

Trabalho 4 - Paradigmas de Programação

Grupo: Pangolim

Componentes: Devis Costa Pereira e Gustavo Arrua Fantinel

Para a resolução do problema, utilizamos duas abordagens baseadas na API OpenMP, as quais se distinguem pelo uso ou não de parâmetros e variáveis de ambiente



Especificações do computador utilizado

Todos os experimentos foram realizados em um computador com as seguintes especificações:

- Processador: Intel Core i7-4500U 1.8 GHz
- Memória RAM: 8 GB
- GPU: Nvidia GT 740M
- Sistema Operacional: Linux Mint 19.3



Solução 1 - Utilização de parâmetros

A primeira solução implementada baseia-se na utilização de parâmetros de execução do programa para a definição de:

- Tamanho da população;
- Número de testes;
- Probabilidade máxima (fixa em 101 para comportar 100% de probabilidade);
- Número de threads;




Execução:

Executamos o programa com os seguintes parâmetros:

- `./omp_virus <tam população> <nro testes> <prob máxima> <nro threads>`

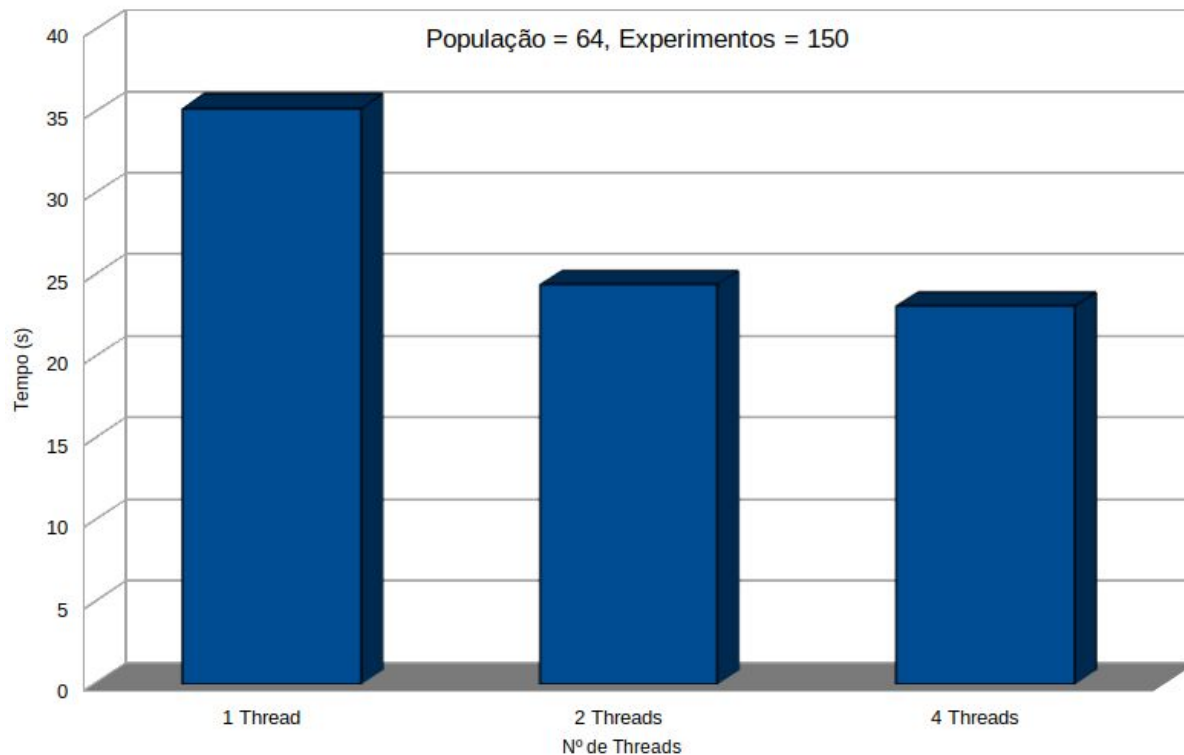
Executamos, portanto, as funções da classe `Population` de forma paralela e, dessa forma, a variável de controle booleana `env` tem seu valor setada para falso, fazendo com que a execução do programa passe pelo seguinte trecho de código:



