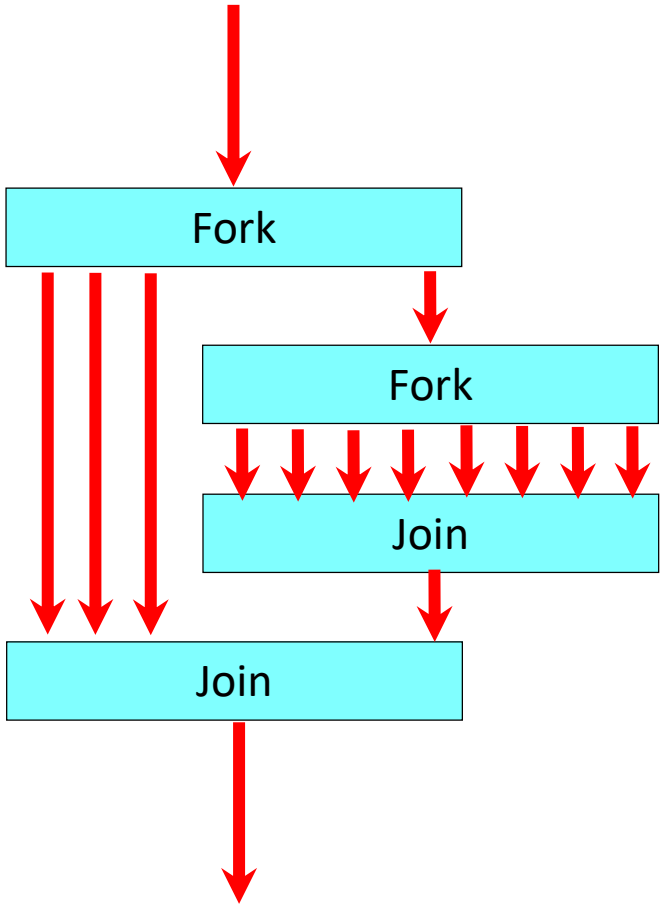
OpenMP

- Fork/Join can be nested
  - Nesting complication is handled “automatically” at compile-time
  - Independent of the number of threads actually running



OpenMP

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
   #pragma omp single
   {
      printf("Level %d: number of threads in the team - %d\n",
            level, omp_get_num_threads());
   }
 }
int main()
{
   omp_set_dynamic(0);
   #pragma omp parallel num_threads(4)
   {
      report_num_threads(1);
      #pragma omp parallel num_threads(8)
      {
         report_num_threads(2);
       }
   }
   return(0);
}
```



OpenMP

- Parallel programs often employ two types of data
  - Shared data, visible to all threads. (often heap-allocated)
  - Private data, visible to a single thread. (often stack-allocated)
- PThreads:
  - Global-scoped variables are shared.
  - Stack-allocated variables are private.
- **OpenMP:**
  - **`shared` variables are shared.**
  - **`private` variables are private.**

```
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calculations here */
  }
}
```

OpenMP

#pragma omp reduction

```
#include <stdio.h>
#include <omp.h>

float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
  for(int i = 0; i < N; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

OpenMP

C / C++ - sections Directive Example

```
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

int i;
float a[N], b[N], c[N], d[N];

/* Some initializations */
for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */

  }  /* end of parallel region */

}
```

