Computational Optimization
ISE407

Lecture 13

Dr. Ted Ralphs

Reading for This Lecture

- Roosta, Chapter 4, Sections 1 and 3, Chapter 5
- "Introduction to High Performance Computing for Scientists and Engineers," G. Hager and G. Wellein, Chapters 5–11.
- MPI Introduction and Specification
- OpenMP Introduction, Specification, and Tutorial
- `https://juliafolds.github.io/data-parallelism/tutorials/quick-introduction/`
- `https://www.csd.uwo.ca/~mmorenom/cs2101a_moreno/Parallel_computing_with_Julia.pdf`

Design Issues

- Platform/Architecture
- Task Decomposition
- Task Mapping/Scheduling
- Communication Protocol

Parallelizing Sequential Algorithms

- The most obvious approach to developing a parallel algorithm is to parallelize a sequential algorithm.
- The primary additional concept one must keep in mind is data access patterns.
  - In the case of shared memory architectures, one must be cognizant of possible collisions in accessing the main memory.
  - In the case of distributed memory architectures, one must be cognizant of the need to move data to where it is needed.
- In either case, losses in efficiency result from either idle time or wasted computation due to lack of availability of data locally.

Task Decomposition

- Fine-grained parallelism
  - Suited for massively parallel systems with many small processors and fast communication links.
  - These are the algorithms we've primarily talked about so far.
- Course-grained parallelism
  - Suited to small numbers of more powerful processors.
  - Data decomposition
    - Recursion/Divide and Conquer
    - Domain Decomposition
  - Functional parallelism
    - Data Dependency Analysis
    - Pipelining

Task Agglomeration

- Depending on the number of processors available, we may have to run multiple tasks on a single processor.
- To do this effectively, we have to determine which tasks should be combined to achieve maximum efficiency.
- This requires the same analysis of communication patterns and data access done in task decomposition.

Mapping

- Concurrency
  - Data dependency analysis
- Locality
  - Interconnection network
  - Communication pattern
- Mapping is an optimization problem.
- These are very difficult to solve in general.

Paradigms for Parallel Programming

- It is difficult to define what we mean by a “paradigm” for parallel programming.
- There are numerous dimensions on which developing parallel algorithms may differ on different platforms.
  - Shared versus distributed memory
  - Processes versus threads
  - Asynchronous versus synchronous
  - Explicit message-passing versus remote function calls
- We will discuss some of the commonly used tools and the associated challenges.
- We’ll also see the abstractions used in Julia.
- This is only scratching the surface of this very broad and complex topic.

Data Movement

- At the core of what changes when one goes from a sequential environment to a parallel one is *data movement* and *communication*.
- Generally speaking, data movement and/or communication happens either through a shared global memory or over a network.
  - When computation is happening in different *threads* of the same process, communication can happen through memory.
  - When computation is happening in separate *processes*, perhaps on different physical compute nodes, data must be moved over the network.
- In the former case, one must take into account many additional details to ensure that *race conditions* don't arise.
- How these differed ways of moving data are reflected in the programming environment differs widely by language.
- In Julia, many of the details are abstracted away.
- There are of course many ways to hybridize.

Communication Protocols: Message Passing

- Used primarily in distributed-memory or “hybrid” environments.
- Data is passed through explicit send and receive function calls.
- There is no explicit synchronization.
- In general, this is the most flexible and portable protocol.
- MPI is the established standard.
- PVM is a similar older standard that is still used.

Communication Protocols: Open MP/Threads

- Used in shared-memory environments.
- Parallelism through "threading".
- Threads communicate through global memory.
- Can have explicit synchronization.
- OpenMP is a standard implemented by most compilers.

MPI Basics

- MPI stands for *Message Passing Interface*.
- It is an API for point-to-point communication that hides the platform-dependent details from the user.
- There many different implementations of MPI and the standard leaves some details unspecified.
- The user launches the MPI processes in a distributed fashion and forms one or more “communicators.”
- Data can be sent explicitly between processes using message-passing calls.
- Allows for portability across different platforms.