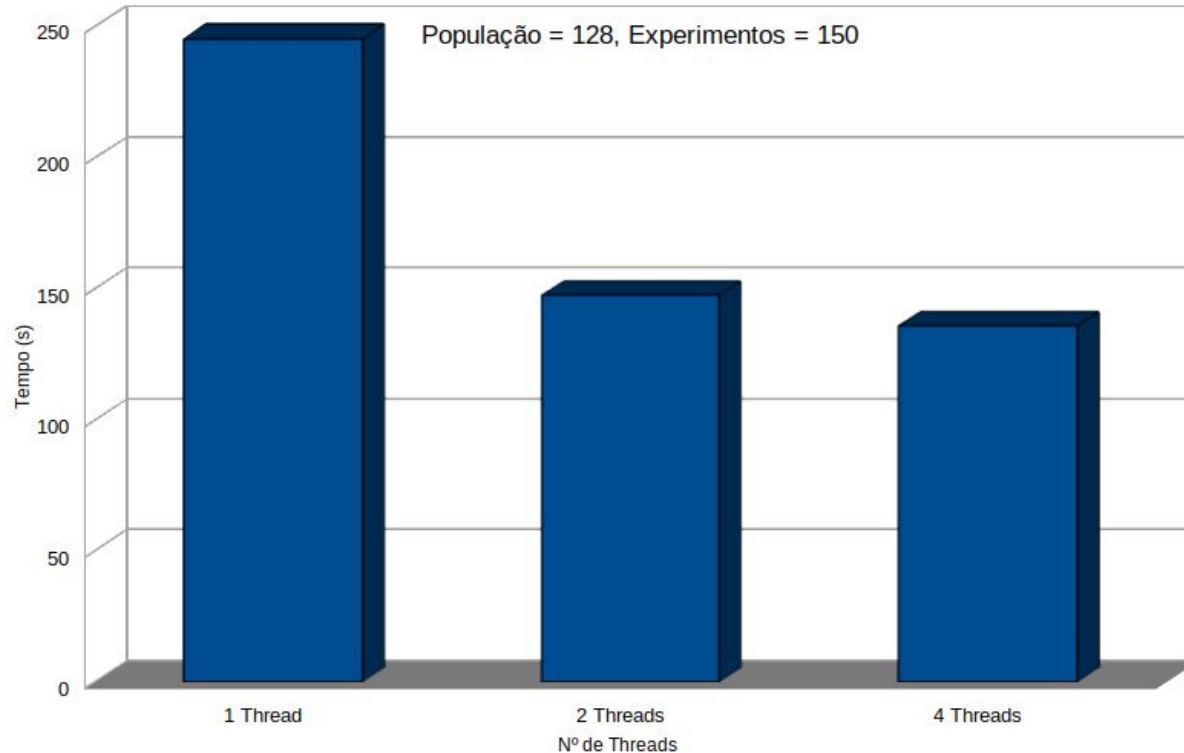


```
if (!env) { // EXECUÇÃO SETANDO THREADS POR PARAMETRO
    omp_set_num_threads(n_threads);
    Population *population = new Population(population_size);
    for (int ip = 0; ip < n_probs; ip++) {
        prob_spread[ip] = prob_min + (double)ip * prob_step;
        percent_infected[ip] = 0.0;
        rand.setSeed(base_seed + ip);
        for (int it = 0; it < n_trials; it++) {
            population->propagateUntilOut(population->centralPerson(),
                                           prob_spread[ip], rand);
            percent_infected[ip] += population->getPercentInfected();
        }
        percent_infected[ip] /= n_trials;
        std::cout << std::fixed << prob_spread[ip] << ", "
                  << percent_infected[ip] << std::endl;
    }
}
```

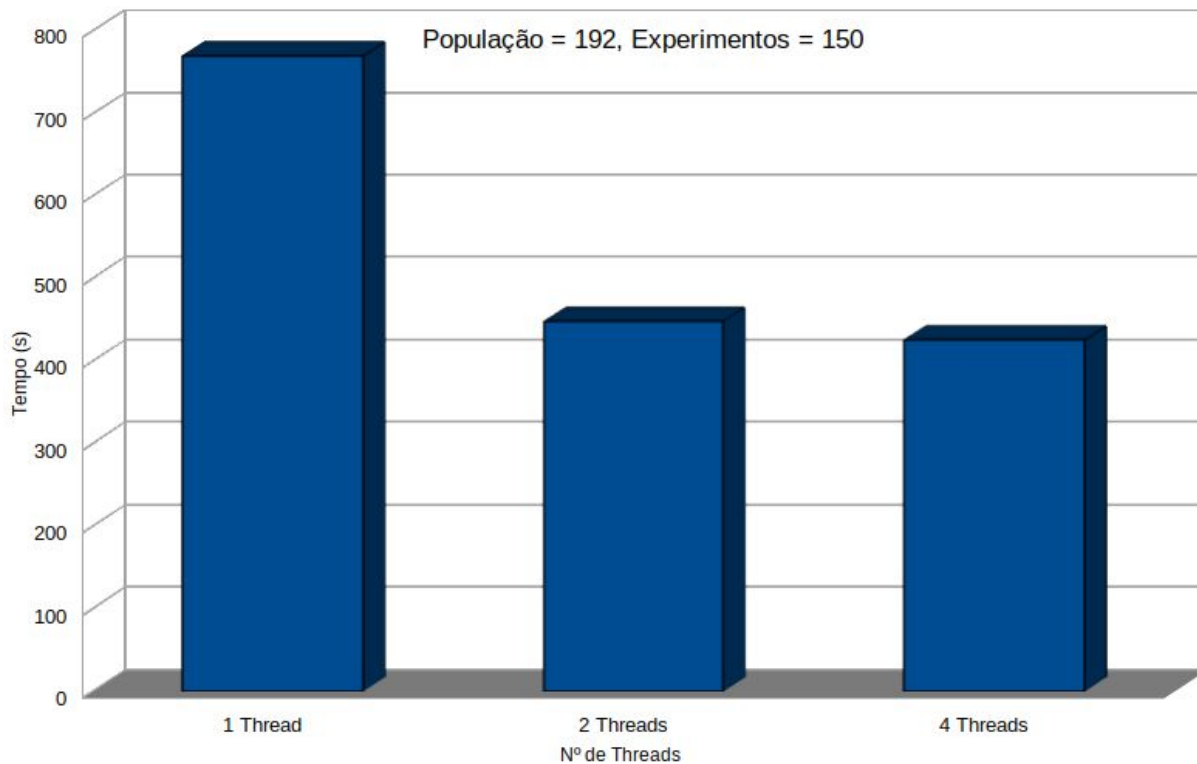
Caso 1 - População = 64, Experimentos = 150