https://computing.llnl.gov/tutorials/openMP/#Directives

OpenMP

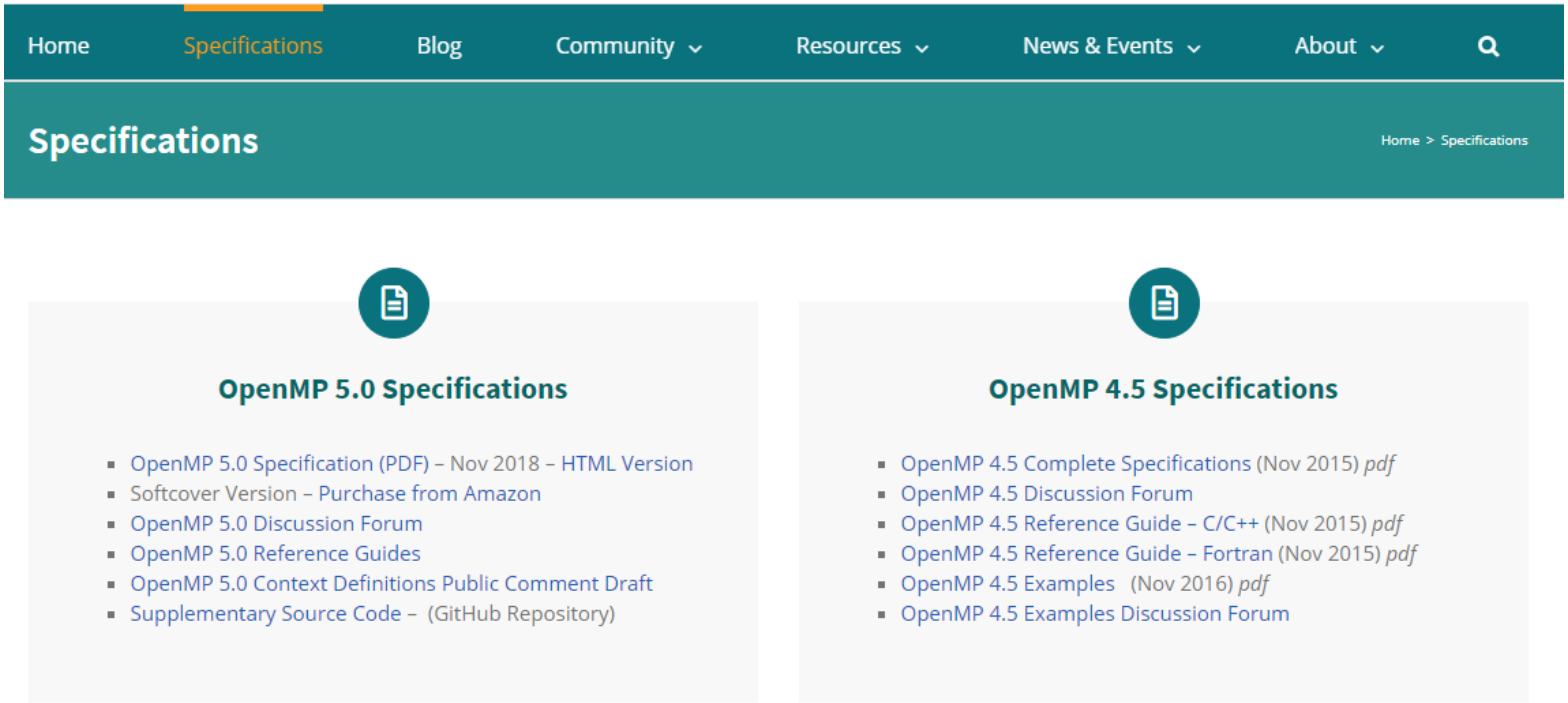
Table 2: OpenMP Functions

Function	Explanation
int omp_get_num_threads()	Gets the number of threads.
int omp_get_thread_num()	Gets the current thread number.
void omp_set_num_threads(int)	Sets the number of threads to be used in future parallel regions.
Locking Functions	
void omp_init_lock(omp_lock_t)*	Initializes a lock.
void omp_set_lock(omp_lock_t)*	Waits and then sets a lock; blocks if the lock is not available.
int omp_test_lock(omp_lock_t)*	Waits and then sets a lock; does not block if the lock is not available.
void omp_unset_lock(omp_lock_t)*	Removes a lock.
void omp_destroy_lock(omp_lock_t)*	Destroys a lock.

admin-magazine.com/HPC/Articles/Programming-with-OpenMP

OpenMP
Enabling HPC since 1997

The OpenMP API specification for parallel programming



https://www.openmp.org/

Lecture Break

R

- There are many parallel computing packages.
- We focus on the following packages:
  - SNOW
  - parallel, foreach, doParallel

R

- SNOW
  - Simple Network Of Workstations

Starting and Stopping clusters
The way to Initialize slave R processes depends on your system configuration. If MPI is installed and there are two computation nodes, then:

> cl <- makeCluster(2, type = "MPI")

Shut down the cluster and clean up any remaining connections

> stopCluster(cl)

R

- SNOW

```
> cl <- makeCluster(2, type = "MPI")
> myfunc <- function(x){x+1}
> myfunc_argument <- 5
> clusterCall(cl, function(x){x+1}, 5)
> stopCluster(cl)
```

```
[[1]]
[1] 6

[[2]]
[1] 6
```

clusterCall(cl, fun, ...)
cl: the computer cluster created with makeCluster.
fun: the function to be applied.
clusterCall calls a specified function with identical arguments on each node in the cluster.
The arguments to clusterCall are evaluated on the master, their values transmitted to the slave nodes which execute the function call.

® Anthony Rossini, Luke Tierney and Na Li, 'Simple parallel statistical computing in R'. ® by Sigal Blay

R

- SNOW

clusterApply(cl, seq, fun, ...)
clusterApply takes a cluster, a sequence of arguments (can be a vector or a list), and a function, and calls the function with the first element of the list on the first node, with the second element of the list on the second node, and so on. The list of arguments must have at most as many elements as there are nodes in the cluster.

```
> cl <- makeCluster(2, type = "MPI")
> clusterApply(cl, 1:2, sum, 3)
[[1]]
[1] 4

[[2]]
[1] 5
> stopCluster(cl)
```

clusterApplyLB is a load balancing version of clusterApply. It hands in a balanced work load to slave nodes when the length of seq is greater than the number of cluster nodes.

R

- SNOW

```
> A<-matrix(1:10, 5, 2)
> A
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> cl <- makeCluster(2, type = "MPI")
> parApply(cl, A, 1, sum)
[1]  7  9 11 13 15
> stopCluster(cl)
```

parApply(cl, X, MARGIN, fun, ...)
X: the array to be used.
MARGIN: a vector giving the subscripts which the function will be applied over. '1' indicates rows, '2' indicates columns, 'c(1,2)' indicates rows and columns.
fun: the function to be applied.
parApply is the parallel version of `the R function apply.

R

- SNOW
  - https://cran.r-project.org/web/packages/snow/index.html

```
clusterSplit(cl, seq)
clusterCall(cl, fun, ...)
clusterApply(cl, x, fun, ...)
clusterApplyLB(cl, x, fun, ...)
clusterEvalQ(cl, expr)
clusterExport(cl, list, envir = .GlobalEnv)
clusterMap(cl, fun, ..., MoreArgs =
NULL, RECYCLE = TRUE)
```

```
parLapply(cl, x, fun, ...)
parSapply(cl, X, FUN, ..., simplify = TRUE,
USE.NAMES = TRUE)
parApply(cl, X, MARGIN, FUN, ...)
parRapply(cl, x, fun, ...)
parCapply(cl, x, fun, ...)
parMM(cl, A, B)
```

R

- Packages: parallel, foreach, doParallel

```
library(parallel)

inputs <- 1:10
processInput <- function(i) {
   i * i
}
numCores <- detectCores()
results =mclapply(inputs, processInput, mc.cores = numCores)

# the above won't work on Windows, but this will:
cl <- makeCluster(numCores)
results = parLapply(cl, inputs, processInput)
stopCluster(cl)
```

® “Easy parallel loops in Python, R, Matlab and Octave” by Nick Elpr

R

- parallel, foreach, doParallel

```
library(parallel)
library(foreach)
library(doParallel)
numCores <- detectCores()
cl <- makeCluster(numCores)
registerDoParallel(cl)
inputs <- 1:10
processInput <- function(i) {
  i * i
}
results <- foreach(i=inputs) %dopar% {
  processInput(i)
}
```

® “Easy parallel loops in Python, R, Matlab and Octave” by Nick Elpr

R

- parallel, foreach, doParallel

```
> x <- iris[which(iris[,5] != "setosa"), c(1,5)]
> trials <- 10000
> ptime <- system.time({
+             r <- foreach(icount(trials), .combine=cbind) %dopar% {
+                         ind <- sample(100, 100, replace=TRUE)
+                         result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
+                         coefficients(result1)
+                         }
+             })[3]
> ptime
elapsed
18.693
```

With 2 cores, it took 18.693 sec.
It can take 30.515 sec on 1 core.

