# Relatório 2: Benchmark IS Em Golang

Kevin S. Pereira, Thomazio Giacobbe

https://github.com/thomaziogiacobbe/NPB-Golang

# Benchmark IS em Golang

- Baseado na implementação em C++ de Dalvan Griebler et al.
  - NPB-OMP; implementação em OpenMP
  - https://github.com/GMAP/NPB-CPP
- Implementação feita utilizando buckets
  - Flag USE_BUCKETS definida
- Adaptações utilizando features de multithreading do Golang
  - goroutines (criação de tarefas)
  - sync (grupos de sincronização)

# Bibliotecas

- math/rand
  - Geração de números aleatórios
- runtime
  - Informações da CPU
- strconv
  - Conversões entre strings e outros tipos

➢ go-pretty (https://github.com/jedib0t/go-pretty)
  - Biblioteca third party para formatação dos resultados em interface

# Funções common: Randlc e Vranlc

- Alterado os métodos para calcular a fórmula de gerar números aleatórios
- O código original quebrava um double em duas partes
- Agora, as operações são feitas em variáveis de ponto fixo (inteiros de 64 bits sem sinal)
- Redução no tempo de execução dentre 14% e 15%

```
func Vranlc(n int, xSeed *float64, a float64, y []float64) {
    ux := uint64(*xSeed)
    ua := uint64(a)
    const ut46 = uint64(t46)
    const it46 = 1 / t46

    for i := 0; i < n; i++ {
        ux = ux * ua % ut46
        y[i] = it46 * float64(ux)
    }
    *xSeed = float64(ux)
}

func Randlc(x *float64, a float64) float64 {
    ux := uint64(*x) * uint64(a) % uint64(t46)
    *x = float64(ux)
    const it46 = 1 / t46
    ret := it46 * (*x)
    return ret
}
```

# Funções common: ParallelFor

- Criada uma função inspirada no OpenMP para lidar com criação de tarefas de um for paralelo
- Escalonamento estático
- O parâmetro f é uma função que será executada em paralelo
- Essa função f deve receber como argumentos:
  - um id do processador
  - o índice i da iteração

```
func ParallelFor(
    n int64,
    numCPU int64,
    f func(id int64, i int64),
) {
    var group sync.WaitGroup
    group.Add(int(numCPU))
    for myid := int64(0); myid < numCPU; myid++ {
        go func(id int64) {
            for it := id; it < n; it += numCPU {
                f(id, it)
            }
            defer group.Done()
        }(myid)
    }
    group.Wait()
}
```

# Benchmark IS

- IS - Integer Sort
- Objetivo do kernel é medir o desempenho de execução e comunicação ao realizar computação em inteiros por meio de um algoritmo de ordenação em paralelo.
- A versão do código ativando a flag USE_BUCKETS implementa o *bucket sort*, um algoritmo de ordenação baseado em distribuição.

# Benchmark IS
# Rotinas principais

- CreateSeq
- Gera a sequência de chaves para serem ordenadas
- Intervalos dos índices para acesso do vetor são calculados em função do ID das CPUs. Com isso, o preenchimento é distribuído entre as CPUs.

```
func CreateSeq(
    seed float64,
    a float64,
    id int,
    group *sync.WaitGroup,
) {
    defer (*group).Done()

    myid := id
    num_procs := runtime.NumCPU()

    an := a
    mq := (num_keys + int64(num_procs) - 1) / int64(num_procs)
    k1 := mq * int64(myid)
    k2 := k1 + mq
    if k2 > num_keys {
        k2 = num_keys
    }
    s := FindMySeed(myid, num_procs, int32(4*num_keys), seed, an)
    k := max_key / 4
    for i := k1; i < k2; i++ {
        x := npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        key_array[i] = int64(float64(k) * x)
    }
}

// Main
groupCreateSec.Add(n_threads)
for i := 0; i < n_threads; i++ {
    go CreateSeq(314159265.00, 1220703125.00, i, &groupCreateSec)
}
groupCreateSec.Wait()
```

# Benchmark IS
# Rotinas principais

- CreateSeq
- Gera a sequência de chaves para serem ordenadas
- Intervalos dos índices para acesso do vetor são calculados em função do ID das CPUs. Com isso, o preenchimento é distribuído entre as CPUs.

```
func CreateSeq(
    seed float64,
    a float64,
    id int,
    group *sync.WaitGroup,
) {
    defer (*group).Done()

    myid := id
    num_procs := runtime.NumCPU()

    an := a
    mq := (num_keys + int64(num_procs) - 1) / int64(num_procs)
    k1 := mq * int64(myid)
    k2 := k1 + mq
    if k2 > num_keys {
        k2 = num_keys
    }
    s := FindMySeed(myid, num_procs, int32(4*num_keys), seed,
an)
    k := max_key / 4
    for i := k1; i < k2; i++ {
        x := npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        x += npb.Randlc(&s, an)
        key_array[i] = int64(float64(k) * x)
    }
}
```