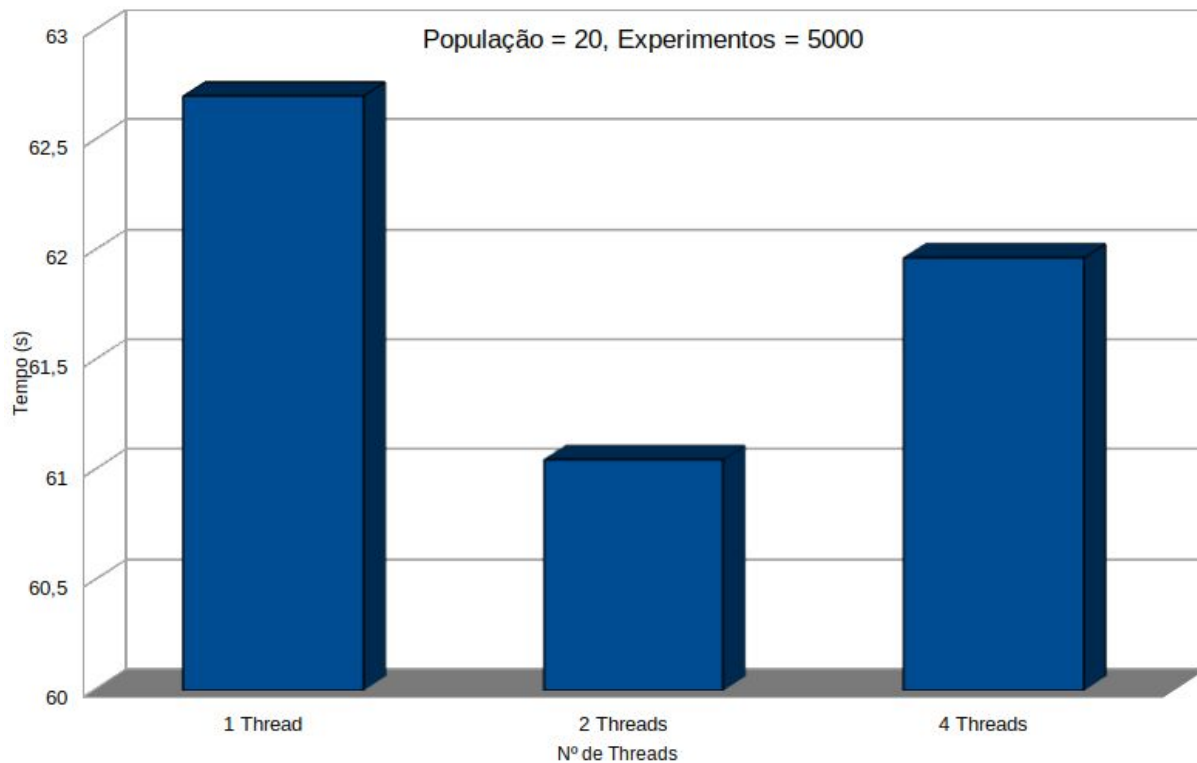
Caso 2 - População = 128, Experimentos = 150



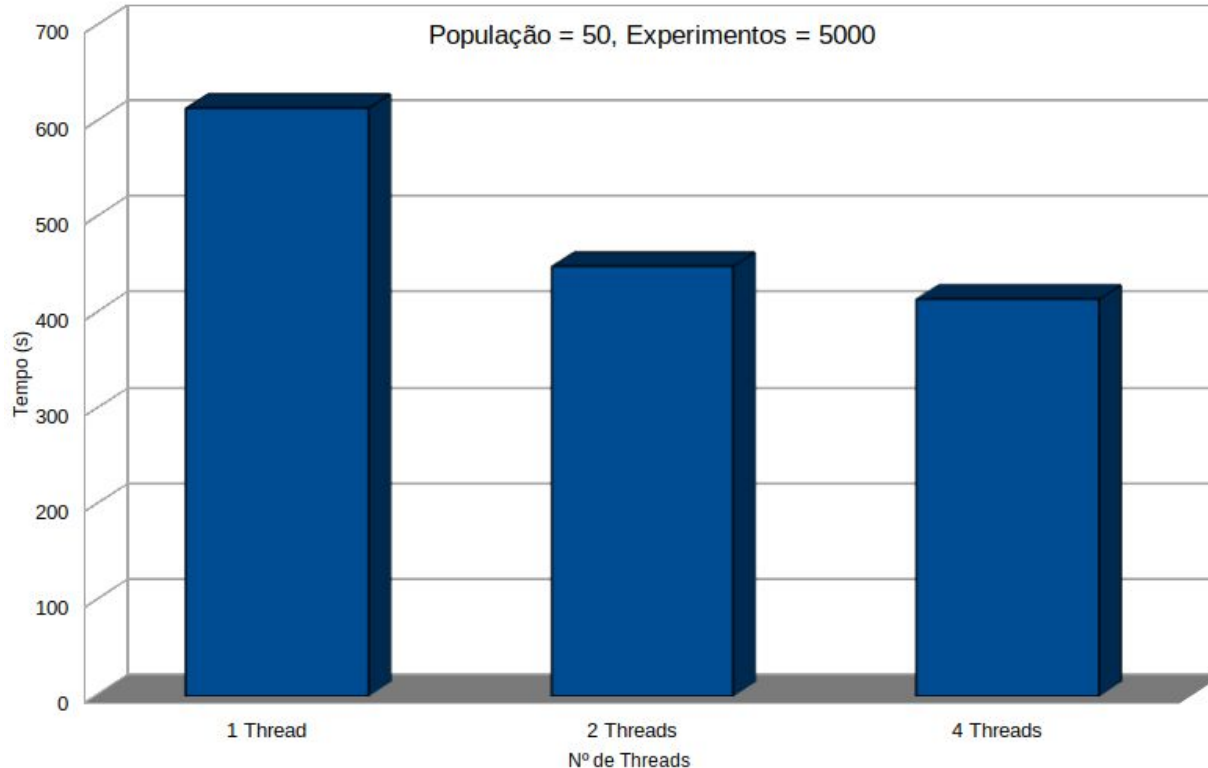
Caso 3 - População = 192, Experimentos = 150