https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf

R

- SparkR

SparkR is an R package that provides a light-weight frontend to use Apache Spark from R. In Spark 3.2.0, SparkR provides a distributed data frame implementation that supports operations like selection, filtering, aggregation etc. (similar to R data frames, dplyr) but on large datasets. SparkR also supports distributed machine learning using MLlib.

```
library(SparkR)
```

```
if (nchar(Sys.getenv("SPARK_HOME")) < 1) {
  Sys.setenv(SPARK_HOME = "/home/spark")
}
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "local[*]", sparkConfig = list(spark.driver.memory = "2g"))
```

```
# Converts Spark DataFrame from an R DataFrame
spark_df <- createDataFrame(mtcars)

# Converts Spark DataFrame to an R DataFrame
collect(spark_df)
```

https://spark.apache.org/docs/latest/sparkr.html

GPU Acceleration

- GPU – Graphics Processing Unit
- Originally designed as a graphics processor
  - single-chip processor for mathematically-intensive tasks
  - transformation of vertices and polygons
  - lighting
  - polygon clipping
  - texture mapping
  - polygon rendering

GPU Acceleration

- Since 1999, computer scientists from various fields started using GPUs to accelerate a range of scientific applications. GPU programming required the use of graphics APIs such as OpenGL and Cg.

- 2002 James Fung (University of Toronto) developed OpenVIDIA.

- NVIDIA greatly invested in GPGPU movement and offered a number of options and libraries, providing seamless user experience for C, C++ and Fortran programmers.

- In November 2006, Nvidia launched CUDA, an API that allows to code algorithms for executions on Geforce GPUs using C programming language.

GPU Acceleration

In 2012, Nvidia presented and demonstrated OpenACC - a set of directives that greatly simplify parallel programming of heterogeneous systems.

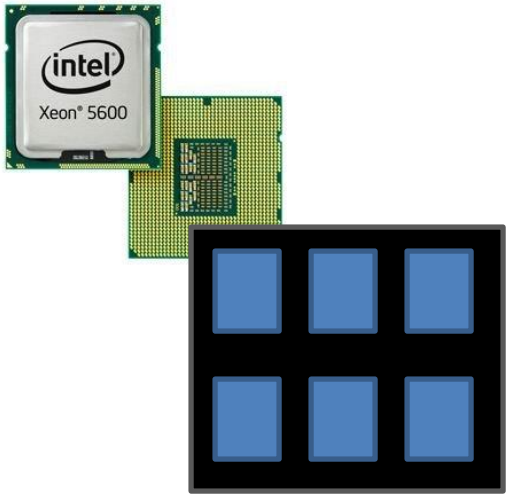
After 2015, the explosive advances in artificial intelligence (e.g. deep learning) has sparked people interests in the wide adoption of GPU accelerations such as “tensorflow-gpu” and “pytorch” (python libraries for deep learning on GPU).

"The performance of GPUs will more than double every two years," coined by Jensen Huang, CEO of NVIDIA in 2018. The law claims that the increased performance is due to AI software as much as it is to increases in chip hardware. Huang's Law is the GPU counterpart to Moore's Law, which deals with the number of transistors on a chip. See Moore's Law and laws.

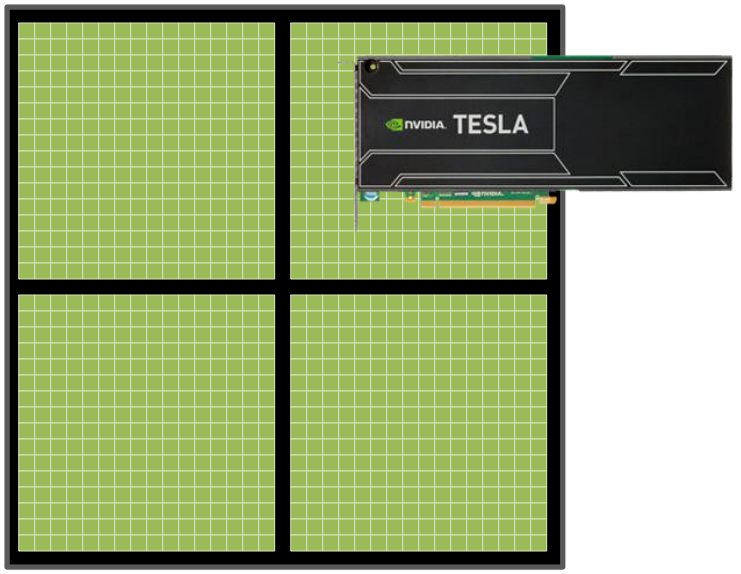
https://www.pcmag.com/encyclopedia/term/huangs-law

GPU Acceleration

CPU



GPU

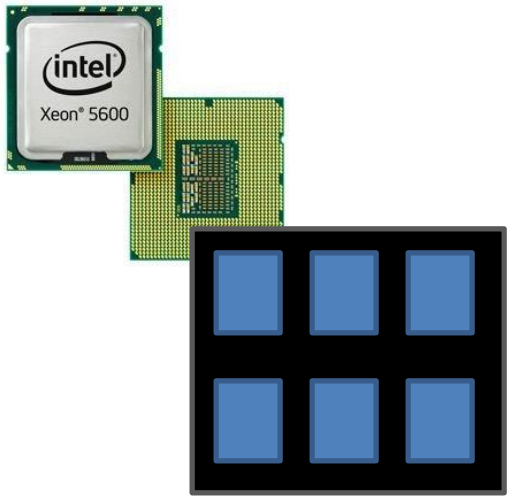


CPUs consist of a few cores optimized for serial processing

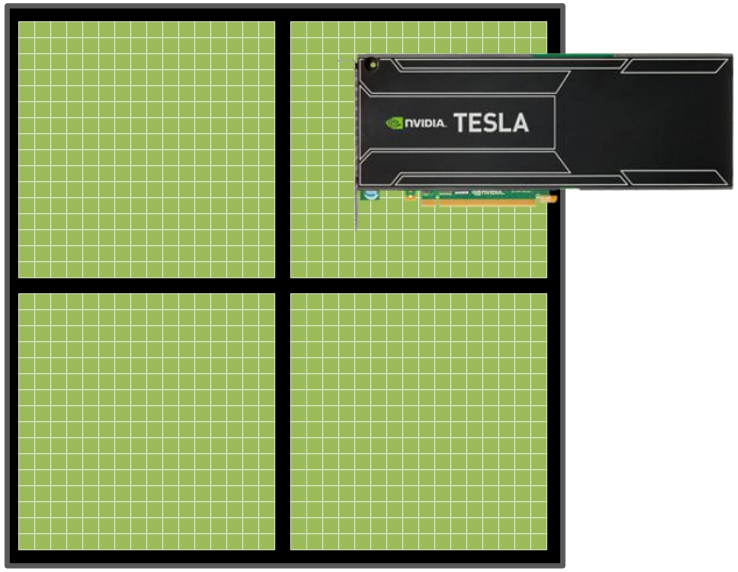
GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance

GPU Acceleration

CPU



GPU



Intel Xeon X5650:

Clock speed: **2.66** GHz
4 instructions per cycle
CPU - **6** cores

2.66 x 4 x 6 =
63.84 Gigaflops double precision

NVIDIA Tesla M2070:

Core clock: **1.15**GHz
Single instruction
448 CUDA cores

1.15 x 1 x 448 =
515 Gigaflops double precision

GPU Acceleration

- GEMM (General Matrix Multiplication) is the fundamental building block for many operations in machine learning (e.g. deep neural networks), for example fully-connected layers, recurrent layers such as RNNs, LSTMs or GRUs, and convolutional layers.

- GEMM is defined as the operation:

$$C = \alpha AB + \beta C$$

with A and B as matrix inputs, α and β as scalar inputs, and C as a pre-existing matrix which is overwritten by the output. A plain matrix product AB is a GEMM with α equal to one and β equal to zero.

- For example, in the forward pass of a fully-connected layer, the weight matrix would be argument A, incoming activations would be argument B, and α and β would typically be 1 and 0, respectively. β can be 1 in some cases, for example, if we're combining the addition of a skip-connection with a linear operation.

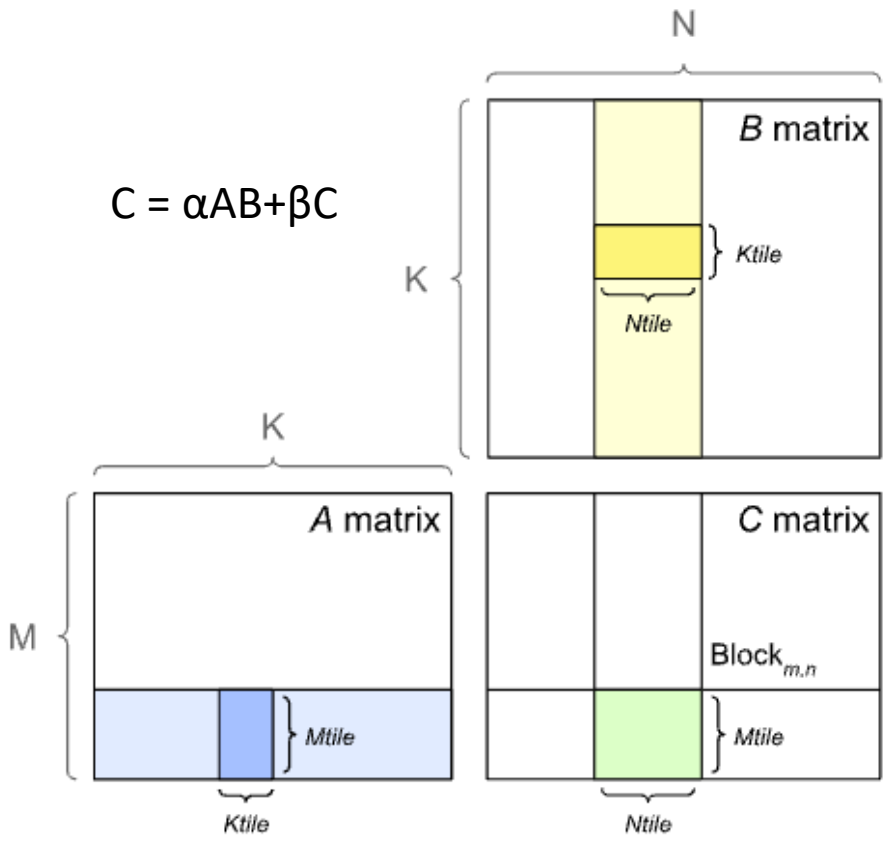
https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

GPU Acceleration

GPUs implement GEMMs by partitioning the output matrix into tiles, which are then assigned to thread blocks. Tile size usually refers to the dimensions of these tiles (*Mtile* x *Ntile* in Figure 1).

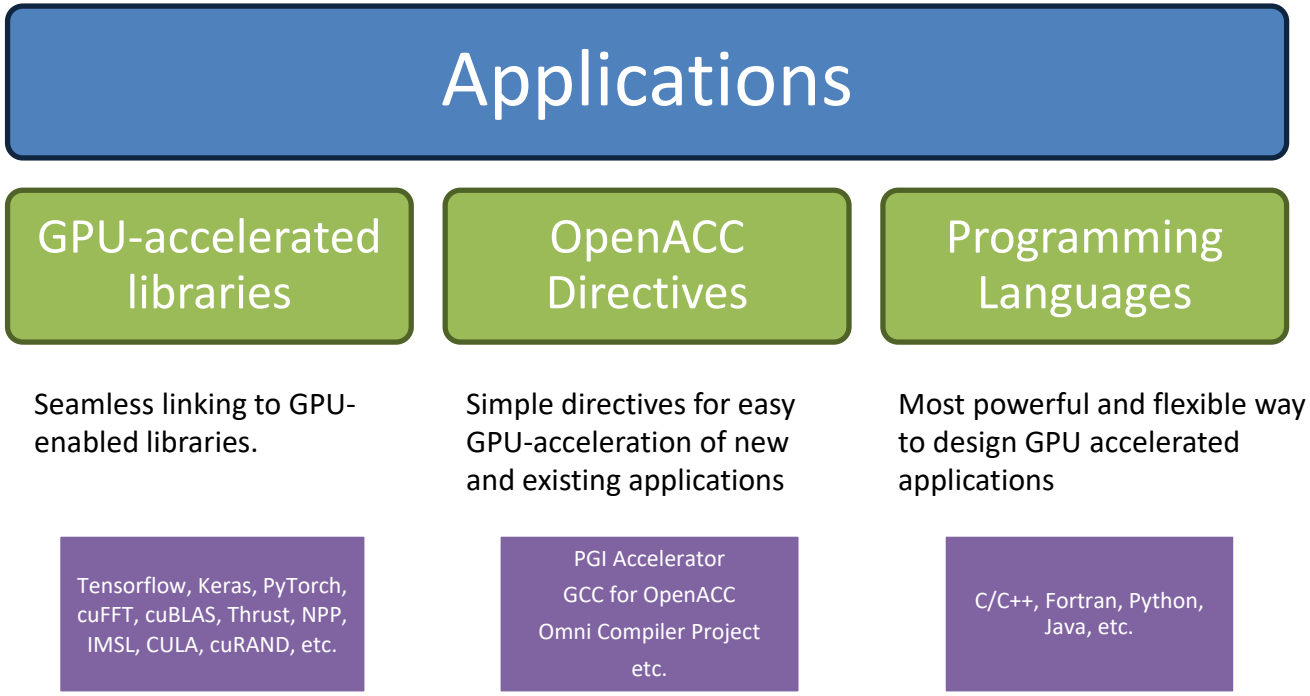
Each thread block computes its output tile by stepping through the K dimension in tiles, loading the required values from the A and B matrices, and multiplying and accumulating them into the output.