Building and Running

- There is only one single executable that is run everywhere.
- It must figure out what it's job is by querying its rank.
- MPI programs are typically built with a compiler that is really a wrapper around a standard compiler (called something `mpicc`).
- The program is launched with the `mpirun` command.

```
~> mpirun -np 8 -hostfile my_machines my_executable
```

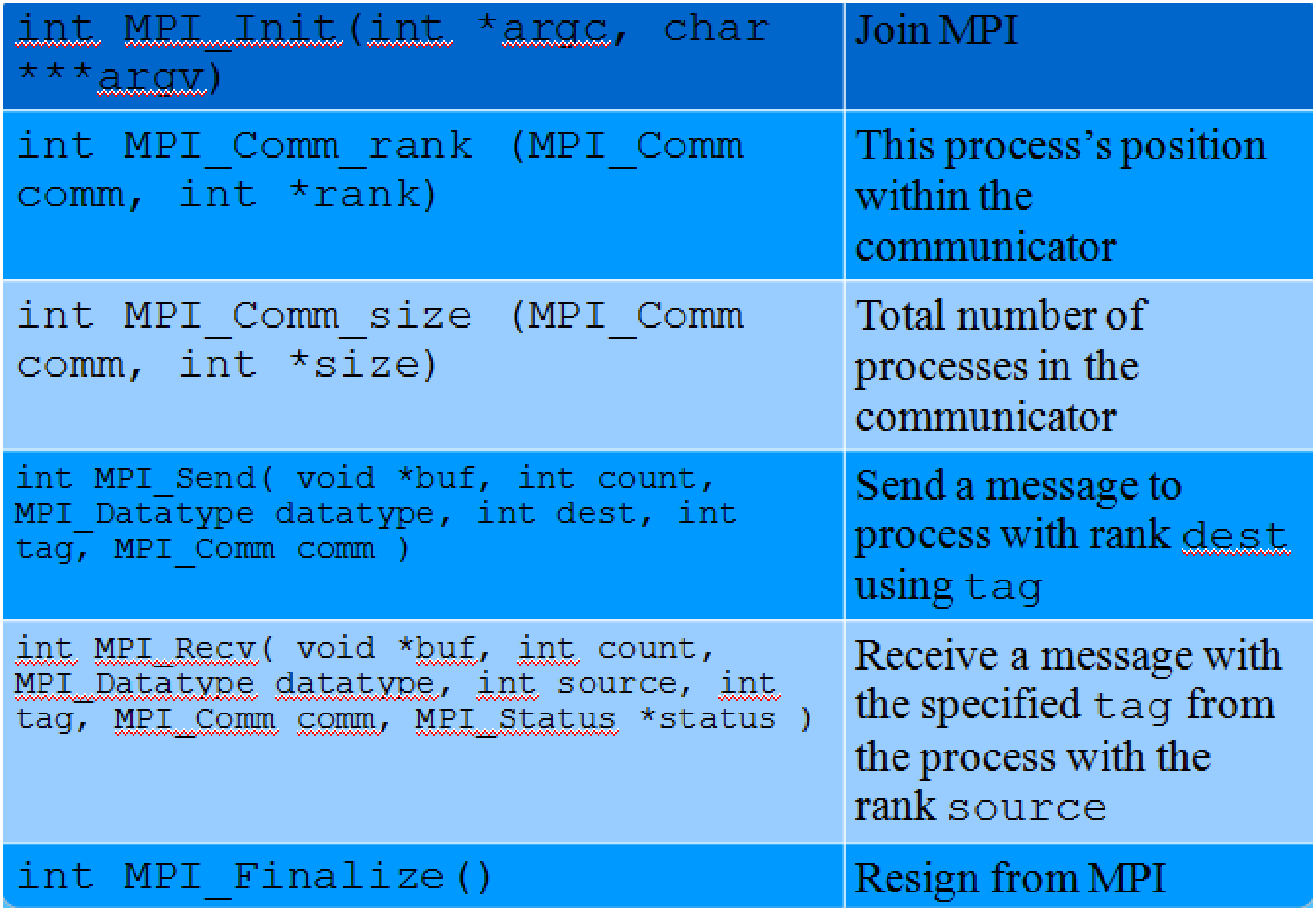
Messaging Concepts

- Buffer
- Source
- Destination
- Tag
- Communicator

Types of Communication Calls

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive
- Buffered send
- Combined send/receive
- “Ready” send

Basic Functions in MPI



`int MPI_Init(int *argc, char ***argv)`	Join MPI
`int MPI_Comm_rank (MPI_Comm comm, int *rank)`	This process's position within the communicator
`int MPI_Comm_size (MPI_Comm comm, int *size)`	Total number of processes in the communicator
`int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`	Send a message to process with rank `dest` using `tag`
`int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status )`	Receive a message with the specified `tag` from the process with the rank `source`
`int MPI_Finalize()`	Resign from MPI

Simple Example

```c
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);
}
```

Simple Example

```
#include <mpi.h>

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int i, sum(0), dest, rc, count, tag=1;

    MPI_Status Stat;

    int skip = 1000000000/size;
    int beg = rank*skip;
    int end = (rank+1)*skip;
    for (i = beg; i < end; i++){
       sum += 1;
    }
    if (rank == 0){
       int sum_other(0), total (0);
       while (total < size-1){
          rc = MPI_Recv(&sum_other, 1, MPI_INT, MPI_ANY_SOURCE, tag,
                        MPI_COMM_WORLD,
                        &Stat);
          std::cout << "Message received from " << Stat.MPI_SOURCE << ": ";
          std::cout << sum << std::endl;
          sum += sum_other;
          total++;
       }
       std::cout << "Total: " << sum << std::endl;
    }else{
          rc = MPI_Send(&sum, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
          std::cout << "Message sent from " << rank << ": " << sum;
          std::cout << std::endl;
       }

    MPI_Finalize();
}
```