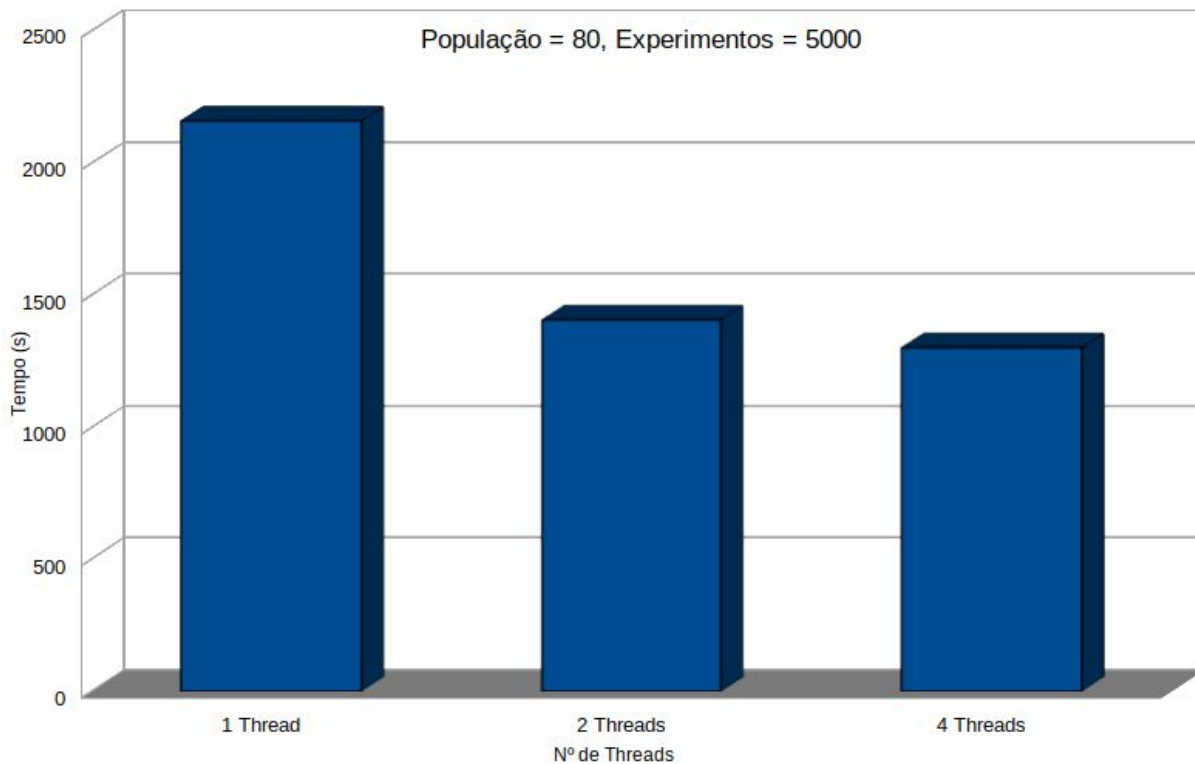
Caso 4 - População = 20, Experimentos = 5000



Caso 5 - População = 50, Experimentos = 5000



Caso 6 - População = 80, Experimentos = 5000



Solução 2 - Utilizando variáveis de ambiente

A segunda solução baseia-se no uso de variáveis de ambiente sendo configuradas antes da execução do programa. Executamos, portanto:

```
export OMP_NUM_THREADS=<nro threads 1º nível>,<nro threads 2º nível>
```

```
./omp_virus <tam população> <nro testes> <prob máxima>
```



Dessa forma, estamos setando o número de threads disponíveis em uma variável de ambiente, número este que será lido no código através da função `std::getenv("OMP_NUM_THREADS")`. Utilizamos, ainda, `omp_set_max_active_levels(2)` para definir que queremos no máximo dois níveis ativos de paralelismo, `omp_set_nested(true)` para permitir paralelismo nos loops aninhados e a diretiva `#pragma omp parallel for schedule(dynamic) reduction(+ : percent_infected[ip])`, para guiar as tasks de cada thread.

