Exercício 9

A mais simples forma de fazer uma soma de um conjunto de partes inteiras se dá por um loop sequencial, um *for loop* é capaz de realizar esta tarefa de forma totalmente sequencial, sem se preocupar com sincronismo.

```
int loop_sum(int length, int parts*)
{
    int sum = 0;
    for (int i = 0; i < length; i++){ sum += parts[i]; }
    return sum;
}
```

É possivel realizar esta mesma soma por divisão e conquista formando uma árvore binária até obter-se um par ou um número singelo, o par é somado e resultado, assim como o número singelo, é imediatamente adicionando a soma total. Este caso resolvido por uma lógica sequencial, pode ser realizado através de uma recursão, conforme o código abaixo. Entretanto, essa solução não ajuda a melhorar o tempo de execução, pois adiciona a complexidade da recursão ao problema, mas deixa a solução nos moldes para adicionar o paralelismo.

```
int divide_and_conquer_sum(int length, int* parts)
{
    if(length == 2)
    {
        return (parts[0] + parts[1]);
    }
    else if(length == 1)
    {
         return  parts[0];
    }
    else
    {
        int _l = length / 2;
        int _lp = length / 2 + length % 2;

        // Left
        int* _div = copy(parts, 0, _l - 1);
        int left = divide_and_conquer_sum(_l, _div);

        // Right
        int* _remainder = copy(parts, _l, _l + _lp - 1);
        int right = divide_and_conquer_sum(_lp, _remainder);

        return left + right;
    }
}
```

Para adicionar o paralelismo, basta que em cada nó seja verificada a necessidade de dividir e quebrar o conjuntos de partes entre dois sub-conjuntos esquerda a direita, cada parte da divisão gera duas threads para cada uma das partes dividas, repetindo o processo recursivamente, até que os sub-conjutos se resumam a um par de partes ou a um número. Então a soma é calculada e o valor total da soma atualizado após o *join* das threads.

```
void* divide_and_conquer_sum_thread(void* arg)
{
    divide_and_conquer_args* args = (divide_and_conquer_args*) arg;
    int* data = (int*) malloc(sizeof(int));

    if(args->length == 2)
    {
        *data = (args->parts[0] + args->parts[1]);
        return data;
    }
    else if(args->length == 1)
    {
        *data = args->parts[0];
         return  data;
    }
    else
    {
        int _l = args->length / 2;
        int _lp = args->length / 2 + args->length % 2;

        // Left
        int* _div = copy(args->parts, 0, _l - 1);
        pthread_t left_thd;
        divide_and_conquer_args largs = {_l, _div};
        pthread_create(&left_thd, NULL, divide_and_conquer_sum_thread, &largs);
        int* left;

        // Right
        int* _remainder = copy(args->parts, _l, _l + _lp - 1);
        pthread_t right_thd;
        divide_and_conquer_args rargs = {_lp, _remainder};
        pthread_create(&right_thd, NULL, divide_and_conquer_sum_thread, &rargs);
        int* right;

        pthread_join(left_thd, &left);
        pthread_join(right_thd, &right);

        *data = *left + *right;
        return data;
    }
}
```

Com essas três abordagens, é possível construir um programa de execução realizando as três formas de soma de partes.

```
int main(int argc, char *argv[])
{
    int arr[5] = {1,2,3,4,5};

    // Sequencial
    int sum = 0;
    sum = soma( NELEMS(arr), arr);
    printf("Sequencial %d\n", sum);

    // Sequencial Recursivo
    sum = 0;
    sum = divide_and_conquer_sum( NELEMS(arr), arr);
    printf("Sequencial Recursivo %d\n", sum);

    // Paralelo Recursivo
    pthread_t divide_and_conquer_thd;
    int* result;
    divide_and_conquer_args args = {NELEMS(arr), arr};
    pthread_create(&divide_and_conquer_thd, NULL, divide_and_conquer_sum_thread,
&args);

    pthread_join(divide_and_conquer_thd, &result);
    printf("Paralelo Recursivo %d\n", *result);
    return 0;
}
```

Ao executar o programa acima, utilizando um conjunto de partes {1,2,3,4,5}, obtem-se a validação pelos logs, com todas as somas resultando no mesmo valor.

```
Sequencial 15
Sequencial Recursivo 15
Paralelo Recursivo 15
```

Exercício 17

Utilizando uma lógica parecida com a do exercício 9, a recursão pode auxiliar o paralelismo e dividir o trabalho em threads. Foi implementado o conceito de semáforos para acessar de forma segura e garantir sincronismo entre o manager central e as threads. Abaixo está descrito um modelo utilizando sintaxe C para um pseudo-código representando a paralelização do branch-and-bound.

```
void* branch_and_bound_thread(void* arg)
{
    branch_and_bound_args* args = (branch_and_bound_args*) arg;

    int resposta = pergunta_ao_manager_por_novos_nodes(args->manager);
    if(!resposta)
    {
        realiza_tarefas(args->data);
```

```
    }
    else
    {

        //wait
        sem_wait(&mutex);

        // Garante comunição sincrona com manager
        atualiza_data(args->data);
        r_type result = verifica_tarefas_faltantes(args->manager);
        notifica_novos_nodes_ao_manager(result, args->manager);

        //signal
        sem_post(&mutex);

        pthread_t thd;
        pthread_create(&thd, NULL, divide_and_conquer_sum_thread, args);

        pthread_join(thd, NULL);
    }
}
```