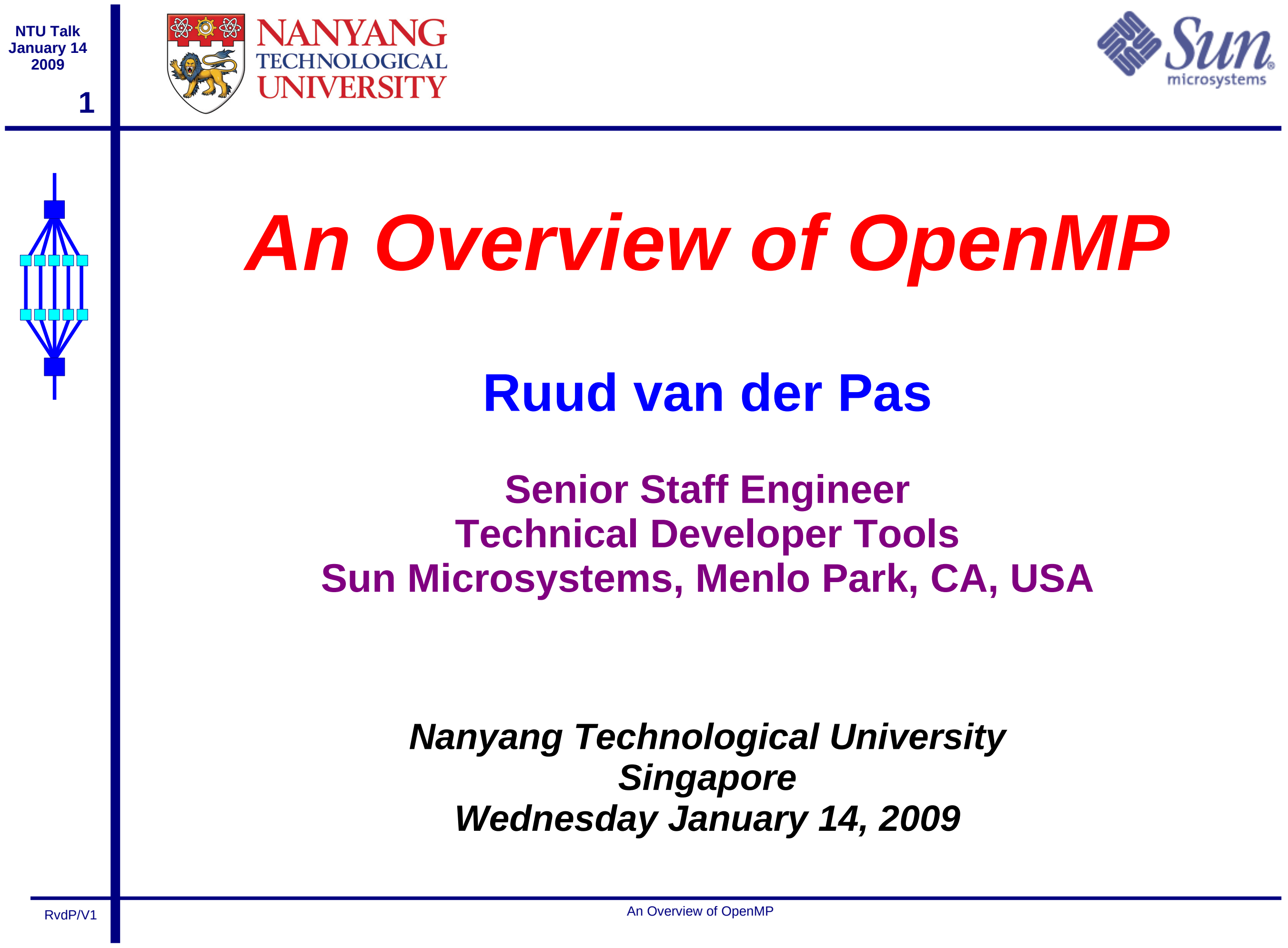
Collective Communication

- Synchronization: processes wait until all members of the group have reached the synchronization point.
- Data Movement: broadcast, scatter/gather, all to all.
- Collective Computation (reductions): one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.).

Virtual Topologies

- Allows the user to specify the topology of the interconnection network.
- This may allow certain features to be implemented more efficiently.

OpenMP/Threads



NTU Talk
January 14
2009

1

NANYANG
TECHNOLOGICAL
UNIVERSITY

Sun microsystems

An Overview of OpenMP

Ruud van der Pas

Senior Staff Engineer
Technical Developer Tools
Sun Microsystems, Menlo Park, CA, USA

Nanyang Technological University
Singapore
Wednesday January 14, 2009

RvdP/V1

An Overview of OpenMP

OpenMP Implementation

- OpenMP is implemented through compiler directives.
- User is responsible for indicating what code segments should be performed in parallel.
- The user is also responsible for eliminating potential memory conflicts, etc.
- The compiler is responsible for inserting platform-specific function calls, etc.

OpenMP Features

- Capabilities are dependent on the compiler.
  - Primarily used on shared-memory architectures
  - Can work in distributed-memory environments (TreadMarks)
- Explicit synchronization
- Locking functions
- Critical regions
- Private and shared variables
- Atomic operations

Using OpenMP

- Compiler directives
  - parallel
  - parallel for
  - parallel sections
  - barrier
  - private
  - critical
- Shared library functions
  - `omp_get_num_threads()`
  - `omp_set_lock()`

OpenMP Example

```
int matvecmult(int **A, int *x, int * b, int m, int n){

   #pragma omp parallel for default(none) private(i,j,sum) \
                                          shared(m,n,A,x,b)
   for (i=0; i<m; i++){
      sum = 0.0;
      for (j=0; j<n; j++){
         sum += A[i][j]*x[j];
      }
      b[i] = sum;
   }
}
```