A implementação se dá da seguinte forma:



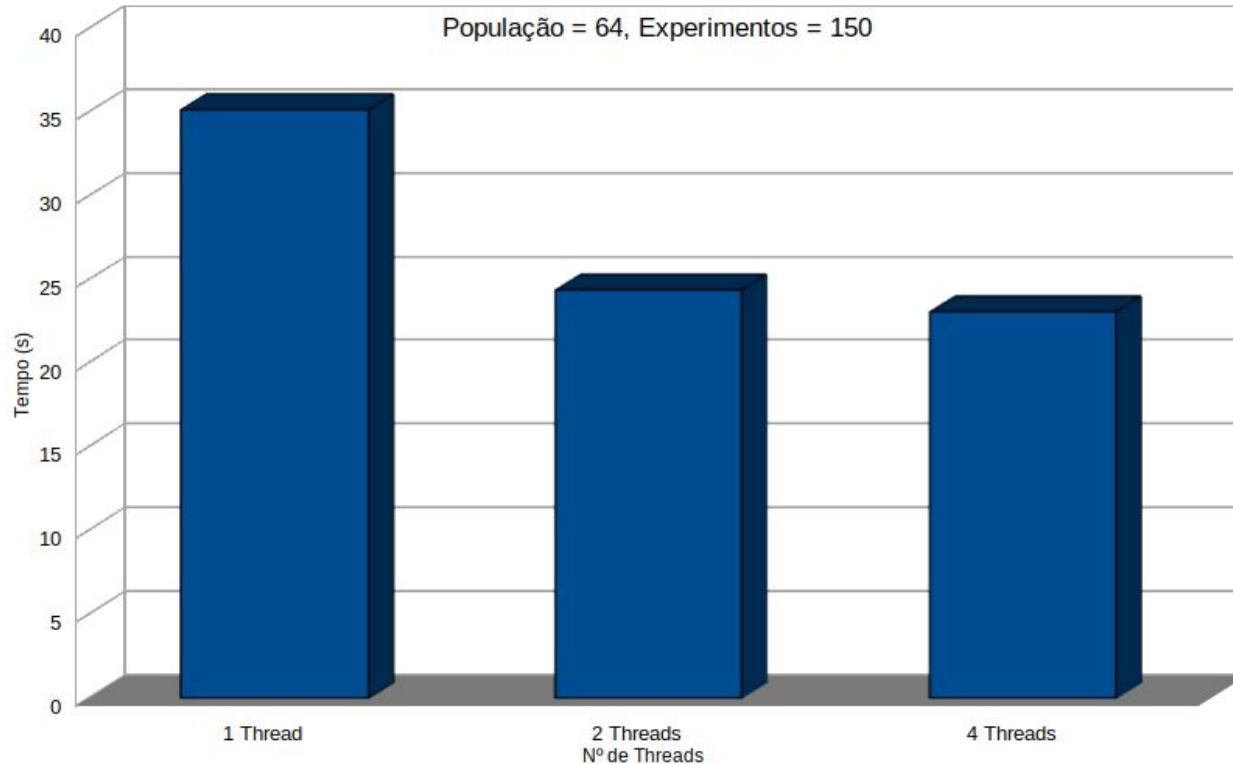
```

} else { // EXECUÇÃO USANDO THREADS DO ENV
    const char *env_th = std::getenv("OMP_NUM_THREADS");
    n_threads = std::stoi(env_th); // Pega as Threads do primeiro nível
    Population **population = new Population *[n_threads];
    for (int i = 0; i < n_threads; i++)
        population[i] = new Population(population_size);

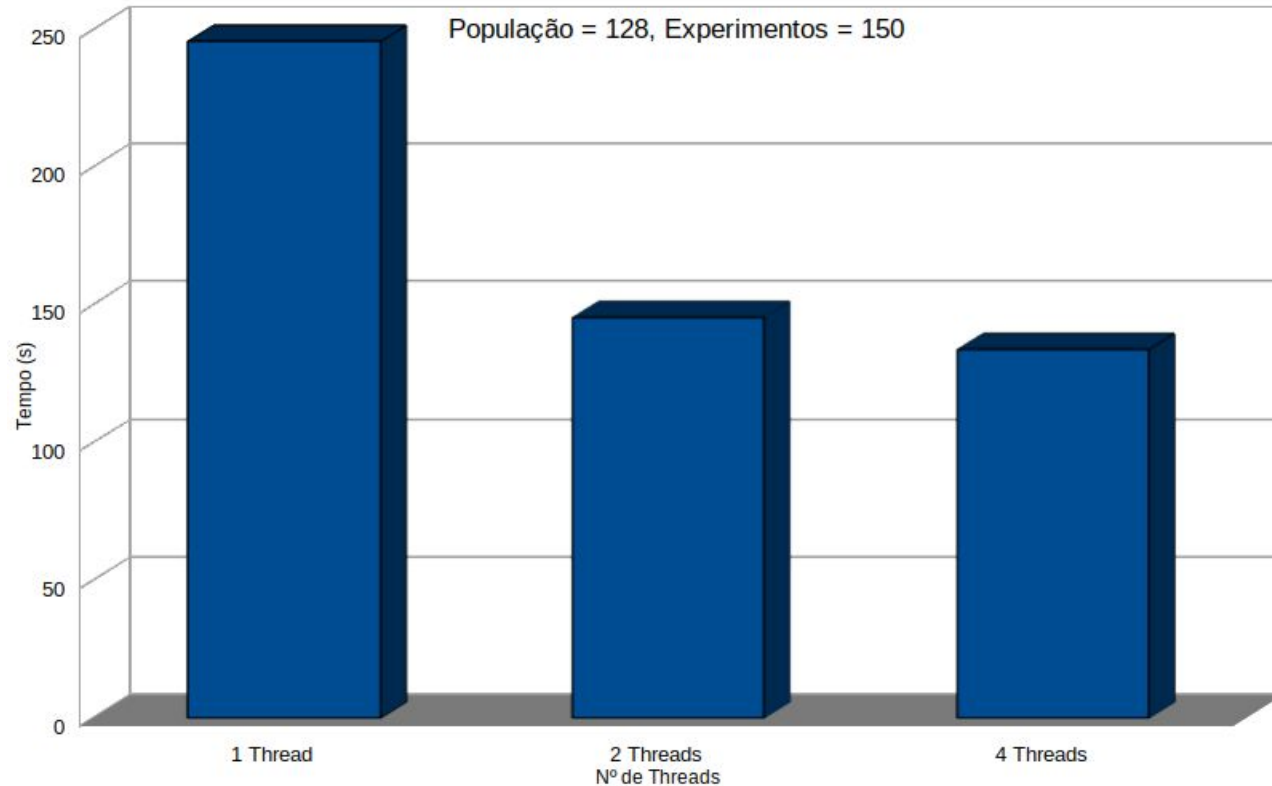
    omp_set_max_active_levels(2);
    omp_set_nested(true);
    for (int ip = 0; ip < n_probs; ip++) {
        prob_spread[ip] = prob_min + (double)ip * prob_step;
        percent_infected[ip] = 0.0;
        rand.setSeed(base_seed + ip);
#pragma omp parallel for schedule(dynamic) reduction(+ : percent_infected[ip])
        for (int it = 0; it < n_trials; it++) {
            int id = omp_get_thread_num();
            population[id]->propagateUntilOut(
                population[id]->centralPerson(), prob_spread[ip], rand);
            percent_infected[ip] +=
                population[id]->getPercentInfected();
        }
        percent_infected[ip] /= n_trials;
        std::cout << std::fixed << prob_spread[ip] << ", "
                    << percent_infected[ip] << std::endl;
    }
}

