WHAT IS PARGO?

- Pargo is a library for parallel programming in Go at imec’s ExaScience Lab:
  - based on our experiences with parallel programming in C++, Common Lisp, and Java
  - released under a BSD-style open source license at https://github.com/exascience/pargo
- Pargo supports numerous common parallel programming patterns:
  - Divide-and-conquer task-based parallelism
  - Parallel ranges, parallel reduction, parallel Boolean functions
  - Speculative parallelism
  - Parallel Quicksort and Mergesort
  - Parallel hash table
  - Parallel pipelines inspired by Java Parallel Streams introduced in JDK 8
    - including support for contexts, cancellation, and Go-style error handling
CONCURRENCY VS. PARALLELISM

- Concurrency is part of the problem domain.
- Needs solution even without multicore/node.
- Go is really good at this!

- Parallelism is part of the solution domain.
- Only needed for performance.
- Pargo is really good at this! ;)

https://xkcd.com/726/

PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    var sum float64
    for _, number := range numbers {
        sum += number
    }
    return sum
}
```
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2
    var wg sync.WaitGroup
    wg.Add(1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()
    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
func sum(numbers []float64) float64 {
    var wg sync.WaitGroup
    half := len(numbers) / 2
    right := sum(numbers[half:])
    left = sum(numbers[:half])
    return left + right
}

if sum(numbers) < threshold {
    var number := range numbers
    sum += number
}
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2
    var wg sync.WaitGroup
    wg.Add(delta: 1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()
    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2

    var wg sync.WaitGroup
    wg.Add(delta: 1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()

    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2
    var wg sync.WaitGroup
    wg.Add(delta: 1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()
    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2
    var wg sync.WaitGroup
    wg.Add(delta: 1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()
    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM
WITH 16 CORES
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM
WITH 4 CORES
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM
WITH LOAD IMBALANCE
DIVIDE-AND-CONQUER TASK-BASED PARALLELISM

- Task-based parallelism allows flexible distribution of work over CPU cores.
- Distributing work evenly over cores is often not optimal, because of load imbalance.
Task-based parallelism allows flexible distribution of work over CPU cores.

…but how are tasks *actually* scheduled over the cores?
• Work stealing is known to be optimal both in theory and practice

• Successfully implemented in many languages and libraries:
  ▪ Cilk for C; Threading Building Blocks for C++; Java fork/join; …
  ▪ …and Go

Wonder Gopher by Ashley McNamara, https://github.com/ashleymcnamara/gophers
WORK STEALING FINDS OPTIMAL DISTRIBUTION ON THE FLY
PARALLEL PROGRAMMING: AN EXAMPLE

```go
func sum(numbers []float64) float64 {
    if len(numbers) < threshold {
        var sum float64
        for _, number := range numbers {
            sum += number
        }
        return sum
    }
    half := len(numbers) / 2
    var wg sync.WaitGroup
    wg.Add(1)
    var left float64
    go func() {
        defer wg.Done()
        left = sum(numbers[:half])
    }()
    right := sum(numbers[half:])
    wg.Wait()
    return left + right
}
```
## The Example Program in PARGO

```go
import "github.com/exascience/pargo/parallel"

func sum(numbers []float64) float64 {
    return parallel.RangeReduceFloat64(low: 0, len(numbers), n: 0,
    func (low, high int) float64 {
        var sum float64
        for i := low; i < high; i++ {
            sum += numbers[i]
        }
        return sum
    },
    func (left, right float64) float64 {
        return left + right
    }
}
```
THE EXAMPLE PROGRAM IN PARGO

```go
import "github.com/exascience/pargo/parallel"

func sum(numbers []float64) float64 {
    return parallel.RangeReduceFloat64(low: 0, len(numbers), n: 0,
        func (low, high int) float64 {
            var sum float64
            for i := low; i < high; i++ {
                sum += numbers[i]
            }
            return sum
        },
        func (left, right float64) float64 {
            return left + right
        }
    )
}
```
THE EXAMPLE PROGRAM IN PARGO

```go
import "github.com/exascience/pargo/parallel"

func sum(numbers [ ]float64) float64 {
    return parallel.RangeReduceFloat64( low: 0, len(numbers), n: 0,
    func (low, high int) float64 {
        var sum float64
        for i := low; i < high; i++ {
            sum += numbers[i]
        }
        return sum
    },
    func (left, right float64) float64 {
        return left + right
    })}
```
THE EXAMPLE PROGRAM IN PARGO

```go
import "github.com/exascience/pargo/parallel"

func sum(numbers []float64) float64 {
    return parallel.RangeReduceFloat64(low: 0, len(numbers), n: 0,
            func (low, high int) float64 {
                var sum float64
                for i := low; i < high; i++ {
                    sum += numbers[i]
                }
                return sum
            },
        func (left, right float64) float64 {
            return left + right
        },
    }
```
...AND LOTS OF OTHER PARALLEL PROGRAMMING PATTERNS

- Parallel Do
- Parallel range
- Parallel reduction over int, float64, string, interface{}
- Parallel range reduction over int, float64, string, interface{}
- Parallel And, Or, RangeAnd, RangeOr
- Speculative variants of many of the above functions
- Sequential variants for debugging
- Parallel Quicksort and merge sort
- A parallel hash table (similar to Go’s sync.Map)
- …and parallel pipelines.
CONCURRENT PIPELINES IN GO