OpenMP Performance

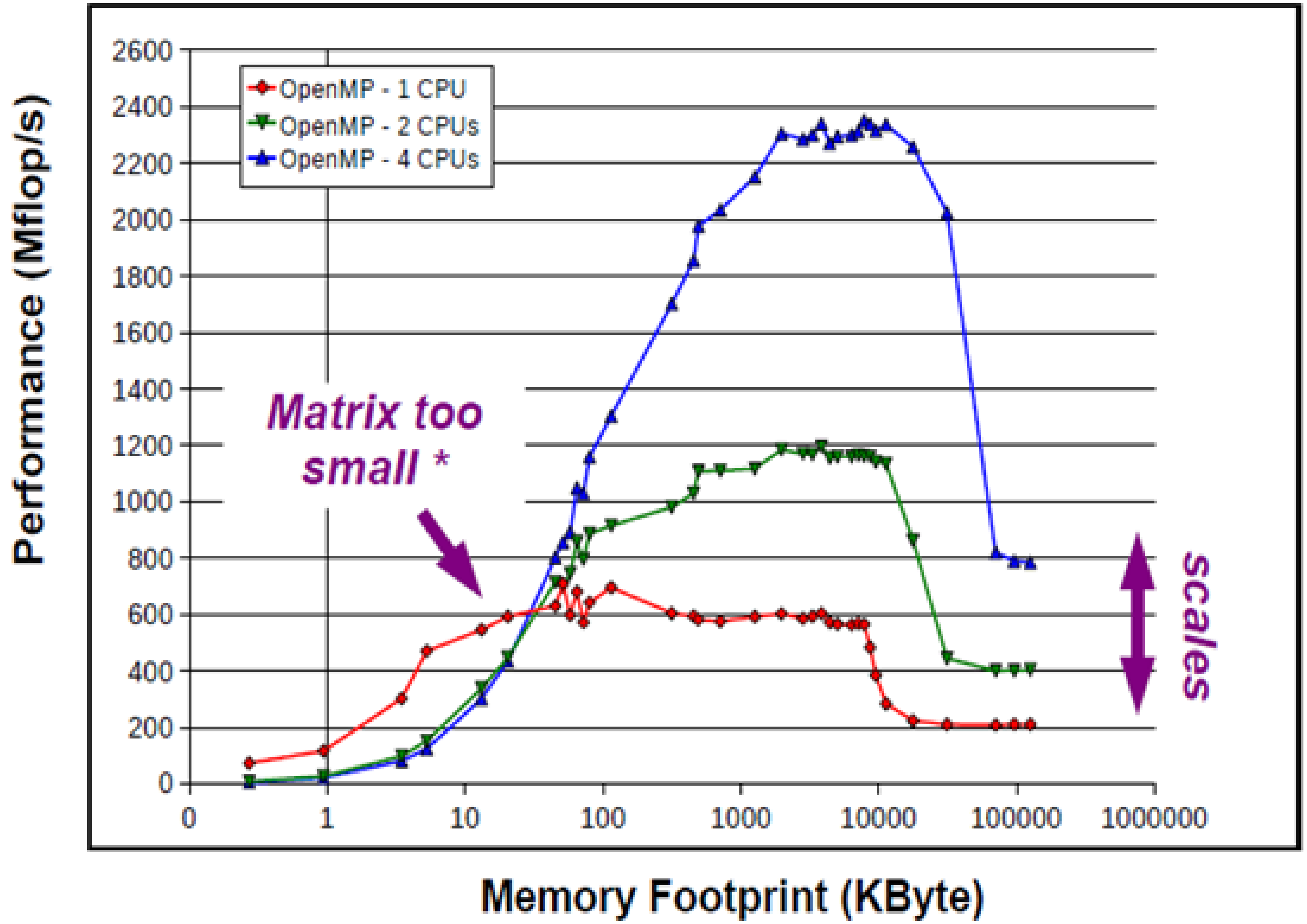


Figure 1: Open MP Performance

OpenMP Concepts and Issues

- Race Conditions (conflicts between processes in updating data)
- Deadlocks
- Critical regions
- Locking functions
- Atomic updates