```

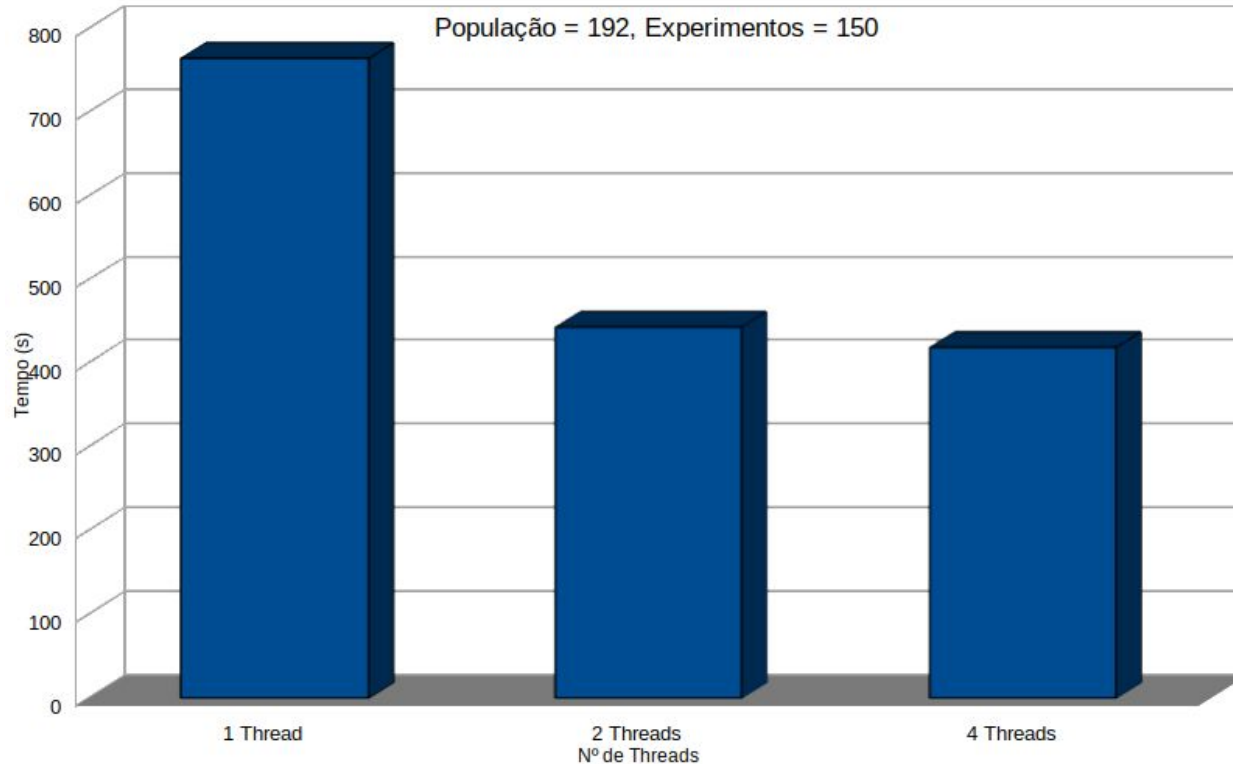
Caso 1 - População = 64, Experimentos = 150



