Resolução do Problema das *n*-Rainhas com Programação Paralela

Matheus Lorenzato Braga
E-mail: <matheus.braga@sombrio.ifc.edu.br>.
Instituto Federal Catarinense (IFC)

Ricardo de la Rocha Ladeira
E-mail: <ricardo.ladeira@blumenau.ifc.edu.br>.
Instituto Federal Catarinense (IFC)

Luciano de Azevedo Telesca Mota
E-mail: <luciano.atm@gmail.com>.
Universidade Federal de Pelotas (UFPel)
Diretor-Presidente da Parknet

Resumo

O problema das *n*-Rainhas é conhecido por suas soluções custosas computacionalmente, especialmente no que diz respeito ao tempo de execução. Pensando nisso, este trabalho apresenta uma forma simples de resolvê-lo com programação paralela, com o objetivo de introduzir este tema e demonstrar a possibilidade de obter soluções para o problema em tempo menor. O trabalho expõe, ainda, a definição do problema, as estratégias utilizadas para resolvê-lo e compara os resultados obtidos aos da solução sequencial, demonstrando que há melhoria no tempo de execução à medida que o tamanho do tabuleiro aumenta.
Palavras-chave: *n*-Rainhas, Programação Paralela, Avaliação de desempenho computacional.

Introdução

Escalabilidade é uma propriedade que garante que o recurso avaliado deve estar preparado para lidar com o crescimento do volume de trabalho. Ela é um fator importante em aplicações que precisam responder requisições em tempo real e que demandam alta disponibilidade, por exemplo, grande parte das aplicações de empresas de telecomunicações. Na Computação, é comum buscar soluções com melhor desempenho focando em escalabilidade. Esta melhora de desempenho pode ser obtida de várias maneiras, entre elas: otimizações no código-objeto, alterações na lógica do algoritmo, aspectos de infraestrutura e escolha de estruturas de dados.

A Programação Paralela é uma forma de computação que pode contribuir com a escalabilidade de aplicações e proporcionar ganho de desempenho em conjunto com os artifícios citados, em contraste à Programação Sequencial. O paralelismo computacional baseia-se em princípios como granularidade, localidade, balanceamento de carga e sincronização, e é considerado um modelo de programação mais difícil em relação ao sequencial (Yelick, 2013; Heroux, Raghavan, & Simon, 2006).

Neste trabalho busca-se identificar se é possível melhorar o tempo de execução de um conhecido problema computacional, intitulado *problema das n-Rainhas*, utilizando paralelismo, em um computador pessoal com processador de dois *cores*. Para isso, executou-se e comparou-se uma

DOI: 10.18256/2359-3539/reit-imed.v3n2p41-47

solução sequencial a uma implementação paralelizada com duas *threads* variando o valor de *n* (dimensão do tabuleiro) entre *1* e *12*, inclusive.

O documento está organizado em seis seções, iniciando por esta introdução. Na seção seguinte está formalizado o problema das *n*-Rainhas. Na terceira seção estão os comentários sobre Computação Paralela. A quarta seção discute o método de trabalho, contendo as estratégias utilizadas para as soluções sequencial e paralela para o problema e configurações do experimento. A quinta seção apresenta os resultados obtidos e a sexta seção apresenta as conclusões e as ideias para trabalhos futuros.

O Problema das *n*-Rainhas

O problema das *n*-Rainhas (em inglês, *n-Queens problem*) é um problema clássico de combinação na área de Inteligência Artificial (Sosič & Gu, 1990). Ele faz alusão à alocação de rainhas no jogo de xadrez, e consiste em posicionar *n* rainhas em um tabuleiro *n*×*n* de forma que nenhuma delas possa ser atacada pelas outras. Em outras palavras, duas ou mais rainhas não podem estar na mesma linha, coluna ou diagonal, pois a rainha pode atacar em todas estas situações.

Formalmente, o tabuleiro é definido como uma matriz *n*×*n*. Cada posição no tabuleiro é definida por um par ordenado (*i, j*), onde *i* e *j* são linha e coluna, respectivamente. Assim,

A linha *k* (onde *k = 1, 2, …, n*) consiste em todos os pares ordenados *(k, j) j = 1,...,n*.

A coluna *k* (onde *k = 1, 2, …, n*) consiste em todos os pares ordenados *(i, k) i = 1,...,n*.

Menor diagonal *k* (onde *k = 2, 3, …, 2n*) consiste em todos os pares ordenados *(i, j)* tais que *i+j = k*.

Maior diagonal *k* (onde *k = 1-n, 2-n, ..., n-1*) consiste em todos os pares ordenados *(i, j)* tais que *i - j = k*.