# Benchmark IS
# Rotinas principais

- CreateSeq
- Gera a sequência de chaves para serem ordenadas
- Intervalos dos índices para acesso do vetor são calculados em função do ID das CPUs. Com isso, o preenchimento é distribuído entre as CPUs.

```
// Main
groupCreateSec.Add(n_threads)
for i := 0; i < n_threads; i++ {
    go CreateSeq(314159265.00, 1220703125.00,
                 i, &groupCreateSec)
}
groupCreateSec.Wait()
```

# Benchmark IS
# Rotinas principais

- Rank
- Realiza a ordenação de fato. Executada múltiplas vezes em chamada iterativa na main.
- Classifica as chaves manipulando os buckets de forma distribuída.

```
// Main
start := time.Now()
for iteration := 1; iteration <= MAX_ITERATIONS; iteration++ {
    if npb.Class != "S" {
        fmt.Println("\t\t", iteration)
    }
    Rank(int64(iteration))
}
tt = time.Since(start)
```

# Benchmark IS Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As instruções paralelas precisam ser definidas manualmente.
- Threads diferentes precisam ser instanciadas em código de maneira explícita.

```
#pragma omp parallel private(i, k)
{
    INT_TYPE *work_buff, m, k1, k2;

    int myid = omp_get_thread_num();
    int num_procs = omp_get_num_threads();

    work_buff = bucket_size[myid];

    /* Initialize */
    for( i=0; i<NUM_BUCKETS; i++ )
        work_buff[i] = 0;

    /* Determine the number of keys in each bucket */
    #pragma omp for schedule(static)
    for( i=0; i<NUM_KEYS; i++ )
        work_buff[key_array[i] >> shift]++;

    // ...
}
```

# Benchmark IS Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Threads diferentes precisam ser instanciadas em código de maneira explícita.

```
// Rank
bucket_size = make([][]int64, 0, num_procs)
for iter := 0; iter < num_procs; iter++ {
    temp := make([]int64, num_buckets)
    bucket_size = append(bucket_size, temp)
}

npb.ParallelFor(
    num_keys,
    int64(num_procs),
    func(myid int64, it int64) {
        bucket_size[myid][key_array[it]>>shift]++
    },
)
```

# Benchmark IS Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Não existe um meio trivial de declarar um threadprivate em Go, igual foi utilizado nas implementações originais.
- Instruções e blocos paralelos foram adaptados para utilizar matrizes compartilhadas, com distribuição no acesso pelo id.

```
INT_TYPE bucket_ptrs[NUM_BUCKETS];
#pragma omp threadprivate(bucket_ptrs)

// rank
bucket_ptrs[0] = 0;
for( k=0; k< myid; k++ )
    bucket_ptrs[0] += bucket_size[k][0];

for( i=1; i< NUM_BUCKETS; i++ ) {
    bucket_ptrs[i] = bucket_ptrs[i-1];
    for( k=0; k< myid; k++ )
        bucket_ptrs[i] += bucket_size[k][i];
    for( k=myid; k< num_procs; k++ )
        bucket_ptrs[i] += bucket_size[k][i-1];
}

/* Sort into appropriate bucket */
#pragma omp for schedule(static)
for( i=0; i<NUM_KEYS; i++ ){
    k = key_array[i];
    key_buff2[bucket_ptrs[k >> shift]++] = k;
}
```

# Benchmark IS
# Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Não existe um meio trivial de declarar um threadprivate em Go, igual foi utilizado nas implementações originais.
- Instruções e blocos paralelos foram adaptados para utilizar matrizes compartilhadas, com distribuição no acesso pelo id.

```
if (myid < num_procs-1) {
    for( i=0; i< NUM_BUCKETS; i++ )
        for( k=myid+1; k< num_procs; k++ )
            bucket_ptrs[i] += bucket_size[k][i];
}
```