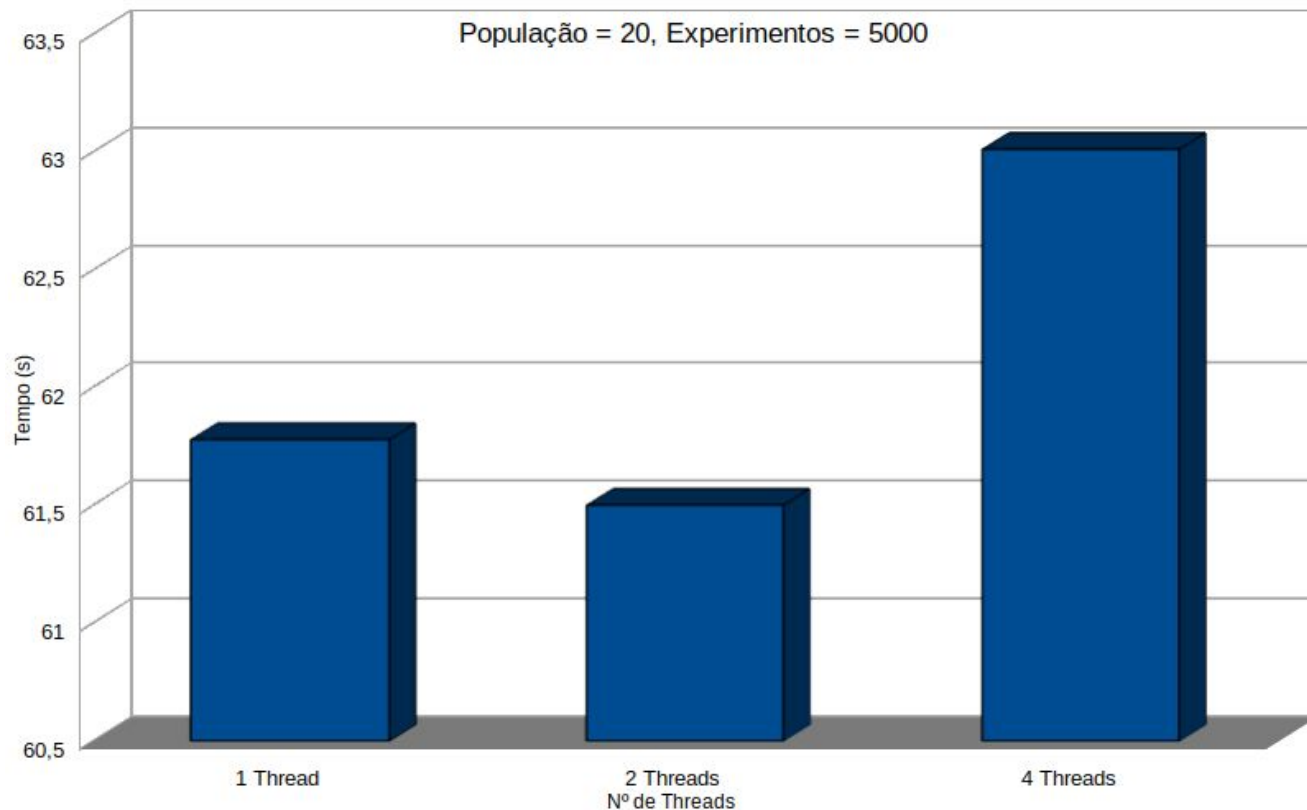
Caso 2 - População = 128, Experimentos = 150



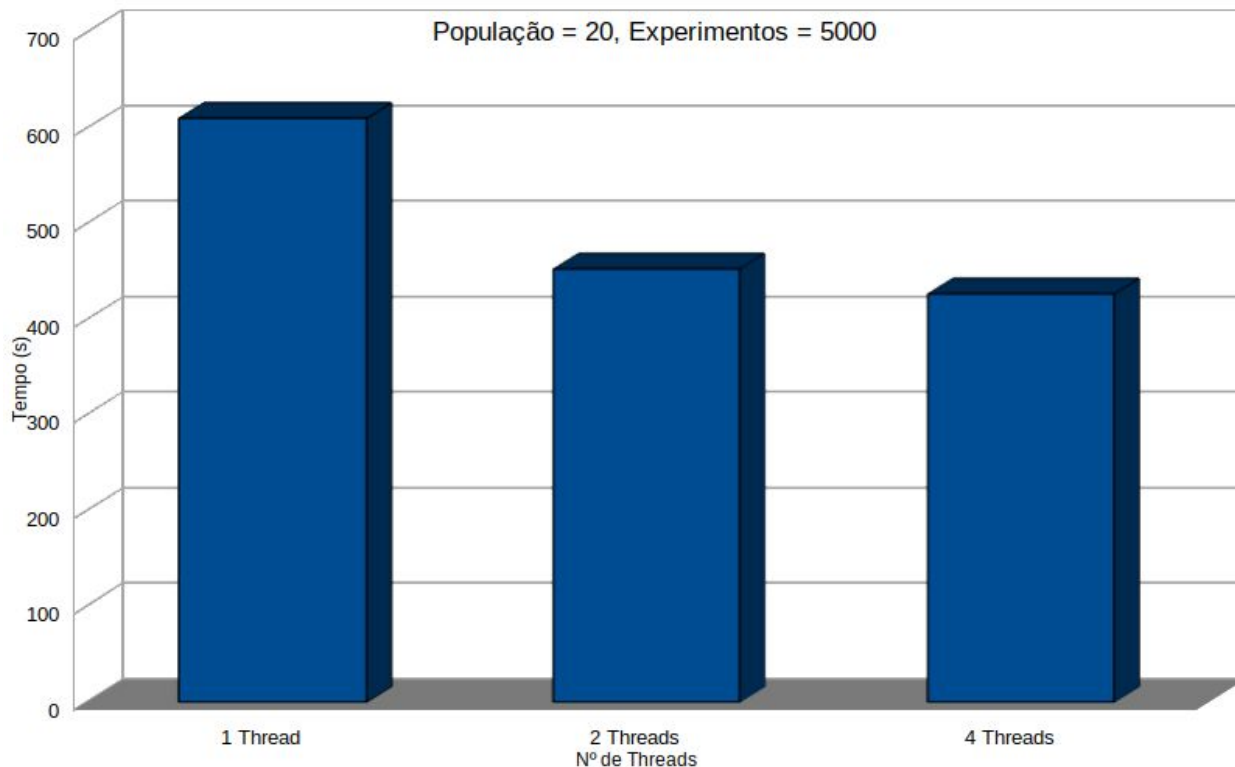
Caso 3 - População = 192, Experimentos = 150



