**ALGORITMO DE ORDENAÇÃO GNOME SORT**

**Joice Ramos1, Rodrigo Curvello2**

1Aluna da 7ª fase do Curso de Bacharelado em Ciência da Computação – Instituto Federal Catarinense

2Professor do Instituto Federal Catarinense, Campus Rio do Sul

(IFC) 89160-240 – Rio do Sul – SC, Brasil

joice\_ramos-rsl@hotmail.com, rodrigo.curvello@ifc.edu.br

**Abstract.** The purpose of this article is to describe the application of the Gnome Sort ordering algorithm and to analyze a parallelization solution of such algorithm in the C ++ programming language. For this it is presented basic concepts on parallel programming - OpenMP, operation of the algorithm of ordering and a parallel optimization of the algorithm, in order to improve the performance and the time of execution of the same one. To verify the validation of the proposal will be used a calculation of Speedup that aims to verify the gains obtained from the algorithm optimized in relation to the original algorithm.

**Key-words:** Gnome Sort; OpenMP; Parallel programming.

**Resumo.** O presente artigo tem como objetivo descrever a aplicação do algoritmo de ordenação Gnome Sort e analisar uma solução de paralelização de tal algoritmo na linguagem de programação C++. Para isto é apresentado conceitos básicos sobre programação paralela – OpenMP, funcionamento do algoritmo de ordenação e uma otimização paralela do algoritmo, a fim de melhorar o desempenho e o tempo de execução do mesmo. Para verificar a validação da proposta será utilizado um cálculo de Speedup que possui como objetivo verificar os ganhos obtidos do algoritmo otimizado em relação ao algoritmo original.

**Palavras-chave:** Gnome Sort; OpenMP; Programação paralela.

# 1. Introdução

O crescente avanço tecnológico dos últimos anos e o desenvolvimento do processamento paralelo para a computação científica, interferiram diretamente no desenvolvimento de API’s de paralelização. Para isto foi desenvolvido a API OpenMP pela ARB. Com o alto crescimento científico a exigência de resultados confiáveis e próximo a realidade se torna um fator de extrema relevância para o desenvolvimento de aplicações. De acordo com MARCONATTO (2017, apud SENA e COSTA, 2008) a computação cientifica encontra-se com várias barreiras que surgem em virtude da limitação física dos computadores, como a demanda por processamento numérico, armazenamento de dados, visualização, entre outros.

A computação de Alto Desempenho surge como uma técnica na área computacional com o objetivo de aumentar a velocidade de processamento ou até mesmo viabilizar as simulações numéricas de problemas físicos complexos e com elevado grau de discretização.

Nesta área encontra-se o processamento paralelo que contém dois ambientes sendo: ambientes de memória distribuída onde as tarefas são distribuídas e executadas por diferentes processadores e ambientes de memória compartilhada consiste na divisão das tarefas entre vários processadores que compartilham o mesmo recurso de memória global (MARCONATTO, 2017, P. 14. SENA e COSTA, 2008).

Conforme Silva (2001), um dos objetivos da programação paralela é resolver problemas complexos e sobre grandes quantidades de dados utilizando de forma eficiente os recursos computacionais disponíveis atualmente em sistemas com múltiplos elementos de processamento.

A partir disto, Marconatto (2017) propôs a paralelização de alguns algoritmos de

ordenação, como Quick Sort, Bubble Sort, Radix Sort e Bitonic Sort. O autor apresenta a versão paralela baseado na versão serial de tais algoritmos. O presente trabalho possui como objetivo avaliar e propor melhoria no algoritmo de ordenação Gnome Sort.

O algoritmo de ordenação Gnome Sort é semelhante ao Insertion sort, que em vez de inserir diretamente o elemento ao seu devido lugar, o algoritmo executa uma série de permutações assim como o Bubble sort [4].