Figure 1. Tiled outer product approach to GEMMs



https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html

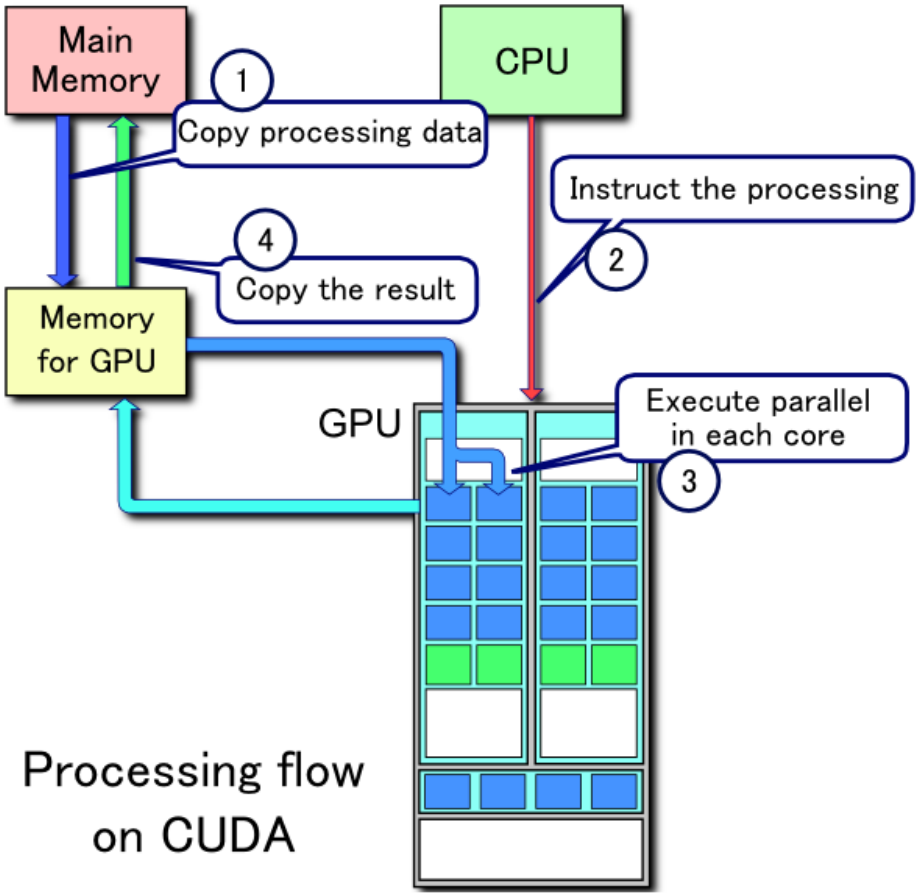
GPU Acceleration



GPU Acceleration
GPU Accelerated Libraries

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.

The CUDA platform is designed to work with programming languages such as C, C++, and Fortran.



Processing flow on CUDA

https://en.wikipedia.org/wiki/CUDA

GPU Acceleration
GPU Accelerated Libraries

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.

The CUDA platform is designed to work with programming languages such as C, C++, and Fortran.

SIMD Programming Model

```
__global__ void kernel(float* x, float* y, float* z, int n){
           int idx= blockIdx.x * blockDim.x + threadIdx.x;
           if(idx <  n) z[idx] = x[idx] * y[idx];
}
int main(){
    ...
    cudaMalloc(...);
    cudaMemcpy(...);
    kernel <<<num_blocks,  block_size>>> (...);
    cudaMemcpy(...);
    cudaFree(...);
    ...
}
```


https://en.wikipedia.org/wiki/CUDA

GPU Acceleration
GPU Accelerated Libraries

A powerful library of parallel algorithms and data structures;

It provides a flexible, high-level interface for GPU programming;

For example, the thrust::sort algorithm delivers **5**x to **100**x faster sorting performance than STL and TBB

```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device (GPU)
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host (Main Memory)
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

GPU Acceleration
GPU Accelerated Libraries

cuBLAS

- A GPU-accelerated version of the complete standard BLAS library*;
- 6x to 17x faster performance than the latest MKL BLAS
- Complete support for all 152 standard BLAS routines
- Single, double, complex, and double complex data types