# Benchmark IS
# Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Não existe um meio trivial de declarar um threadprivate em Go, igual foi utilizado nas implementações originais.
- Instruções e blocos paralelos foram adaptados para utilizar matrizes compartilhadas, com distribuição no acesso pelo id.

```
// "// <" indica linha que não existe com OpenMP

// Rank
group.Add(num_procs) // <
for myid := 0; myid < num_procs; myid++ { // <
    go func(myid int) { // <
        bucket_ptrs[myid][0] = 0
        for k := 0; k < myid; k++ {
            bucket_ptrs[myid][0] += bucket_size[k][0]
        }
        for it := int64(1); it < num_buckets; it++ {
            bucket_ptrs[myid][it] = bucket_ptrs[myid][it-1]
            for k := 0; k < num_procs; k++ {
                if k < myid {
                    bucket_ptrs[myid][it] += bucket_size[k][it]
                } else {
                    bucket_ptrs[myid][it] += bucket_size[k][it-1]
                }
            }
        }
        defer group.Done() // <
    }(myid) // <
} // <
group.Wait() // <

npb.ParallelFor(
    num_keys,
    int64(num_procs),
    func(myid int64, it int64) {
        k := key_array[it]
        key_buff2[bucket_ptrs[myid][k>>shift]] = k
        bucket_ptrs[myid][k>>shift]++
    },
)
```

# Benchmark IS Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Não existe um meio trivial de declarar um threadprivate em Go, igual foi utilizado nas implementações originais.
- Instruções e blocos paralelos foram adaptados para utilizar matrizes compartilhadas, com distribuição no acesso pelo id.

```
group.Add(num_procs - 1) // <
for myid := 0; myid < num_procs-1; myid++ { // <
    go func(myid int) { // <
        for it := int64(0); it < num_buckets; it++ {
            for k := myid + 1; k < num_procs; k++ {
                bucket_ptrs[myid][it] += bucket_size[k][it]
            }
        }
        defer group.Done() // <
    }(myid) // <
} // <
group.Wait() // <
```

# Benchmark IS Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Também não existe um meio trivial de realizar escalonamento dinâmico de um for paralelo igual é feito em OpenMP.
- Neste caso, decidimos trocar para escalonamento estático usando a função do nosso módulo.

```
#pragma omp for schedule(dynamic)
for( i=0; i< NUM_BUCKETS; i++ ) {
    k1 = i * num_bucket_keys;
    k2 = k1 + num_bucket_keys;
    for ( k = k1; k < k2; k++ )
        key_buff_ptr[k] = 0;
    m = (i > 0)? bucket_ptrs[i-1] : 0;
    for ( k = m; k < bucket_ptrs[i]; k++ )
        key_buff_ptr[key_buff_ptr2[k]]++;
    key_buff_ptr[k1] += m;
    for ( k = k1+1; k < k2; k++ )
        key_buff_ptr[k] += key_buff_ptr[k-1];
}
```

# Benchmark IS
# Rank

- Em Go, não há cláusulas equivalentes às do OpenMP. As declarações paralelas precisam ser definidas manualmente.
- Também não existe um meio trivial de realizar escalonamento dinâmico de um for paralelo igual é feito em OpenMP.
- Neste caso, decidimos trocar para escalonamento estático usando a função do nosso módulo.

```
npb.ParallelFor(
    num_buckets,
    int64(num_procs),
    func(myid int64, i int64) {
        var (
            k1, k2 int64
            m      int64
        )
        k1 = i * num_bucket_keys
        k2 = k1 + num_bucket_keys
        for k := k1; k < k2; k++ {
            key_buff_ptr[k] = 0
        }
        if i > 0 {
            m = bucket_ptrs[myid][i-1]
        }
        for k := m; k < bucket_ptrs[myid][i]; k++ {
            key_buff_ptr[key_buff_ptr2[k]]++
        }
        key_buff_ptr[k1] += m
        for k := k1 + 1; k < k2; k++ {
            key_buff_ptr[k] += key_buff_ptr[k-1]
        }
    },
)
```

# Benchmark IS
# Rotinas principais

- Rank
- Ainda no final da função, é feito o procedimento da verificação parcial.
- Verifica se os valores apontados pelas chaves de teste estão ordenados corretamente.

```
for i = 0; i < TEST_ARRAY_SIZE; i++ {
    k := partial_verify_vals[i]
    if 0 < k && k <= num_keys-1 {
        keyRank := key_buff_ptr[k-1]
        failed := false
        switch npb.Class {
        case "S":
            if i <= 2 && keyRank != (test_rank_array[i] + iteration) {
                failed = true
            } else if i > 2 && keyRank != (test_rank_array[i] - iteration) {
                failed = true
            } else {
                passed_verification++
            }
            break
        // Continua os cases...
        }
        if failed {
            fmt.Println("Failed partial verification: iteration ",
                        iteration, ", test key ", i, "\n")
        }
    }
}
```