```go
func gen(numbers []int) <- chan int {
    out := make(chan int)
    go func() {
        for _, number := range numbers {
            out <- number
        }
        close(out)
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for number := range in {
            out <- number * number
        }
        close(out)
    }()
    return out
}

func main() {
    for n := range square(gen([]int{2, 3}))) {
        fmt.Println(n)
    }
}
```
func gen(numbers []int) <- chan int {
    out := make(chan int)
    go func() {
        for _, number := range numbers {
            out <- number
        }
        close(out)
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for number := range in {
            out <- number * number
        }
        close(out)
    }()
    return out
}

func main() {
    for n := range square(gen([]int{2, 3})) {
        fmt.Println(n)
    }
}
CONCURRENT PIPELINES IN GO

```go
func gen(numbers []int) <-chan int {
    out := make(chan int)
    go func() {
        for _, number := range numbers {
            out <- number
        }
        close(out)
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for number := range in {
            out <- number * number
        }
        close(out)
    }()
    return out
}

func main() {
    for n := range square(gen([]int{2, 3})) {
        fmt.Println(n)
    }
}
```
CONCURRENT PIPELINES IN GO

```go
func gen(numbers []int) <- chan int {
    out := make(chan int)
    go func() {
        for _, number := range numbers {
            out <- number
        }
        close(out)
    }()
    return out
}

func square(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for number := range in {
            out <- number * number
        }
        close(out)
    }()
    return out
}

func main() {
    for n := range square(gen([]int{2, 3})) {
        fmt.Println(n)
    }
}
```
PARALLEL PIPELINES USING PARGO

```go
func main() {
    var p pipeline.Pipeline
    p.Source([]int{2, 3})
    p.Add(
        pipeline.Par(pipeline.Receive(func (_ int, data interface{}) interface{} { 
            numbers := data.([]int)
            for i := range numbers { 
                numbers[i] *= numbers[i]
            }
            return numbers
        })),
        pipeline.Ord(pipeline.Receive(func (_ int, data interface{}) interface{} { 
            numbers := data.([]int)
            for _, number := range numbers { 
                fmt.Println(number)
            }
            return nil
        })),
    )
    p.Run()
}
```
PARALLEL PIPELINES USING PARGO

```go
36 func main() {
37     var p pipeline.Pipeline
38     p.Source([]int{2, 3})
39     p.Add(
40         pipeline.Par(pipeline.Receive(func (_ int, data interface{}) interface{} {  
41             numbers := data.([]int)
42             for i := range numbers {
43                 numbers[i] *= numbers[i]
44             }
45             return numbers
46         })),
47         pipeline.Ord(pipeline.Receive(func (_ int, data interface{}) interface{} {  
48             numbers := data.([]int)
49             for _, number := range numbers {
50                 fmt.Println(number)
51                 return nil
52             }
53         })),
54     )
55     p.Run()
56 }
```
func main() {
    var p pipeline.Pipeline
    p.Source([]int{2, 3})
    p.Add(
        pipeline.Par(pipeline.Receive(func (_ int, data interface{}) interface{} { 
            numbers := data.([]int)
            for i := range numbers {
                numbers[i] *= numbers[i]
            }
            return numbers
        })),
    
    pipeline.Ord(pipeline.Receive(func (_ int, data interface{}) interface{} { 
            numbers := data.([]int)
            for _, number := range numbers {
                fmt.Println(number)
            }
            return nil
        })),
    
    p.Run()
}
PARALLEL PIPELINES USING PARGO

```go
func main() {
  var p pipeline.Pipeline
  p.Source([]int{2, 3})
  p.Add(
    pipeline.Par(pipeline.Receive(func (_ int, data interface{}) interface{} { 
      numbers := data.([]int)
      for i := range numbers {
        numbers[i] *= numbers[i]
      }
      return numbers
    })�,

    pipeline.Ord(pipeline.Receive(func (_ int, data interface{}) interface{} { 
      numbers := data.([]int)
      for _, number := range numbers {
        fmt.Println(number)
      }
      return nil
    })�,

  )�
  p.Run()
}
```
PARALLEL PIPELINES USING PARGO

```go
func main() {
    var p pipeline.Pipeline
    p.Source([][2]int{2, 3})
    p.Add(
        pipeline.Par(pipeline.Receive(func (_ int, data interface{}) interface{} { numbers := data.([]int)
            for i := range numbers {
                numbers[i] *= numbers[i]
            }
            return numbers
        })),
        pipeline.Ord(pipeline.Receive(func (_ int, data interface{}) interface{} { numbers := data.([]int)
            for _, number := range numbers {
                fmt.Println(number)
            }
            return nil
        })),
    )
    p.Run()
}
```
PARALLEL PIPELINES IN PARGO

- Predefined pipeline sources for arrays, slices, strings, channels, and bufio.Scanner.
- Support for user-defined sources through the pipeline.Source interface.
- Support for several kinds of nodes (stages):
  - Sequential, ordered, parallel
  - Strictly ordered, limited parallel
  - Skip and Limit nodes
- Support for several kinds of filters:
  - Generic receive and finalize
  - Boolean filters: Every, Some, NotEvery, NotAny
  - Counting filter
  - Slice filter
- Support for contexts, including cancellation
- Support for error handling, including cancellation on error
- Support for fine-tuning of batch sizes
ELPREP: A HIGH-PERFORMANCE TOOL FOR SEQUENCING

- High-performance tool for preparing SAM files for variant calling.
- Multi-threaded application that runs entirely in RAM and merges multiple steps to avoid repeated file I/O.
- Can improve performance by a factor of up to x10 compared to standard tools.
- elPrep implemented in Go since version 3.0
  - https://github.com/exascience/elprep
PARGO

- Pargo available at https://github.com/exascience/pargo
- Documentation: https://godoc.org/github.com/exascience/pargo
- More documentation: https://github.com/exascience/pargo/wiki
- elPrep: https://github.com/exascience/elprep
embracing a better life