Com o intuito de verificar a validade da otimização proposta, é utilizado uma métrica de desempenho chamada de speedup, que representa a razão entre o tempo de execução de um programa paralelo e o tempo de execução de sua versão sequencial (SILVA, 2011). Esta técnica será aplicada para representar o ganho de desempenho obtido no algoritmo otimizado

# 2. Programação paralela – OpenMP

A computação paralela, segundo Sebesta (2003), possibilita que duas ou mais tarefas sejam executadas simultaneamente por mais de um processador e tem como objetivo reduzir o tempo de execução. O OpenMP, como descrito em Cenapad (2010, p.7), é “uma interface de programação (API), portável, baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores”.

OpenMP define um conjunto de diretivas de compilação, chamadas de função e variáveis de ambiente, onde a paralelização do código para uma arquitetura diferente é de responsabilidade do compilador dessa arquitetura em que deseja rodar o programa, não modificando expressivamente o algoritmo e admitindo que o programa siga trabalhando de forma sequencial (BONAT & GIUSTI,2008).

A interface de programação paralela com memória compartilhada, OpenMP define como os cálculos serão distribuídos entre os fluxos de execução, a forma como os dados estão replicados ou compartilhados entre os fluxos de execução e as sincronizações entre os fluxos de execução (MAILLARD & CERA, 2010).

O desenvolvimento de aplicações OpenMP dá-se pelo pragmas adicionados aos programas C, C++ e Fortan, juntamente com a definição de rotinas definidas através das bibliotecas. A distribuição dos cálculos é feita utilizando a diretiva parrallel: “ela especifica o disparo de threads concorrentes (sendo o número de threads determinado na execução), e que o bloco entre {. . . } subsequente ao pragma deverá ser executado em paralelo”, de acordo com Maillard & Cera (2010).

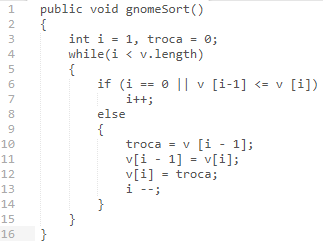
# 3. Algoritmo de ordenação Gnome Sort

* + - 1. Gnome sort foi originalmente proposto por Hamid Sarbazi-Azad (professor de engenharia da computação na Universidade de tecnologia de Sharif) nos anos 2000 e foi chamado de Stupid sort. Mais tarde, foi descrito por Dick Grune e chamado de "Gnome sort". É um algoritmo de ordenação semelhante ao Insertion sort, que em vez de inserir diretamente o elemento ao seu devido lugar, o algoritmo executa uma série de permutações assim como o Bubble sort. A média de tempo de execução é [O](https://en.wikipedia.org/wiki/Big_O_notation)(n2), mas tende para O(n) se a lista é ordenada quase inicialmente (KANASZ, ROBERT. 2010).

Esse algoritmo foi baseado na técnica usada pelo gnomo de jardim holandês (du: tuinkabouter). Um gnomo de jardim classifica uma linha de vasos de flores. Basicamente, ele olha para o vaso de flores ao lado dele e do anterior; se eles estão na ordem certa, que ele passa um vaso para a frente, caso contrário ele irá trocá-los, passa o vaso para trás caso um deles estiver na posição errada. Ou seja, se não há nenhum vaso anterior, ele avança para a frente; se não há nenhum vaso para trocar, está pronto (GRUNE, DICK).

* + - 1. O algoritmo percorre o vetor comparando seus elementos dois a dois, assim que ele encontra um elemento que está na posição incorreta, ou seja, um número maior antes de um menor, ele troca a posição dos elementos, e volta com este elemento até que encontre seu respectivo lugar, quando encontra o lugar, ele volta na última posição e começa a comparar novamente, assim sucessivamente até o vetor estar todo ordenado.

