Parallelization Strategies

Eckart Bindewald

Frederick National Laboratory for Cancer Research

RNA Biology Laboratory

RNA Structure and Design Section

eckart@mail.nih.gov

Parallelism in a Fast-Food Drive-Through …

1. Parallel Ordering
2. Sequential purchasing
3. Parallel food preparation
4. Sequential checkout
5. …

Overview

- Shared memory approaches:
  - Threading
  - GPU
- Cluster computing:
  - ad hoc clusters
  - Apache Spark

Threads, Threading

- Thread is smallest sequence of instructions that can be managed by a scheduler
- Threads share resources if they are part of the same process
- Typically no sharing of resources between processes
- Threads of a process share the executable code and the values of its variables
- A scheduler may orchestrate parallel execution of threads on different cores or time-sliced (serial, non-parallel execution).

OpenMP: A Cross-Language Standard for Multi-core Processing

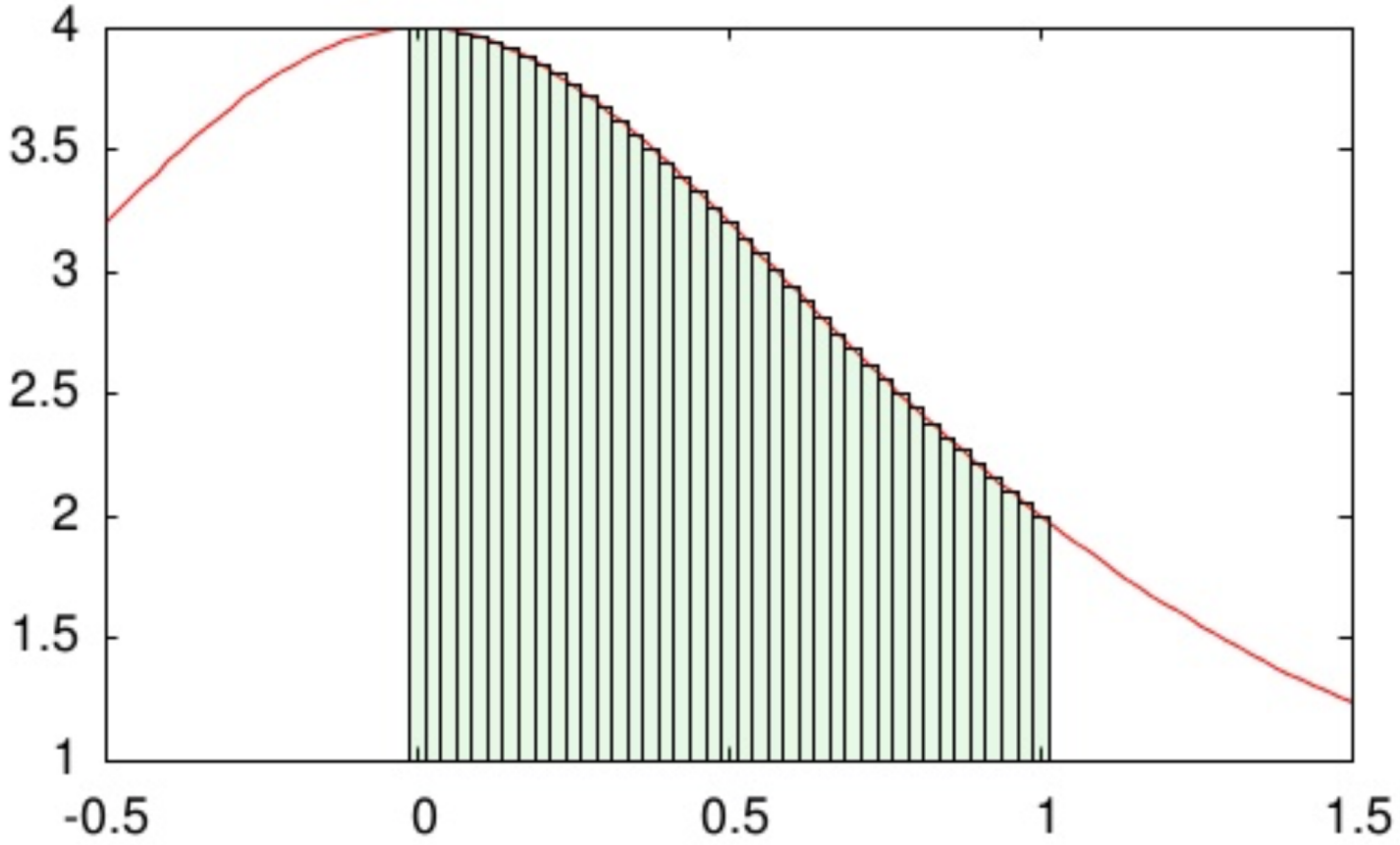
- OpenMP: set of compiler directives, library routines, and
- environment variables can be used to specify high-level parallelism
- Supports programs written in Fortran, C, C++
- First release 1997
- **Not** to be confused with OpenMPI (message passing interface) which is based on message passing between loosely coupled computers.