# Benchmark IS
# Rotinas principais

- Full Verify
- Possui dois laços paralelos:
- Um laço para ordenar todos os valores dentro de cada bucket.
- O código original definia um escalonamento dinâmico.
- Implementamos usando nossa função de escalonamento estático.

```
func FullVerify() {
    var (
        j        int64
        numProcs = runtime.NumCPU()
    )
    npb.ParallelFor(
        num_buckets,
        int64(numProcs),
        func(myid int64, j int64) {
            var k, k1 int64
            if j > 0 {
                k1 = bucket_ptrs[myid][j-1]
            } else {
                k1 = 0
            }
            for i := k1; i < bucket_ptrs[myid][j]; i++ {
                key_buff_ptr_global[key_buff2[i]] -= 1
                k = key_buff_ptr_global[key_buff2[i]]
                key_array[k] = key_buff2[i]
            }
        },
    )
    // ...
}
```

# Benchmark IS
# Rotinas principais

- Full Verify
- ➢ Possui dois laços paralelos:
- Um laço para verificar se ainda resta algum número desordenado.
- O código original definia um for paralelo com redução para a variável de contagem.
- Inicialmente, consideramos usar goroutines com channels ou instruções atômicas, mas trocamos para uma contagem serial devido às dificuldades.

```
func FullVerify() {
    var (
        j        int64
        numProcs = runtime.NumCPU()
    )

    // ...
    j = 0
    for i := int64(1); i < num_keys; i++ {
        if key_array[i-1] > key_array[i] {
            j++
        }
    }
    if j != 0 {
        fmt.Println("Full_verify: number of keys out of sort: ", j)
    } else {
        passed_verification += 1
    }
}
```

# Benchmark IS
# Dificuldades encontradas

- Inicialmente, a função ParallelFor foi criada para instanciar uma goroutine para cada tarefa, análogo ao for do OpenMP.
- O runtime do Go ficava responsável por escalonar as n tarefas entre os processadores.
- Permitia escalonamento estático e dinâmico.
- A função f receberia um WaitGroup para sinalizar quando foi encerrada.

```
func ParallelFor(
    n int64,
    step int64,
    group *sync.WaitGroup,
    f func(id int64, i int64, group *sync.WaitGroup),
    scheduling string,
) {
    (*group).Add(int(n))
    switch scheduling {
    case "static":
        for it := int64(0); it < n; it += step {
            for myid := int64(0); myid < step && it+myid < n; myid++ {
                go f(myid, it+myid, group)
            }
        }
        break
    case "dynamic":
        for it := int64(0); it < n; it++ {
            // Sorteia um id aleatório para cada task
            go f(rand.Int63()%step, it, group)
        }
        break
    }
    (*group).Wait()
}
```

# Benchmark IS
# Dificuldades encontradas

- Problemas na verificação parcial e verificação total!
- As iterações em que a verif. parcial falhava não eram constantes.
- Quantidade de chaves desordenadas na verif. final também não era constante, inclusive podia encerrar ordenada.
- Keys com problema eram quase sempre a 3 e 4, em execuções consecutivas, falhas na key 2 apareciam.

```
C:\FOR_WORK\Go\NPB-Golang\go_build_NPB_Golang.exe IS CLASS=S


 NAS Parallel Benchmarks 4.1 Parallel Golang version - IS Benchmark
 Size:  65536  (class  S )
 Iterations:  10



Failed partial verification: iteration  4 , test key  4
Failed partial verification: iteration  4 , test key  3
Failed partial verification: iteration  6 , test key  3

Failed partial verification: iteration  6 , test key  4

Failed partial verification: iteration  9 , test key  3

Failed partial verification: iteration  9 , test key  4

Failed partial verification: iteration  10 , test key  3

Failed partial verification: iteration  10 , test key  4

Full_verify: number of keys out of sort:  3

 IS BENCHMARK COMPLETED
```

# O que vamos fazer agora?

1. Se necessitar uma 2ª apresentação da implementação deste benchmark, também iremos portar a versão sem buckets.
    - Como a implementação do Dalvan em C++ está incompleta, a nossa será baseada na original da NASA em C
2. Fazer a análise estatística dos testes.
    - Tempos de execução já estão disponíveis no repositório em docs/is
3. Começar a implementação do kernel CG: Conjugate Gradient.

# Relatório 2: Benchmark IS Em Golang

Kevin S. Pereira, Thomazio Giacobbe

https://github.com/thomaziogiacobbe/NPB-Golang