```
1  int main() {
2      // Allocate 3 arrays on CPU
3      int nr_rows_A, nr_cols_A, nr_rows_B, nr_cols_B, nr_rows_C, nr_cols_C;
4
5      // for simplicity we are going to use square arrays
6      nr_rows_A = nr_cols_A = nr_rows_B = nr_cols_B = nr_rows_C = nr_cols_C = 3;
7
8      thrust::device_vector<float> d_A(nr_rows_A * nr_cols_A), d_B(nr_rows_B * nr_cols_B), d_C(
9
10     // Fill the arrays A and B on GPU with random numbers
11     GPU_fill_rand(thrust::raw_pointer_cast(&d_A[0]), nr_rows_A, nr_cols_A);
12     GPU_fill_rand(thrust::raw_pointer_cast(&d_B[0]), nr_rows_B, nr_cols_B);
13
14     // Optionally we can print the data
15     std::cout << "A =" << std::endl;
16     print_matrix(d_A, nr_rows_A, nr_cols_A);
17     std::cout << "B =" << std::endl;
18     print_matrix(d_B, nr_rows_B, nr_cols_B);
19
20     // Multiply A and B on GPU
21     gpu_blas_mmul(thrust::raw_pointer_cast(&d_A[0]), thrust::raw_pointer_cast(&d_B[0]), thrus
22
23     //Print the result
24     std::cout << "C =" << std::endl;
25     print_matrix(d_C, nr_rows_C, nr_cols_C);
26
27     return 0;
28 }
```

***Basic Linear Algebra Subprograms (BLAS)**

https://solarianprogrammer.com/2012/05/31/matrix-multiplication-cuda-cublas-curand-thrust/

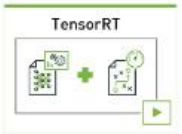
® "GPU Programming" by Scientific Computing and Visualization at Boston University

GPU Acceleration

Deep Learning Libraries

cuDNN

GPU-accelerated library of primitives for deep neural networks



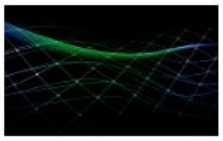
GPU-accelerated neural network inference library for building deep learning applications

Advanced GPU-accelerated video inference library

Linear Algebra and Math Libraries

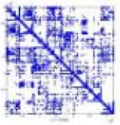
cuBLAS

GPU-accelerated standard BLAS library



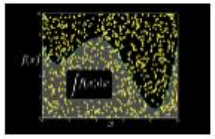
CUDA Math Library

GPU-accelerated standard mathematical function library



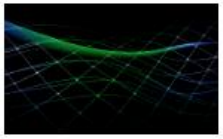
cuSPARSE

GPU-accelerated BLAS for sparse matrices



cuRAND

GPU-accelerated random number generation (RNG)



cuSOLVER

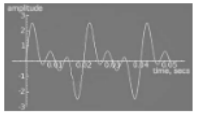
Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications

AmgX

GPU accelerated linear solvers for simulations and implicit unstructured methods

GPU Acceleration

Signal, Image and Video Libraries



cuFFT

GPU-accelerated library for Fast Fourier Transforms

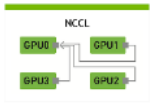
NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

NVIDIA Codec SDK

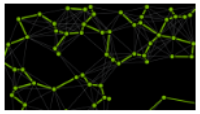
High-performance APIs and tools for hardware accelerated video encode and decode

Parallel Algorithm Libraries



NCCL

Collective Communications Library for scaling apps across multiple GPUs and nodes



nvGRAPH

GPU-accelerated library for graph analytics

Thrust

GPU-accelerated library of parallel algorithms and data structures

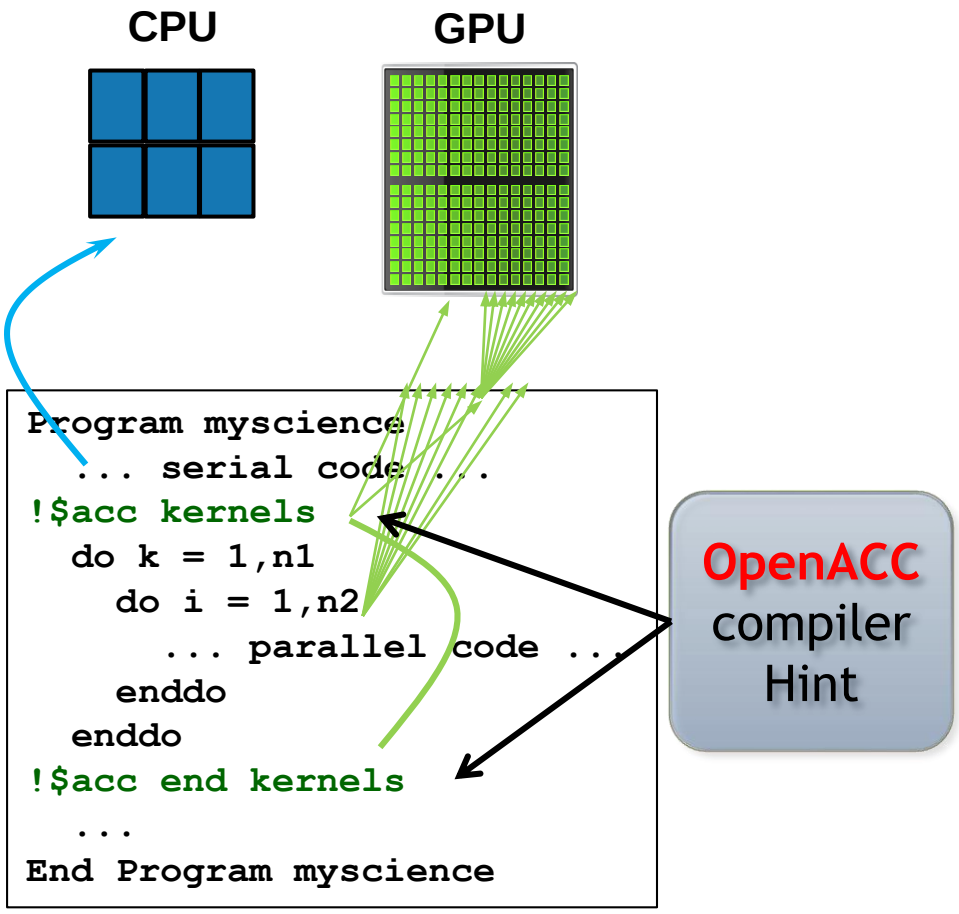
Partner Libraries

GPU-accelerated open-source library for computer vision, image processing and machine learning, now supporting real-time operation

Open-source multi-media framework with a library of plugins for audio and video processing

GPU-accelerated open source library for matrix, signal, and image processing

GPU Acceleration



Your original Fortran or C code

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

GPU Acceleration

- Example: Compute *a*x + y*, where *x* and y are vectors, and *a* is a scalar.

C (saxpy_array.c)