Atomic Update Example

The advantage of `atomic` over `critical` is that it allows us to update different parts of an array in parallel.

```
void atomic_example(int *x, int *y, int *index, int n)
{
#pragma omp parallel for shared(x, y, index, n)
   for (int i = 0; i < n; i++) {
 #pragma omp atomic update
      x[index[i]] += i;
      y[i] += work2(i);
   }
}

int main()
{
 int x[1000], y[10000], index[10000], i;
 for (i = 0; i < 10000; i++) {
    index[i] = i % 1000;
    y[i]=0
 }
 for (i = 0; i < 1000; i++)
    x[i] = 0;
 atomic_example(x, y, index, 10000);
 return 0;
}
```

Source:
https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf

Parallel Programming in Julia

- Several different paradigms are possible, but abstractions bind them together.
  - Aysnchronous programming
  - Producer-consumer
  - Multi-threading
  - Distributed computation
- One can start Julia with multiple threads and/or multiple processes/
- The number of threads cannot be changed dynamically, but processes can be added and removed.

```
~> julia -t 4
julia> Threads.nthreads()
4
~> julia -p 4
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
```

Asynchronous Programming (Tasks)

- One can create and schedule independent tasks with `@task` and `@async`
- Using `@task` creates a task, but does not run it.
- `schedule` is used to run the task.
- One can also explicitly `wait` for a task to finish.
- `@async` creates a task and immediately runs it.

```
julia> t = @task sum(rand(100))
Task (runnable) @0x00007f13a40c0eb0
julia> schedule(t);
julia> wait(t);
julia> fetch(t)
49.803227895281665
```

Communicating with Channels

Channels allow tasks to communicate while running.

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:4
               put!(c, 2n)
           end
           put!(c, "stop")
       end;
julia> chnl = Channel(producer);
julia> take!(chnl)
"start"
julia> take!(chnl)
2
julia> take!(chnl)
4
julia> take!(chnl)
6
julia> take!(chnl)
8
julia> take!(chnl)
"stop"
```

Multithreading

- The `@threads` macro can be used to parallelize loops.
- You are explicitly responsible for avoiding race conditions by using locks and atomic variables.

```julia
using Base.Threads
function matmult_naive_parallel!(C, A, B)
    @threads for i ∈ 1:size(A, 1)
        for j ∈ 1:size(B, 2)
            C[i, j] = A[i, :]'B[:, j]
        end
    end
    return(C)
end
```

Locks

- Locking can be used to prevent data conflicts.
- Julia offers two kinds of locks: `SpinLock()` and `ReentrantLock()`.
- The latter should be used in general, especially when the task may need to invoke the lock multiple times.

```
julia> l = ReentrantLock()
julia> lock(l) do
           foo()
       end

julia> if !islocked(l)
          lock(l)
          foo()
          unlock(l)
       end
```

Atomic Variables

- It is also possible to create variables that can only be accessed by one thread at a time.