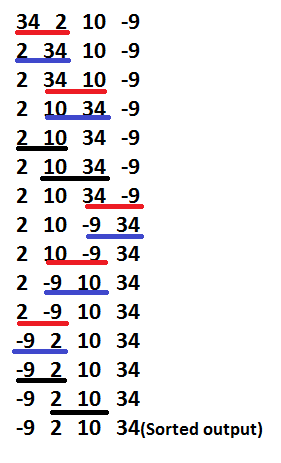
1. O pseudocódigo do Gnome sort é ilustrado abaixo na figura 1.



1. **Figura 1: Pseudocódigo Gnome sort [4]. Modificado pelo autor.**

O algoritmo pega a entrada e compara os elementos de dois em dois. Na linha 4, se o elemento atual da matriz é maior ou igual ao elemento de matriz anterior então passa para o próximo elemento, caso seja menos, ele vai fazer a troca, vai diminuindo a posição do elemento até chegar ao índice correto, linhas 10 à 13. Após a troca, ele volta onde parou e começa a comparar novamente.

1. Abaixo na figura 2 é ilustrado o funcionamento do algoritmo Gnome sort, onde, os elementos que estão sublinhados na cor preta, indica o par que será comparado; os elementos sublinhados na cor vermelha, indica que deverá ser efetuado a troca entre os elementos; e os elementos sublinhado na cor azul, indica que a troca foi efetuada e que esses dois elementos estão ordenados.



1. **Figura 2: Imagem ilustrativa do funcionamento do algoritmo. Fonte: SUDHAKAR, SUMIT.**

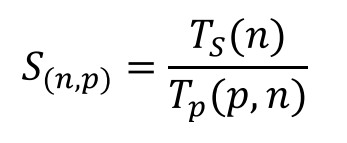
# 4. Métrica de Desempenho

Existe diferentes métricas de desempenho que visam avaliar a performance, desempenho de aplicações paralelas que são: Speedup, Eficiência, Utilização e Qualidade (Rocha, 2007). A métrica utilizada neste trabalho a fim de avaliar a performance da aplicação será a speedup.

Está métrica descreve o ganho de velocidade de processamento de uma aplicação quando executada com 𝑛 processadores. Quanto maior o speedup, mais rápido se encontra o código paralelo (MARCONATTO, 2017).

O Speedup representa a razão entre o tempo de execução de um programa paralelo e o tempo de execução de sua versão sequencial. Por isso, trata-se de uma boa medida para avaliarmos quantitativamente a melhoria de desempenho trazida pela versão paralela de um programa em relação à sua versão sequencial (SILVA, 2011).

A mesma é calculada pela seguinte equação (equação 1):



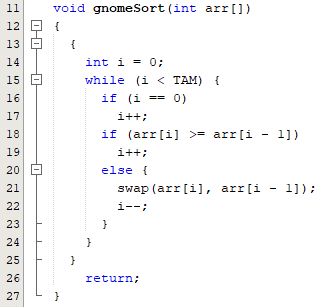
**Equação 1: Equação Speedup para cálculo de desempenho. Fonte: Marconatto, 2017**

Onde, Ts (n) que depende apenas do tamanho 𝑛 do problema, e o tempo de execução do algoritmo paralelo 𝑇𝑝(𝑝, 𝑛), que depende do tamanho do problema e da quantidade 𝑝 de processadores utilizados.