```
int main(int argc, char **argv){
   int N=1000;
   float a = 3.0f;
   float x[N], y[N];
   for (int i = 0; i < N; ++i) {
      x[i] = 2.0f;
      y[i] = 1.0f;
   }
   #pragma acc kernels
   for (int i = 0; i < N; ++i) {
      y[i] = a * x[i] + y[i];
   }
}
```

Fortran

```
program main
  integer :: n=1000, i
  real :: a=3.0
  real, allocatable :: x(:), y(:)
  allocate(x(n),y(n))
  x(1:n)=2.0
  y(1:n)=1.0
  !$acc kernels
  do i=1,n
     y(i) = a * x(i) + y(i)
  enddo
  !$acc end kernels
end program main
```

GPU Acceleration

```
$ pgcc -acc -Minfo=accel saxpy_array.c -o saxpy_array
main:
    17, Generating copyin(x[:1000])
        Generating copy(y[:1000])
    19, Loop is parallelizable
        Accelerator kernel generated
        Generating Tesla code
        19, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

https://www.pgroup.com/doc/openacc14_gs.pdf

Li, Xuechao, and Po-Chou Shih. "An Early Performance Comparison of CUDA and OpenACC." *MATEC Web of Conferences*. Vol. 208. EDP Sciences, 2018

GPU Acceleration

- The loop is not parallelized if there is data dependency. For example,

```
#pragma acc kernels
 for (int i = 0; i < N-1; i++) {
    x[i] = a * x[i+1] ;
 }
```

- The compiling output:

```
.....
14, Loop carried dependence of x-> prevents parallelization
      Loop carried backward dependence of x-> prevents vectorization
      Accelerator scalar kernel generated
      Loop carried backward dependence of x-> prevents vectorization
```

- The compiler creates a serial program, which runs slower on GPU than on CPU!

GPU Acceleration
MATLAB with GPU-acceleration

Use GPUs with MATLAB through Parallel Computing Toolbox

- GPU-enabled MATLAB functions such as fft, filter, and several linear algebra operations
- GPU-enabled functions in toolboxes: Communications System Toolbox, Neural Network Toolbox, Phased Array Systems Toolbox and Signal Processing Toolbox
- CUDA kernel integration in MATLAB applications, using only a single line of MATLAB code

```
A=rand(2^16,1);

B=fft(A);
```

```
A=gpuArray(rand(2^16,1));

B=fft(A);
```

GPU Acceleration

Deep Learning (Python Tensorflow) with GPU

```
……
def matpow(M, n):
    if n < 1: #Abstract cases where n < 1
        return M
    else:
        return tf.matmul(M, matpow(M, n-1))

with tf.device('/gpu:0'):
    a = tf.placeholder(tf.float32, [10000, 10000])
    b = tf.placeholder(tf.float32, [10000, 10000])
    # Compute A^n and B^n and store results in c1
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))

with tf.device('/cpu:0'):
  sum = tf.add_n(c1) #Addition of all elements in c1, i.e. A^n + B^n
……
```


https://gist.github.com/j-min/baae1aa56e861cab9831b3722755ae6d

GPU Acceleration

Deep Learning (Python PyTorch) with GPU

Part of the model on CPU and part on the GPU

Let's look at a small example of implementing a network where part of it is on the CPU and part on the GPU

```
device = torch.device("cuda:0")

class DistributedModel(nn.Module):

    def __init__(self):
        super().__init__(
            embedding=nn.Embedding(1000, 10),
            rnn=nn.Linear(10, 10).to(device),
        )

    def forward(self, x):
        # Compute embedding on CPU
        x = self.embedding(x)

        # Transfer to GPU
        x = x.to(device)

        # Compute RNN on GPU
        x = self.rnn(x)
        return x
```

https://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html

GPU Acceleration
Programming Languages

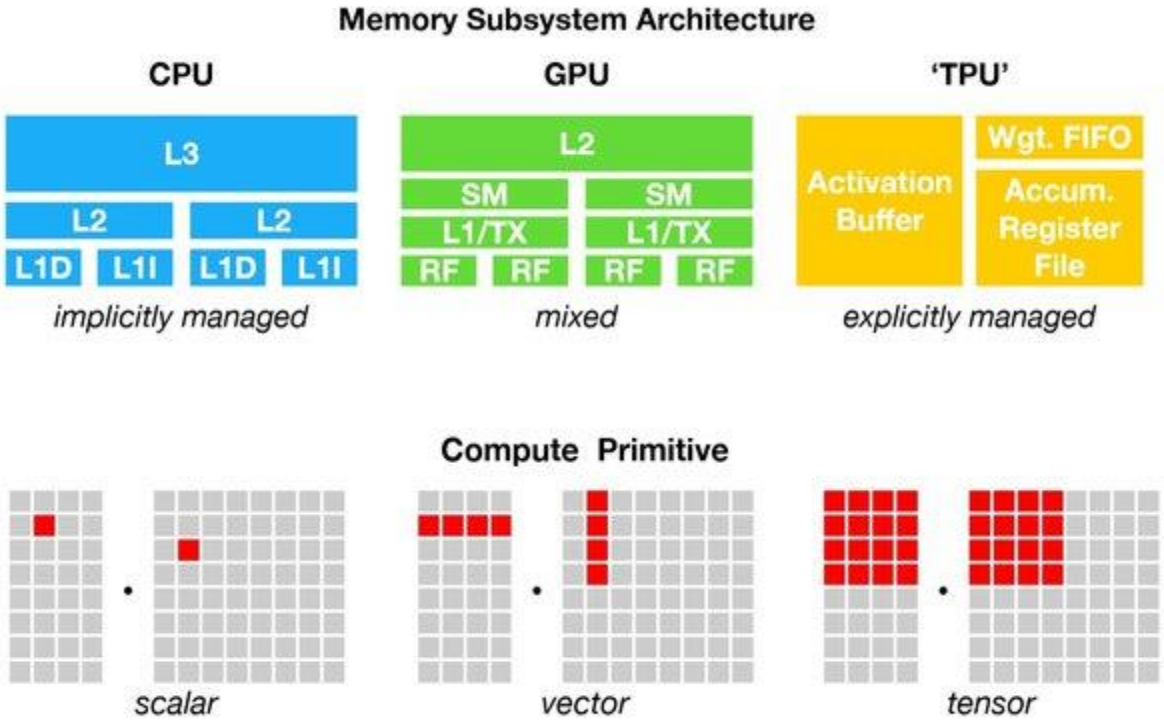
Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran, ……
C	OpenACC, CUDA C, Thrust, ……
C++	OpenACC, CUDA C++, Thrust, ……
Python	PyCUDA, Copperhead, Tensorflow, …