https://www.openmp.org

Currently supported: Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, C11, C++11, and C++14

Example: Approximating Pi

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



41 points : $\pi \approx 3.1380$; 10^9 points : $\pi \approx 3.14159265$

From: Albert DeFusco, https://www.slideshare.net/AlbertDeFusco/integrate-47112918

Compute π in C++ and Fortran 90

```cpp
#include <iostream>
using namespace std;

double arc(double x);
const long int num_steps=1000000000;
double step;
int main()
{
  double pi, sum=0.0;
  step=1.0/(double) num_steps;

  for (int i=0;i<=num_steps;i++)
  {
    double x=(i+0.5)*step;
    sum+= arc(x);
  }
  pi = sum*step;

  cout.precision(10);
  cout << "pi is probably "
    << fixed << pi << endl;

  return 0;
}
double arc(double x)
{
  double y = 4.0/(1+x*x);
  return y;
}
```

```fortran
program integratePi

  implicit none
  integer(kind=8) :: num_steps,i
  real(kind=8) :: sum,step,pi,x

  num_steps=1000000000
  sum=0.d0
  step=1.d0/num_steps

  do i=1,num_steps
    x=(i+0.5d0)*step
    sum=sum+arc(x)
  enddo

  pi=sum*step

  write(6, &
   '("pi is probably ",f12.10)') &
    pi

  contains
    function arc(x)
      implicit none
      real(kind=8) :: arc,x
      arc = 4.d0/(1.d0+x*x)
    end function arc
end program integratePi
```

From: Albert DeFusco, https://www.slideshare.net/AlbertDeFusco/integrate-47112918

Compute π in C++ and Fortran 90 in parallel

```
1  #pragma omp parallel for reduction(+:sum)
2    for (int i=0;i<=num_steps;i++)
3    {
4      double x=(i+0.5)*step;
5      sum+= arc(x);
6    }
```

```
1    !$omp parallel do reduction(+:sum) &
2    !$omp private(x)
3    do i=1,num_steps
4      x=(i+0.5d0)*step
5      sum=sum+arc(x)
6    enddo
7    !$omp end parallel do
```

Not all C compilers have OpenMP installed!

```
gcc simple.c -std=c99 -fopenmp
```

From: Albert DeFusco, https://www.slideshare.net/AlbertDeFusco/integrate-47112918

Threading in Java

- Step 1: Implement class that derives from predefined “Runnable”
  Implement method “run”
- Step 2: Instantiate an object of class “Thread” with your new class:
  Thread t1 = new Thread(Runnable myRunnable);
  Thread t2 = new Thread(Runnable myRunnable);
- Step 3: Start threads with “start” method:
  t1.start()
  t2.start()

https://www.tutorialspoint.com/java/java_multithreading.htm

```java
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name) {
      threadName = name;
      System.out.println("Creating " +  threadName );
   }

   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
      } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
```

```java
public class TestThread {

   public static void main(String args[]) {
      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

Output

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Threading in R

- Several R packages for implementing thread-parallelism (historically packages “multicore”, “snow”) were integrated into R package “parallel”
- Package may utilize different cores, graceful fallback if not available

http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html

https://www.r-bloggers.com/how-to-go-parallel-in-r-basics-tips/

Serial Execution in R – for loop

```
my_square <- function(x) { x * x }

v <- c(1,2,3) # input vector
result <- list() # empty list

for (i in 1:length(v)) {
 result[[i]] <- my_square(v[i])
}
print(result)
```

Output:

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

[[2]]
[1] 4

[[3]]
[1] 9
```