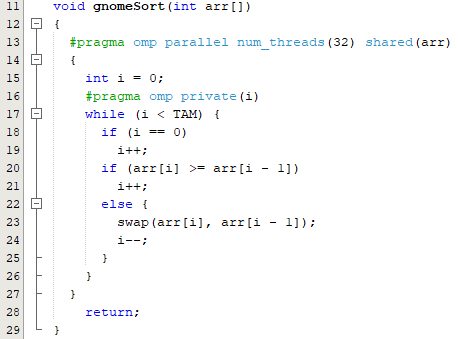
# 5. Otimização do Código Gnome Sort

A otimização do algoritmo utilizou um construtor paralelo, que indica onde inicia a região paralela, a região que será otimizada. Este construtor é a diretiva mais importante do OpenMP, segundo MARCONATTO (2017, apud SENA e COSTA, 2008) (SENA e COSTA, 2008), tendo em vista que ele é o responsável pela indicação da região do código que será paralelizada. Foi utilizada a cláusula Shared, que indica quais variáveis terão endereço de memória acessível por todas as threads dentro da região paralela associada a cláusula. E por último, utilizei a cláusula Private, ela indica que cada thread possui acesso exclusivo a uma cópia das variáveis assim definidas, no caso de uma thread modificar a sua cópia de variáveis, ela não influencia no valor da mesma variável presente nas outras threads. Nesta cláusula as variáveis entram com os valores indefinidos.

A figura 3, logo abaixo, mostra a função do Gnome Sort, código ainda sem a paralelização. Ao lado, na figura 4, temos o mesmo código, porém com o bloco construtor paralelo e as cláusulas utilizadas.



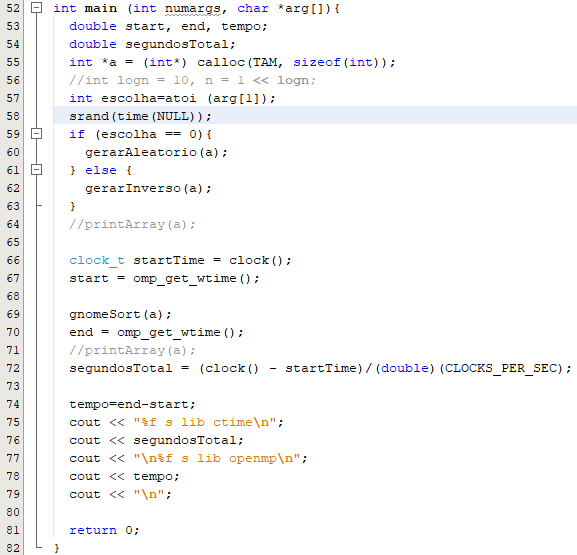
**Figura 3: Código sem paralelização. Fonte: acervo do autor.**



**Figura 4: Código paralelizado. Fonte: acervo do autor.**

É possível verificar onde inicia e termina o bloco paralelo, indicado da linha 13 até a 29 da figura 4. Na linha 13 ainda, foi incluído a cláusula shared, que compartilha a variável “arr”. Na linha 16, incluí a cláusula private, dando acesso exclusivo a variável “i”.

Abaixo na figura 5, temos o main, onde foi criado uma função para calcular o speedup.



**Figura 5: Cálculo speedup. Fonte: acervo do autor.**

Foi utilizado uma função de medição do tempo. Estas funções têm por objetivo retornar tempo, para que possa ser feita a análise de tempo de execução de determinadas partes do código.

A função utilizada foi *omp\_get\_wtime*, linha 67, que retorna um valor em precisão dupla igual ao tempo atual do computador.

Na linha 66, o clock pega o tempo do computador, linha 67, no start, atribui o tempo em que iniciou a execução do bloco paralelo, e linha 70, no end, atribui o tempo quando termina a execução do bloco paralelo. Após isso, na linha 72, a variável “segundosTotal”, atribui o clock e divide por “CLOCKS\_PER\_SEC”. E assim, a variável “tempo”, calcula o tempo, speedup, que levou a paralelização.

# 6. Plataforma de testes

A plataforma de teste utilizada possui as seguintes especificações: processador Intel (R) Core (TM) i7-7500U CPU @ 2.70GHz 2.90GHz, memória RAM de 8 GB, sistema operacional 64 bits, processador base em x64 e rodando na máquina virtual Deepin 15.10.