```
julia> let x = 0
       @threads for i in 1:1000
          x += 1
       end
       println(x)
       end
828
julia> let x = Atomic{Int}(0)
       @threads for i in 1:1000
          atomic_add!(x ,1)
       end
       println(x[])
       end
1000
```

- Note that there is potential inefficiency being introduced here.
- This may not the right way to do this computation in practice.

Threaded Mapping

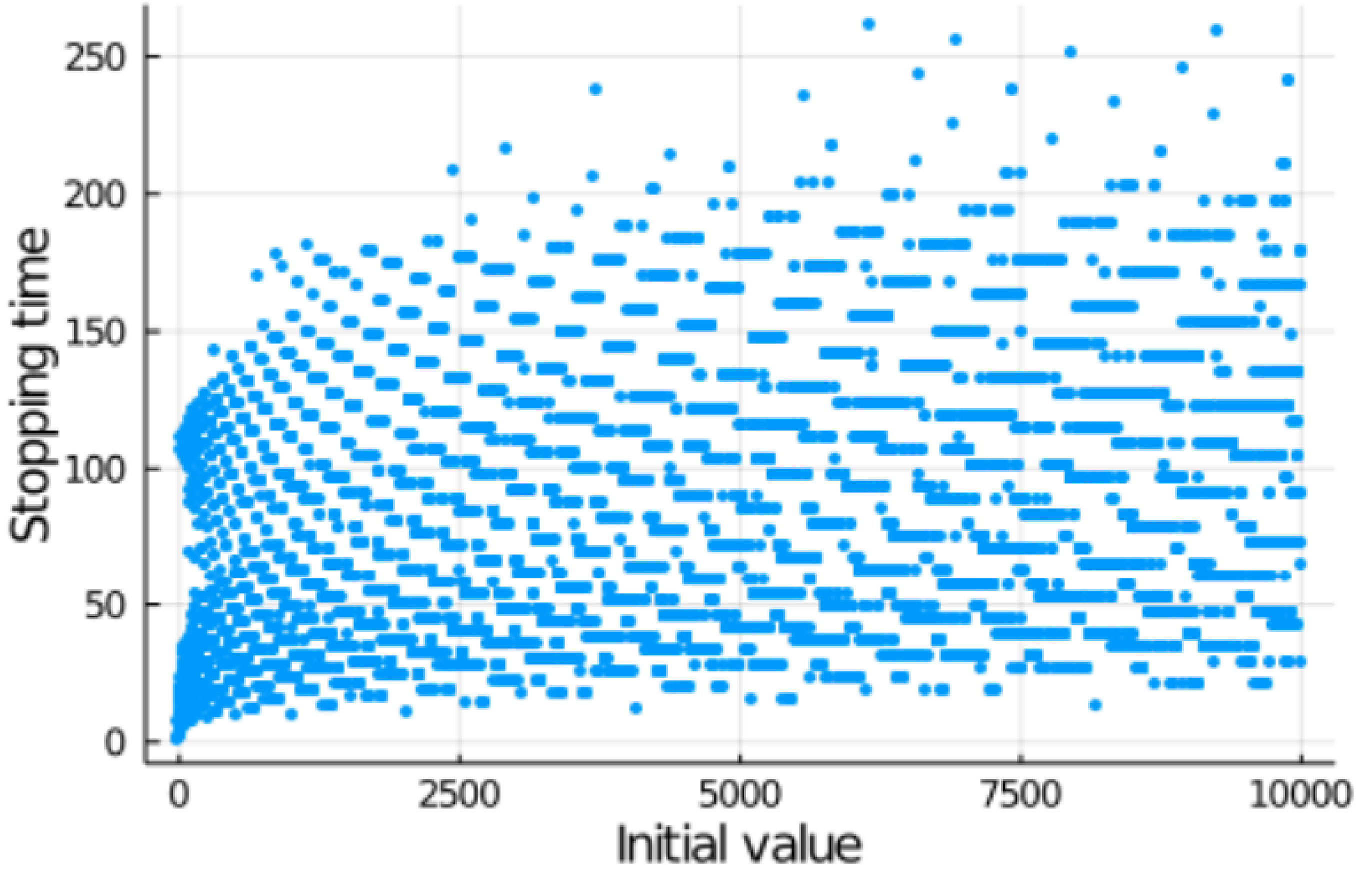
```julia
function collatz(x)
    if iseven(x)
        x ÷ 2
    else
        3x + 1
    end
end