R: Using **lapply** and **mclapply**

- R “apply” functions are much faster compared to for-loops.
- R “lapply” (list-apply) function easy to parallelize by instead calling “mclapply” from the R package “parallel”.

```
my_square <- function(x) { x * x }

v <- c(1,2,3) # input vector

cat("serial: apply function to each element of vector:\n")
lapply(v, FUN=my_square)

library(parallel)

cat("potentially run on different cores:\n")
mclapply(v, FUN=my_square)
```

Output of R Example:

```
serial: apply function to each element of vector:
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

potentially run on different cores:
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9
```

- Basic optimization before you jump to parallelization
  - avoid needless copying of objects
  - R: prefer “apply” over ”for” loops
- Slow serial code produces slow parallel code
- Start with simple paradigms. Complexity is not a goal
  - command line parameters that will run your program on parts of the data facilitates cluster usage

C++: Intel Threading Building Blocks

- Platform-independent coding of multi-core processing
- Provides parallel versions of basic algorithm: **parallel_for**, **parallel_sort** etc
- Provides thread-save containers:
  - array (**concurrent_vector**)
  - set (**concurrent_set**)
  - hash map (**concurrent_hashmap**)

  etc.
- Key idea: implement class that performs number crunching via its "()" operator to which a data structure indicating the index range is passed
- Alternative: Parallel Patterns Library (PPL) from Microsoft

```cpp
#include <vector>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

struct IncrementElements
{
    IncrementElements(std::vector<double>* dataPtr) :
        m_data{dataPtr}
    {
    }

    void operator()(const tbb::blocked_range<size_t>& range) const
    {
        for (size_t i = range.begin(); i < range.end(); ++i)
            (*m_data)[i] += 1;
    }

private:
    std::vector<double>* m_data;
};

int main()
{
    std::vector<double> data(10000, 0);
    for (size_t t = 0; t < 5; ++t)
        tbb::parallel_for(tbb::blocked_range<size_t>(0, data.size()), IncrementElements(&data));
}
```

https://texus.me/2015/11/17/intel-tbb-parallel-for/

```cpp
#include <vector>
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

struct IncrementElements
{
    IncrementElements(std::vector<double>* dataPtr) :
        m_data{dataPtr}
    {
    }

    void operator()(const tbb::blocked_range<size_t>& range) const
    {
        for (size_t i = range.begin(); i < range.end(); ++i)
            (*m_data)[i] += 1;
    }

private:
    std::vector<double>* m_data;
};

int main()
{
    std::vector<double> data(10000, 0);
    for (size_t t = 0; t < 5; ++t)
        tbb::parallel_for(tbb::blocked_range<size_t>(0, data.size()), IncrementElements(&data));
}
```

Careful: class “vector” is not thread-safe. OK here, because threads operate on non-overlapping indices. Otherwise use concurrent_vector

Intel Threading Building Blocks - Requirements

- Download source code from https://www.threadingbuildingblocks.org
- Follow installation instructions. Can be as simple as uncompressing directory. Maybe define variable TBB_ROOT as installation directory
- Provide compiler with path name to include files
  (like –I $TBB_ROOT/include)
- Provide compiler with name and path to linked compiled code:
  -l tbb -L $TBB_ROOT/lib
- Provide compiled binary with path to linked library:
  export LD_LIBRARY_PATH=$TBB_ROOT/lib
- Now your compiled program may be ready to run

Support for Multithreading in C++/11

```cpp
#include <iostream>
#include <thread>

static const int num_threads = 10;

//This function will be called from a thread

void call_from_thread(int tid) {
    std::cout << "Launched by thread " << tid << std::endl;
}

int main() {
    std::thread t[num_threads];

    //Launch a group of threads
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }

    std::cout << "Launched from the main\n";

    //Join the threads with the main thread
    for (int i = 0; i < num_threads; ++i) {
        t[i].join();
    }

    return 0;
}
```