Supondo que $d_{i,j} = 1$ represente que há uma rainha na posição *(i, j)* e $d_{i,j} = 0$ em caso contrário, para *i* e *j* variando entre *1* e *n*, a solução para o problema pode ser obtida conforme mostra a Figura 1 (Letavec & Ruggiero, 2002):

$$Max \quad \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}$$

$$\sum_{j=1}^{n} d_{ij} \le 1 \quad \forall\, i = 1,\ldots,n$$

$$\sum_{i=1}^{n} d_{ij} \le 1 \quad \forall\, j = 1,\ldots,n$$

$$\sum_{\substack{i=1 \\ i+j=k}}^{n}\sum_{j=1}^{n} d_{ij} \le 1 \quad \forall\, k = 2,\ldots,2n$$

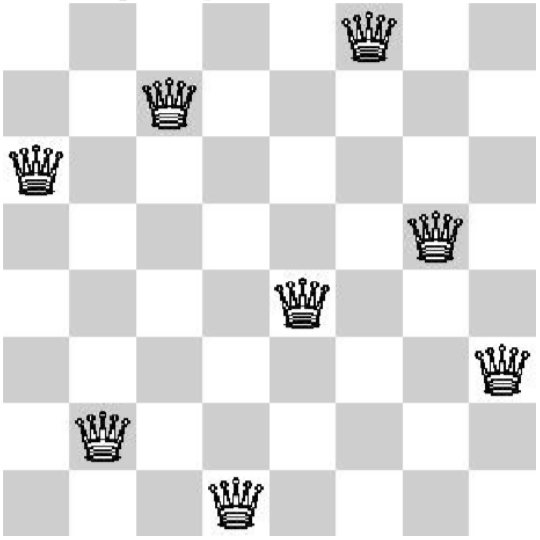
$$\sum_{\substack{i=1 \\ i-j=k}}^{n}\sum_{j=1}^{n} d_{ij} \le 1 \quad \forall\, k = 1-n,\ldots,n-1$$

$$d_{ij} \in (0,1) \quad \forall\, i,j = 1,\ldots,n$$

Figura 1 – Formalização matemática da solução do problema das n-Rainhas. (Fonte: Letavec & Ruggiero, 2002)

O problema das *n*-Rainhas possui soluções para $n \neq 2$ e $n \neq 3$. Em geral, o objetivo é encontrar todas as combinações possíveis onde não há ataques. A Figura 2 exemplifica uma resposta válida para alocação das rainhas em um tabuleiro com *n = 8*, pois nenhuma rainha está na mesma linha, coluna ou diagonal.

Figura 2 – Uma das soluções válidas para o n-Queens problem em um tabuleiro 8×8. (Fonte: http://www.billthelizard.com/2011/06/sicp-242-243-n-queens-problem.html)



Existem diversas abordagens para resolver o problema, por exemplo, a de força bruta e a de *backtracking* (Erickson, 2014), as quais apresentam complexidade exponencial (Bridge, s.d.). Há tam-

bém soluções que prometem resolver o problema em tempo polinomial com base em meta-heurísticas (Sosič & Gu, 1990; Martinjak & Golub, 2007).

Computação Paralela

Embora os computadores atuais entreguem respostas cada vez mais rápidas, é crescente também o volume de processamento gerado nas máquinas, já que as aplicações também se tornaram mais complexas e demandam grande quantidade de recursos. Com isso, não basta utilizar máquinas mais rápidas; é importante também utilizar algoritmos mais eficientes e explorar a computação paralela.

A computação paralela consiste no "uso de diversas unidades de processamento ou computadores para a resolução de um problema em comum" (Galante, 2013). Assim, o uso de arquiteturas *multicore* pode contribuir na diminuição do tempo de execução de diversos algoritmos.

Os grandes desafios da programação paralela são a granularização e a sincronização. Granularidade diz respeito à decomposição da tarefa em subtarefas, e com isso mudar a forma de pensar em soluções, já que o ser humano é frequentemente treinado para pensar de forma sequencial. É necessário ainda garantir que o recurso necessário em um dado momento está atualizado, não gerando valores incorretos por falha de sincronização. Para este problema, algumas soluções são os monitores, os semáforos e os *mutexes,* utilizados na solução proposta neste trabalho.

Os monitores são contêineres que fazem o controle de acesso aos processos, indicando ao programador quando estes devem esperar ou executar. Os semáforos são estruturas de dados com filas de descritores de processos, visando à organização destes, e têm seu funcionamento semelhante ao dos semáforos do trânsito. Os *mutexes* são construções simples para implementar exclusão mútua em linhas de produção (*threads*).

Toda solução de sincronização deve resolver o problema de *acesso à seção crítica*. Segundo Moreno (s.d.), a seção crítica é um bloco de código que designa acesso a algum recurso compartilhado, e por este motivo é dita *crítica*. É necessário garantir que todo acesso que uma *thread* realiza a esta seção inicia e finaliza sem interrupções.

A sincronização desencadeia ainda um outro desafio: o uso de cache de memória. Esta traz à computação paralela a dificuldade em garantir que um valor não será armazenado mais de uma vez em locais diferentes. Para hierarquias de cache paralelas, sugere-se a leitura de (Blelloch, 2010).

O balanceamento de carga é uma técnica de equilíbrio de carga de trabalho entre mecanismos computacionais. Na programação paralela, o balanceamento de carga é uma preocupação comum para otimizar o uso de recursos, evitando ociosidade e sobrecarga, e diminuir o tempo de execução das aplicações. Trabalhos recentes envolvem, por exemplo, otimização do balanceamento de carga e da localidade de dados com programação (Wang, Zhou, Li, Zhao, Lang & Raicu, 2014) e *frameworks* de balanceamento de carga paralela para decomposição ortogonal de dados geométricos (Magalhães, Tauheed, Heinis, Ailamaki & Schürmann, 2016).