function collatz_stopping_time(x)
    n = 0
    while true
        x == 1 && return n
        n += 1
        x = collatz(x)
    end
end

plt = scatter(
    ThreadsX.map(collatz_stopping_time, 1:10_000),
    xlabel = "Initial value",
    ylabel = "Stopping time",
    label = "",
    markercolor = 1,
    markerstrokecolor = 1,
    markersize = 3,
    size = (450, 300),
)
```

Source: `https://juliafolds.github.io/data-parallelism/tutorials/quick-introduction/`

Collatz Plot



Distributed Computing

- Julia can add and remove processes dynamically.
- These processes can be running locally or remotely.
- The message-passing is implicit and implemented through remote references and remote procedure calls.

```
julia> addprocs(3)
julia> workers()
3-element Array{Int64,1}:
 1
 2
 3
julia> workers()
julia> addprocs([("polyps.ie.lehigh.edu", 1)])
```

- Note that the above requires that you have passwordless ssh access set up and also that the directory structure is the same on the two machines.

Remote References

In distributed mode, you ask for a function to be run on a remote process/machine and the result returned to the local process.

```
julia> s = @spawnat 3 sum(rand(1000))
Future(3, 1, 5, nothing)
julia> fetch(s)
506.92191123610934
julia> s = @spawnat 3 sleep(10)
       @elapsed wait(s)
10.0018157
julia> @elapsed s = remotecall_wait(sleep, 3, 10)
10.2019344
```

Defining Functions Remotely

- There is some subtlety around ensuring that variables and functions are defined on remote processes that you need to take care of.
- The `@everywhere` macro can be used to define a function within all processes.

```
julia> @everywhere function fib(n)
           if n < 2
               return n
           else
               return fib(n-1) + fib(n-2)
           end
       end
julia> @elapsed begin
           for i in 1:4
               t[i] = fib(45)
           end
       end
19.8323551
julia> @elapsed begin
          for i in workers()
             t[i] = @spawnat i fib(45)
          end
          for i in workers()
             wait(t[i])
          end
       end
7.4685307
```

Parallel Fibonacci

```
julia> @everywhere function fib_parallel(n)
           if n < 40
               return fib(n)
           else
               x = @spawn fib_parallel(n-1)
               y = fib_parallel(n-2)
               return fetch(x) + y
           end
       end

julia> @elapsed fib_parallel(45)
2.6285944
```

Shared Arrays

- In distributed mode, each process gets its own copy of variables.
- The following will not work as one might expect.

```
a = zeros(100000)
@distributed for i = 1:100000
    a[i] = i
end
```

- Shared arrays are made for this purpose.
- Note that there is implicit data maovement happening, though.

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

Other Constructs

```
nheads = @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end

using FLoops

@floop for (x, y) in zip(1:3, 1:2:6)
    a = x + y
    b = x - y
    @reduce(s += a, t += b)
end
(s, t)
```

Comments

- Making even simple code efficient in parallel is difficult and requires attention to details.
- The built-in `@threads` and `@distributed` are a good starting point, but have limitations.
- There are many gotchas, such as the fact that there is no threaded garbage collector.
- Results may be non-intuitive.

```
julia> @btime @distributed (+) for i in 1:100000000
                 1
       end
  314.100 µs (400 allocations: 16.83 KiB)
100000000

julia> @btime let x = Threads.Atomic{Int}(0)
           Threads.@threads for i in 1:100000000
              Threads.atomic_add!(x, 1)
           end
       end
  1.982 s (52 allocations: 6.19 KiB)
```

- We have only scratched the surace here.

Libraries for Parallelism in Julia

- `Dagger.jl`
- `FLoops.jl`
- `KissThreading.jl`
- `Parallelism.jl`
- `Strided.jl`
- `TensorOperations.jl`
- `ThreadPools.jl`
- `ThreadTools.jl`
- `ThreadsX.jl`
- `Transducers.jl`
- `Tullio.jl`