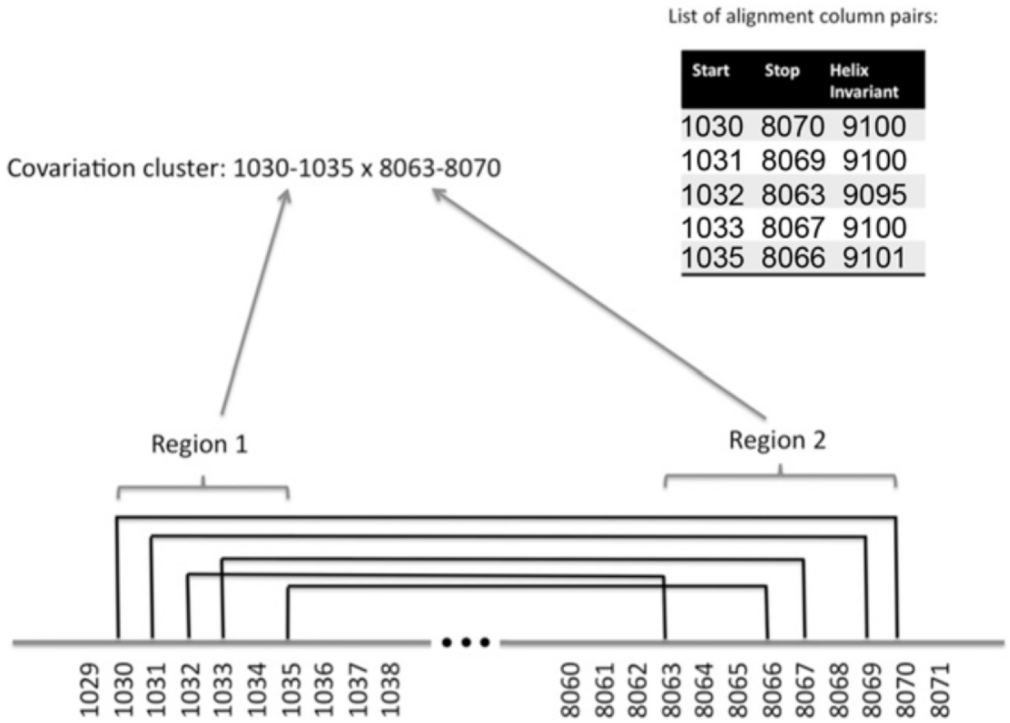
Output:

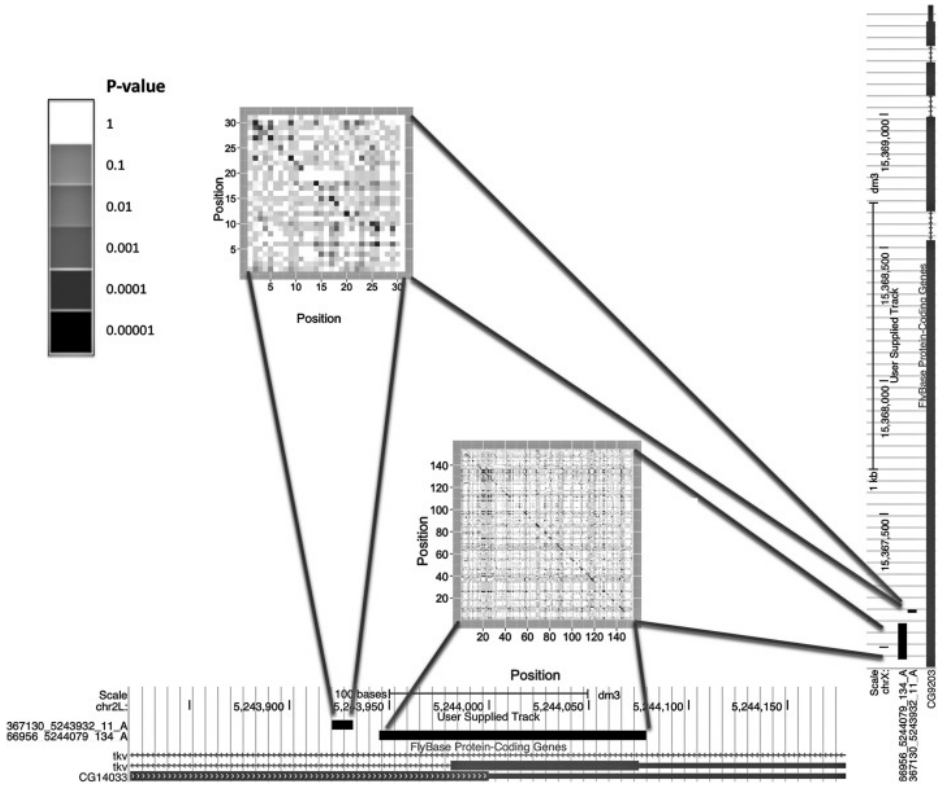
```
Sol$ ./a.out
Launched by thread 0
Launched by thread 1
Launched by thread 2
Launched from the main
Launched by thread 3
Launched by thread 5
Launched by thread 6
Launched by thread 7
Launched by thread Launched by thread 4
8L
aunched by thread 9
Sol$
```