Para obter a performance de tempo dos dois algoritmos (serial e paralelo), foram realizadas 30 repetições para cada código a fim de eliminar algum ruído na obtenção do tempo. O grupo de threads utilizado possuía o tamanho de 2, 4, 8, 16 e 32 threads para as regiões paralelas, contendo na sua matriz inversa com 1000 posições como dado de entrada.

# 7. Resultados Obtidos

A seguir são apresentados os dados obtidos através de 30 repetições para cada grupo de threads conforme seu tamanho, a partir dos valores coletados foi realizado a média, e então gerado a taxa de desempenho baseado no speedup (tabela 1).

**Tabela 1: Média do tempo de execução (em segundos) dos**

**algoritmos e cálculo de speed up para entrada aleatória.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Paralelo Aleatório** | | | | |
| **2** | **4** | **8** | **16** | **32** |
| 0,00337829 | 0,00365427 | 0,00307670 | 0,00349437 | 0,00339402 |

**Fonte: Elaborado pelo autor, 2019.**

Na tabela 1, podemos observar que a execução com 4 threads foi a que teve melhor desempenho com a paralelização. As demais execuções, com 2, 8, 16 e 32 threads não teve muita diferença entre elas, os resultados foram bem próximos uns dos outros. O menor tempo de execução obtido, ou seja, melhor algoritmo, foi o que utilizou 4 threads na região paralela, com um índice de speedup de aproximadamente 0,00365427. Já o algoritmo com 8 threads na região paralela foi o que apresentou maior índice, obtendo um speedup de 0,00307670 aproximadamente.

**Tabela 2: Média do tempo de execução (em segundos) dos**

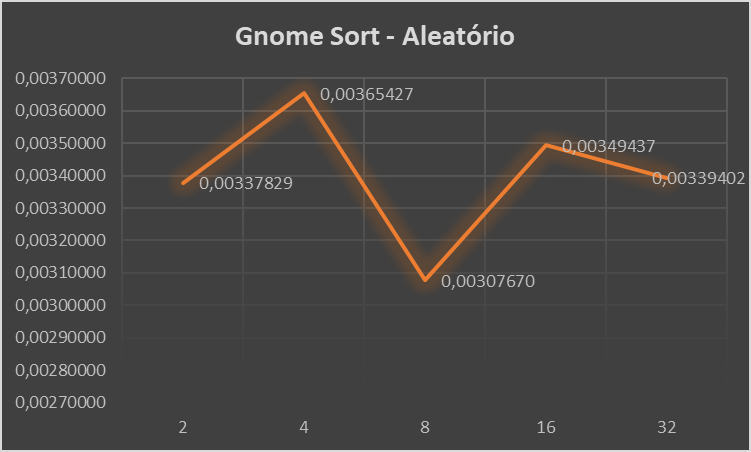
**algoritmos e cálculo de speed up para entrada inversa.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Paralelo Inverso** | | | | |
| **2** | **4** | **8** | **16** | **32** |
| 0,00817272 | 0,00918180 | 0,00893544 | 0,00993868 | 0,01145770 |

**Fonte: Elaborado pelo autor, 2019.**

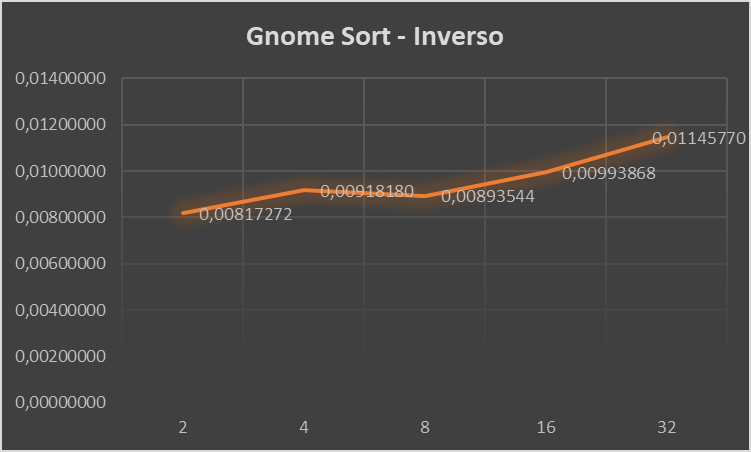
Acima na tabela 2, o algoritmo inverso, com menor tempo de execução obtido, ou seja, melhor algoritmo, foi o que utilizou 32 threads na região paralela, com um índice de speedup de aproximadamente 0,01145770. Já o algoritmo com 2 threads na região paralela foi o que apresentou maior índice, obtendo um speedup de 0,00817272 aproximadamente.