GPU Acceleration

- When should GPU Acceleration be used?
  - **Computationally Intensive Scenarios**
  - The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.
    - Single precision performance is better than double precision.
    - Algorithms that require little communication between threads.
  - **Massively Parallel Scenarios**
  - The computations can be broken down into hundreds or thousands of independent units of work.
    - Doing the same calculation with many pieces of input data.
    - The number of processing steps should be at least an order of magnitude greater than the number of pieces of input/output data.
    - Algorithms where most of the cores will follow the same branch paths most of the time.

® "GPU Programming" by Scientific Computing and Visualization at Boston University
® "Introduction to GPU Programming with CUDA and OpenACC " Alabama Supercomputer Center Alabama Research and Education Network

(Reference Only)
CPU vs GPU vs TPU



1.The chip starts up, buffers and DDR3 empty
2.The user loads a TPU-compiled model, placing weights into the DDR3 memory
3.The host fills the activation buffer with input values
4.A control signal is sent to load a layer of weights into the MXU (through the weight FIFO)
5.The host triggers execution and activations propagate through the MXU and into the accumulators
6.As outputs come out, they're run through the activation pipeline, and the new layers replace old ones in the buffer
7.We repeat 4 through 6 until we reach the last layer
8.Activations of the last layer are sent back to the host machine

https://www.quora.com/What-is-TPU-and-GPU-Why-and-when-do-we-need-them
https://medium.com/@antonpaquin/whats-inside-a-tpu-c013eb51973e

(Reference Only)
CPU vs GPU vs TPU for Deep Learning

Observation	ParaDnn*	Proof	Insight/Explanation
1. TPU does not exploit the parallelism from the model depth (layer count).	✓	Fig 2	To design/upgrade new specialized systems, architects need to consider interactions between the operation mix from key workloads (arithmetic intensity) and system configurations (FLOPS, memory bandwidth/ capacity, and intra-chip and host-device interconnect). TPU serves as a great example.
2. Many FC and CNN operations are bottlenecked by TPU memory bandwidth.	✓	Fig 3	
3. TPU suffers large overheads due to inter-chip communication bottlenecks.	✓	Fig 4	
4. TPU performance can be improved by $\geq 34\%$ by improving data infeed.	-	Fig 5	
5. TPU v3 optimizes compute-bound MatMuls by 2.3×, memory-bound ones by 3×, and large embeddings by $> 3\times$, compared to v2.	✓	Fig 6	
6. The largest FC models prefer CPU due to memory constraints.	✓	Fig 7	Need for model parallelism on GPU and TPU.
7. Models with large batch size prefer TPU. Those with small batch size prefer GPU.	-	Fig 8 Fig 10	Large batches pack well on systolic arrays; warp scheduling is flexible for small batches.
8. Smaller FC models prefer TPU and larger FC models prefer GPU.	✓	Fig 8	FC needs more memory bandwidth per core (GPU).
9. TPU speedup over GPU increases with larger CNNs.	✓	Fig 10	TPU architecture is highly optimized for large CNNs.
10. TPU achieves 2× (CNN) and 3× (RNN) FLOPS utilization compared to GPU.	✓	Fig 11	TPU is optimized for both CNN and RNN models.
11. GPU performance scales better with RNN embedding size than TPU.	✓	Fig 10	GPU is more flexible to parallelize non-MatMuls.
12. Within seven months, the software stack specialized for TPU improved by up to 2.5× (CNN), 7× (FC), and 9.7× (RNN).	✓	Fig 12	It is easier to optimize for certain models than to benefit all models at once.
13. Quantization from 32 bits to 16 bits significantly improves TPU and GPU performance.	-	Fig 5 Fig 12	Smaller data types save memory traffic and enable larger batch sizes, resulting in super-linear speedups.
14. TensorFlow and CUDA teams provide substantial performance improvements in each update.	✓	Fig 12	There is huge potential to optimize compilers even after the hardware has shipped.

* Without ParaDnn the insights are not revealed, and/or lack deep explanations.

Table 1: A summary of major observations and insights grouped by section of the paper.

Different types of processors are suited for different types of machine learning models. TPUs are well suited for CNNs, while GPUs have benefits for some fully-connected neural networks, and CPUs could have advantages for some RNNs, by considering costs, bandwidth, and latency.[5]

[5] Wang, Yu Emma; Wei, Gu-Yeon; Brooks, David (2019-07-01). "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning". arXiv:1907.10701 [cs.LG].

https://en.wikipedia.org/wiki/Tensor_Processing_Unit

Summary

- Classical Paradigm
  - OpenMP
  - MPI
- Modern Paradigm
  - ~~Python~~ (already covered in lab)
  - R
- GPU Acceleration

Reference

OpenMP

Table 1: Clauses

Clause	Meaning
shared(variable_list)	Only one version of the variable exists, and all parallel program sections access it. All threads have read and write access. If a thread changes a variable, this also affects the other threads. Default: All variables are *shared()* except the loop variables in *#pragma omp for*.
private(variable_list)	Each thread has a private, non initialized copy of the variable. Default: Only loop variables are private.
default(shared|private|none)	Defines the default behavior of the variables: *none* means that you must explicitly declare each variable as *shared()* or *private()*.
firstprivate(variable_list)	Just like *private()*; however, in this case, all copies are initialized with the value of the variable before the parallel loop/region.
lastprivate(variable_list)	The variable is assigned the value from the last thread to change the variable in sequential processing after the parallel loop/region has been completed.

```
a = 0 ; b = 0 ;
#pragma omp parallel for private(i) shared(x, y, n) reduction(+:a, b)
for (i=0; i<n; i++) {
  a = a + x[i] ;
  b = b + y[i] ;
}
```

admin-magazine.com/HPC/Articles/Programming-with-OpenMP