However, standard containers (like vector, set, etc) of C++11 have limited thread-safety
see https://en.cppreference.com/w/cpp/container

Use case: Genome Scanning using Threading

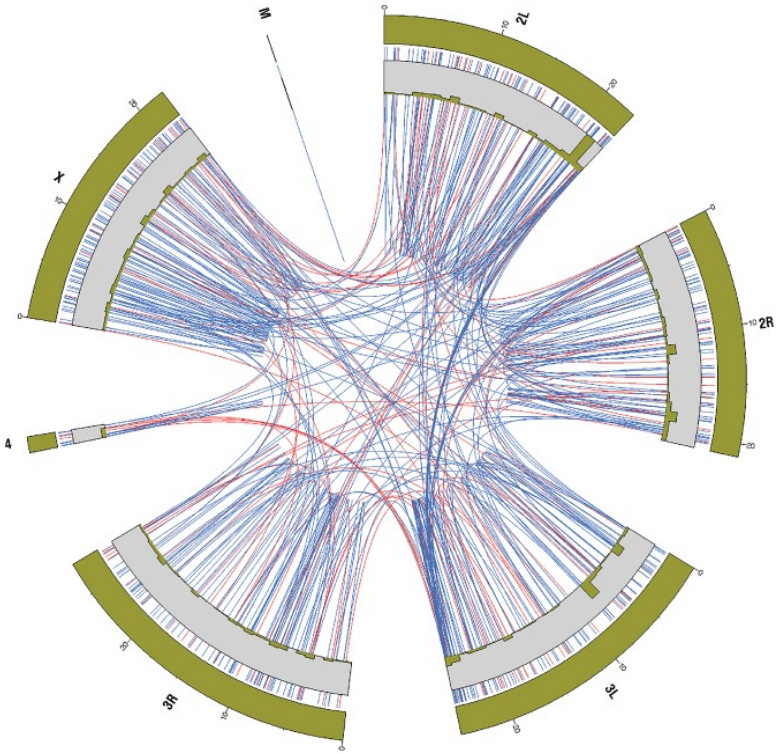
- Genome-wide all-versus-all search of correlated mutations (fruit fly)
- Computationally highly demanding, novel algorithm





Why Threading was Important for this Project:ß

- Memory-intensive: several gigabyte input data
- Simple running of program with multiple parameters would have only one compute job per node due to large memory. Many cores would be unused.
- Better alternative: load input data once into memory, have different threads scan multiple regions of the same input data in parallel
- Pooled result required thread-safe data structure
- Use of Intel Threading Building Blocks (TBB)