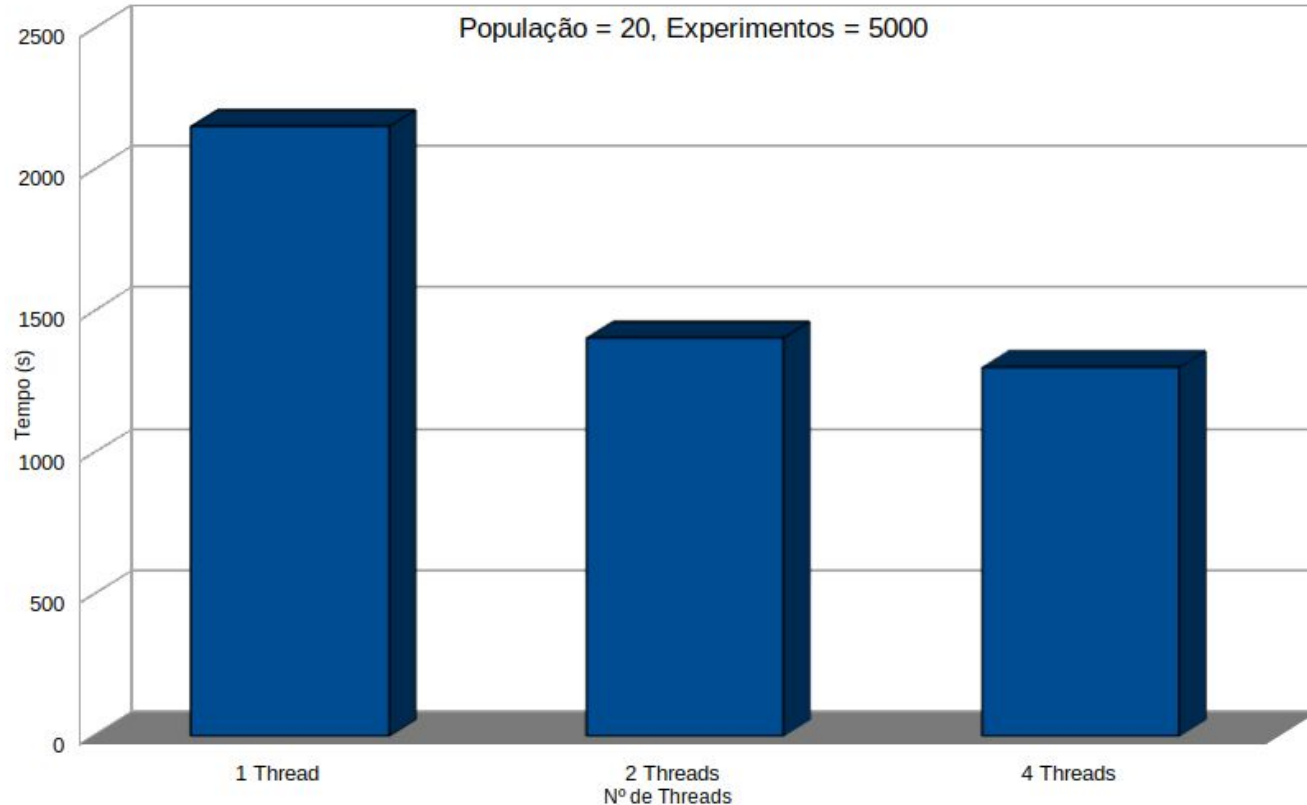
Caso 4 - População = 20, Experimentos = 5000



Caso 5 - População = 50, Experimentos = 5000



Caso 6 - População = 80, Experimentos = 5000



Resultados e discussões

De posse dos resultados obtidos durante os experimentos, é perceptível que um problema como o Método de Monte Carlo proposto na especificação do trabalho pode usufruir largamente dos benefícios de performance obtidos pela utilização da api OpenMP. Notório, também, foi o comportamento do programa quando induzido a executar o método em pequenas populações, onde foi possível, inclusive, obter valores menores de tempo de execução com 1 ou 2 threads ao invés de utilizar 4 threads.



Referências

- <https://docs.microsoft.com/pt-br/cpp/parallel/openmp/reference/openmp-functions?view=vs-2019>
- <https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf>
- <https://www.ibm.com/developerworks/br/aix/library/au-aix-openmp-framework/index.html>