Abaixo nos gráficos 1 e 2, pode se observar bem os resultados. No gráfico 1, temos o algoritmo com 4 threads como melhor resultado obtido e com 8 threads o pior desempenho com a entrada aleatória.



**Gráfico 1: resultado speedup obtido com entrada aleatória. Fonte: Elaborado pelo autor, 2019.**

No gráfico 2, observa-se o resultado com 32 threads como o melhor desempenho na execução do algoritmo. Já o pior resultado é com 2 threads, que teve um desempenho inferior aos demais.



**Gráfico 2: resultado speedup obtido com entrada inversa. Fonte: Elaborado pelo autor, 2019.**

Os algoritmos e códigos podem ser encontrados no github. Disponível pelo link https://github.com/joiceramos/SortAlgorithm.

# 8. Conclusão

O crescente avanço tecnológico dos últimos anos e o desenvolvimento do processamento paralelo para a computação científica, interferiram diretamente no desenvolvimento de API’s de paralelização. Esse trabalho, teve como objetivo estudar o algoritmo Gnome Sort utilizando uma biblioteca de paralelização. Onde que feito um cálculo, o Speedup, que é uma métrica de desempenho, que permitiu observar a utilização da biblioteca OpenMp realizada no algoritmo é válida e traz um desempenho melhor em relação ao algoritmo utilizado sem esse complemento.

# Referências

GRUNE, DICK. **Gnome Sort - The Simplest Sort Algorithm**. Disponível em: <https://dickgrune.com/Programs/gnomesort.html>. Acesso em: 21 maio 2019.

KANASZ, ROBERT. **Gnome Sort.**2010. Disponível em: <https://kanasz.blogspot.com/2010/12/gnome-sort.html>. Acesso em: 21 maio 2019.

SUDHAKAR, SUMIT. **Gnome Sort.**Disponível em: <https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/>. Acesso em: 21 maio 2019.

CASAIS, FERNANDA; JANINE. **Ordenação de Vetores Gnome Sort.**2008. Disponível em: <https://pt.slideshare.net/f2008/ordenao-de-vetores-gnome-sort>. Acesso em: 21 maio 2019.

MARCONATTO, M.A.; **Sumarização da especificação OpenMP em linguagens de programação: Desempenho e maturidade**. 2017. 70 folhas. Trabalho Técnico Científico de Conclusão de Curso (Graduação em Ciência da Computação) - Instituto Federal Catarinense, Rio do Sul. 2017.

SILVA, M. O. **Controle de granularidade de tarefas em OpenMP**. 2011. f66. Monografia (Bacharelado, Ciência da Computação) - Universidade Federal do Rio Grande do Sul Instituto de Informática Curso de Bacharelado em Ciência da Computação – Porto Alegre, 2018.

MAILLARD, N; CERA, M. C. Desenvolvendo Aplicações OpenMP. 2010. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2010/006.pdf>. Acesso em: 14 jun. 2019.

SEBESTA, R. W. Conceitos de linguagens de programação. Porto Alegre: Bookman, 2003.

BONAT, A; GIUSTI, F. Análise de compiladores com suporte a OpenMP. 2008. Disponível em: <http://www.lbd.dcc.ufmg.br/colecoes/erad-rs/2008/0037.pdf>. Acesso em: 14 jun. 2019.