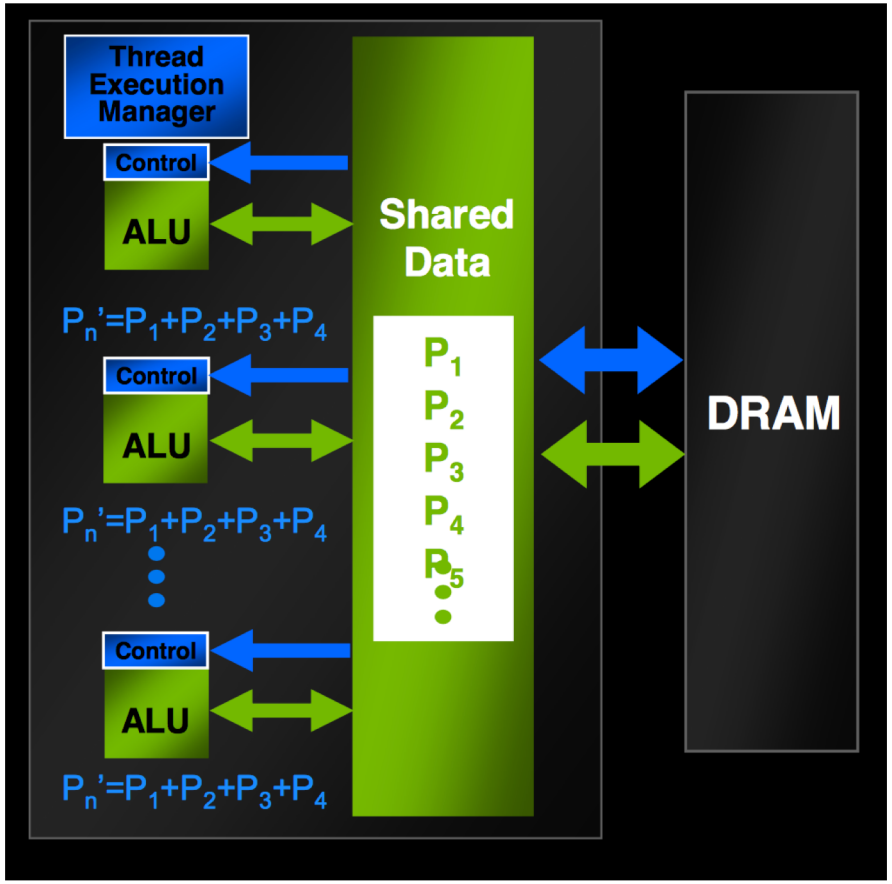


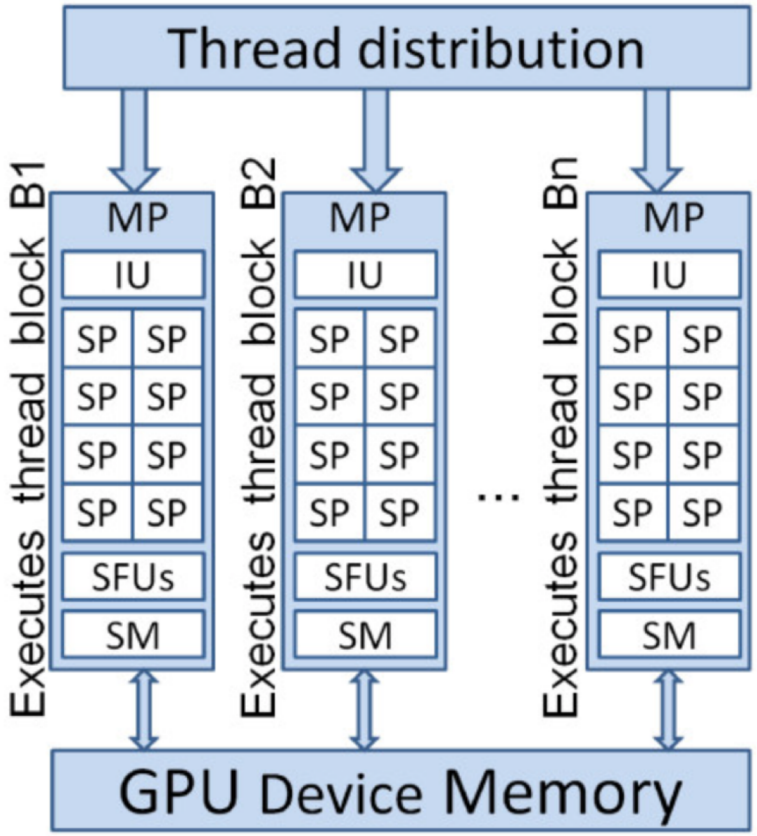
Bindewald and Shapiro, RNA, 2012

Graphical Processing Unit (GPU)

- Specialized chips for accelerating graphics have been available since the 1980’s.
- Early tasks: drawing lines, filling polygons, later texture mapping
- NVIDIA developed GPUs for general purpose programming
- C library CUDA: Compute Unified Device Architecture

GPU Architecture (Simplified)





MP = Multi Processor, SM = Shared Memory
SFU = Special Functions Unit
IU = Instruction Unit, SP = Streaming processor (core).

http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf

Chikkagoudar et al, BMC Research Notes 4(1):158 · May 2011

GPU Programming With CUDA: Example

```cpp
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
```

```cpp
  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

https://devblogs.nvidia.com/even-easier-introduction-cuda/

R package “gpuR”: GPU Programming ”for All”

- Performs matrix operations on GPU if available
- Straightforward: code is similar to regular R programming
- Simple creation of vector or matrix objects that persist on GPU

GPU Programming Example in R with “gpuR”:

Preliminaries (within R):
>install.packages(“gpuR”)
>library(gpuR)
>dectectGPUs()

```
library(gpuR)
# vclMatrix: create object that persists on GPU if available
a = vclMatrix(1:16, nrow = 4, ncol=4, type='float')
b = vclMatrix(2, nrow = 4, ncol=4, type='float') # 4x4 matrix filled with "2"
result <- a * b  # element-wise multiplication, result still on GPU
cat("Info on result (still on GPU):\n")
print(result) # will only print GPU device info, because still on GPU
cat("Result (transferred back from GPU to regular RAM):\n")
result2 <- as.matrix(result) # create object in regular RAM
print(result2) # print actual result
```

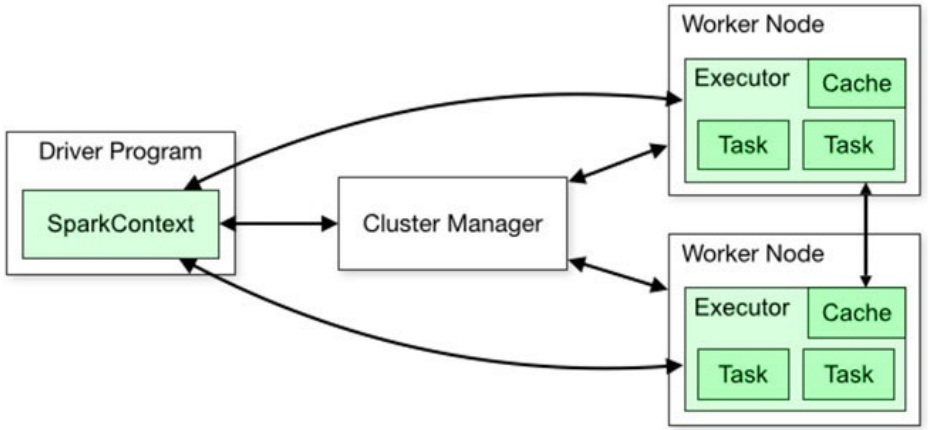
Result of GPU Example:

```
Info on result (still on GPU):
An object of class "fvclMatrix"
Slot "address":
<pointer: 0x7fdc7c48c640>

Slot ".context_index":
[1] 1

Slot ".platform_index":
[1] 1

Slot ".platform":
[1] "Apple"

Slot ".device_index":
[1] 1

Slot ".device":
[1] "Iris Pro"

Result (transferred back from GPU to regular RAM):
     [,1] [,2] [,3] [,4]
[1,]    2   10   18   26
[2,]    4   12   20   28
[3,]    6   14   22   30
[4,]    8   16   24   32
```

It detected graphics card of my laptop!

“Big Data” using Apache Spark

- Spark is a cluster-computing framework
- Datasets may be so large that they do not fit into the memory of one single node (“resilient distributed dataset”, RDD)
- Data can be operated on (with restrictions) as if using single computer
- Data sets are immutable, but functions can be applied to them (“map”)
- Summary results can be generated with “accumulators” (“reduce” step)



Toy dataset (text file containing 5 lines)

```
chr7    127471196       127472363       Pos1    0       +       127471196       127472363       255,0,0 3       300,200,100     0,500,1067
chr7    127472363       127473530       Pos2    2       +       127472363       127473530       255,0,0 2       250,500 0,667
chr7    127473530       127474697       Neg1    0       -       127473530       127474697       255,0,0 1       1167    0
chr9    127474697       127475864       Pos3    5       +       127474697       127475864       255,0,0 1       1167    0
chr9    127475864       127477031       Neg2    5       -       127475864       127477031       0,0,255 1       1167    0
```

Spark Example Session

Start Spark cluster (biowulf):

module load spark
spark start -t 120 3 # launch cluster (3 nodes, time limit 120 minutes)

spark list –d # obtain details about launched Spark clusters

Launch Python interactive session:

```
pyspark --master spark://cn0600:7077 --executor-memory=40g
```

```
>>> txt = spark.sparkContext.textFile("test.bed")
>>> txt.count()
5
>>> chr7 = txt.filter(lambda l: l.startswith('chr7'))
>>> chr7.count()
3
>>> txt.map(lambda l: len(l)).reduce(lambda a, b: a if (a>b) else b)
86
```

https://github.com/NIH-HPC/spark-slurm

General Guidelines

- Before parallelizing code, use common good practices:
  - appropriate algorithm
  - avoid unneeded copying of data structures
  - avoid recomputing of results
  - vectorizing (especially important for functional languages likes R)
- Use of compute cluster with non-parallel program
- Parallelization: start with ”high-level” approaches, going to lower levels if needed
- Beware of tradeoffs: execution speed, development time, hardware and software dependencies