Outros estudos recentes na área envolvem o uso da Computação Paralela em Nuvem (Ekanayake, Qiu, Gunarathne, Beason & Fox, 2010), criação de linguagens de domínio específico para aplicações paralelas (Janjic et al., 2016), processamento digital de imagens (Olmedo, De La Calleja, Benitez, & Medina, 2012) e otimização de algoritmos para *Big Data* (Facchinei, Sagratella & Scutari, 2014).

Método

Neste trabalho, o *n-Queens* foi implementado em C++ de modo sequencial para que depois se construísse a versão paralelizada. Os códigos desenvolvidos estão disponíveis em documentos suplementares. Na sequência do capítulo estão detalhadas a solução sequencial e a solução paralelizada.

Solução Sequencial

A solução sequencial foi desenvolvida com a abordagem de força bruta, de modo que fossem testadas todas as configurações de posições possíveis e descartadas aquelas que não eram soluções válidas, alocando uma rainha por coluna. Essa solução foi escolhida por ser facilmente paralelizável.

O programa sequencial foi chamado de `seqnqueens`, e era executado informando na linha de comando o número de rainhas (parâmetro `N`) – e, consequentemente, o tamanho do tabuleiro.

A saída era dada informando a cada linha uma solução possível, exibindo o posicionamento de cada rainha, conforme exposto abaixo:

Entrada: ./seqnqueens N

Saída:

Pos_Rainha_1 Pos_Rainha_2 … Pos_Rainha_N
Pos_Rainha_1 Pos_Rainha_2 … Pos_Rainha_N
Pos_Rainha_1 Pos_Rainha_2 … Pos_Rainha_N
[...]
Pos_Rainha_1 Pos_Rainha_2 … Pos_Rainha_N
Total de soluções: x

Onde N deve ser um valor inteiro positivo, x será o total de soluções encontradas e Pos_Rainha_i será a posição (a linha) em que se encontra a rainha alocada na *i-ésima* coluna. A saída é armazenada em um arquivo de texto, no caso, SO.txt.

A força bruta foi feita utilizando uma lista de vetores chamada de filaTarefas, que guardou vetores com as configurações de posições das rainhas no tabuleiro. Inicialmente, pegava-se a primeira configuração (que seria um mapa em branco), como no exemplo 4×4 que segue:

o o o o
o o o o
o o o o
o o o o

Um teste verificava se o mapa estava completo, com todas rainhas. Em caso negativo, criava todas as posições possíveis para a primeira rainha, na primeira coluna (expandia), ou seja, partindo de um estado anterior, gerava-se todas as possibilidades para posicionar a próxima rainha, testando antes se não haveria ataque, e armazenava-se na filaTarefas:

R o o o	o o o o	o o o o	o o o o
o o o o	R o o o	o o o o	o o o o
o o o o	o o o o	R o o o	o o o o
o o o o	o o o o	o o o o	R o o o

Depois o *loop* continuava testando o novo primeiro caso. No exemplo acima, como o tabuleiro não está com todas as rainhas, abre então todas as possibilidades de posicionamentos de rainhas na segunda coluna que não geram ataque e coloca na filaTarefas:

R o o o	o o o o	o o o o	o o o o	R o o o	R o o o
o o o o	R o o o	o o o o	o o o o	o o o o	o o o o
o o o o	o o o o	R o o o	o o o o	o R o o	o o o o
o o o o	o o o o	o o o o	R o o o	o o o o	o R o o

Depois disso, a configuração expandida era descartada da filaTarefas:

o o o o	o o o o	o o o o	R o o o	R o o o
R o o o	o o o o	o o o o	o o o o	o o o o
o o o o	R o o o	o o o o	o R o o	o o o o
o o o o	o o o o	R o o o	o o o o	o R o o

O processo era refeito. Quando uma configuração do tabuleiro alcançasse *n*-Rainhas, então a solução era válida. Esta solução era, então, armazenada no arquivo de texto e o contador global contendo o número de soluções era incrementado.

Solução Paralelizada

Para paralelizar a solução sequencial desenvolvida, a técnica utilizada foi a exclusão mútua (*mutex*), utilizando duas *threads*. O *mutex* serve para controlar o acesso às variáveis constantes na chamada *seção crítica*, que, no caso deste problema, são o vetor tabuleiro e a lista filaTarefas. Quando uma das duas *threads* fizesse acesso a estas variáveis, este acesso estaria entre as funções lock e unlock do *mutex*. A Figura 3 mostra a execução do programa com *n = 8* e o arquivo de saída SO.txt.

O programa foi chamado de parnqueens, e era executado informando na linha de comando o número de rainhas (parâmetro N), assim como na versão sequencial. A saída era dada informando a cada linha uma solução possível, exibindo o posicionamento de cada rainha.

Cenário de Execução do Experimento

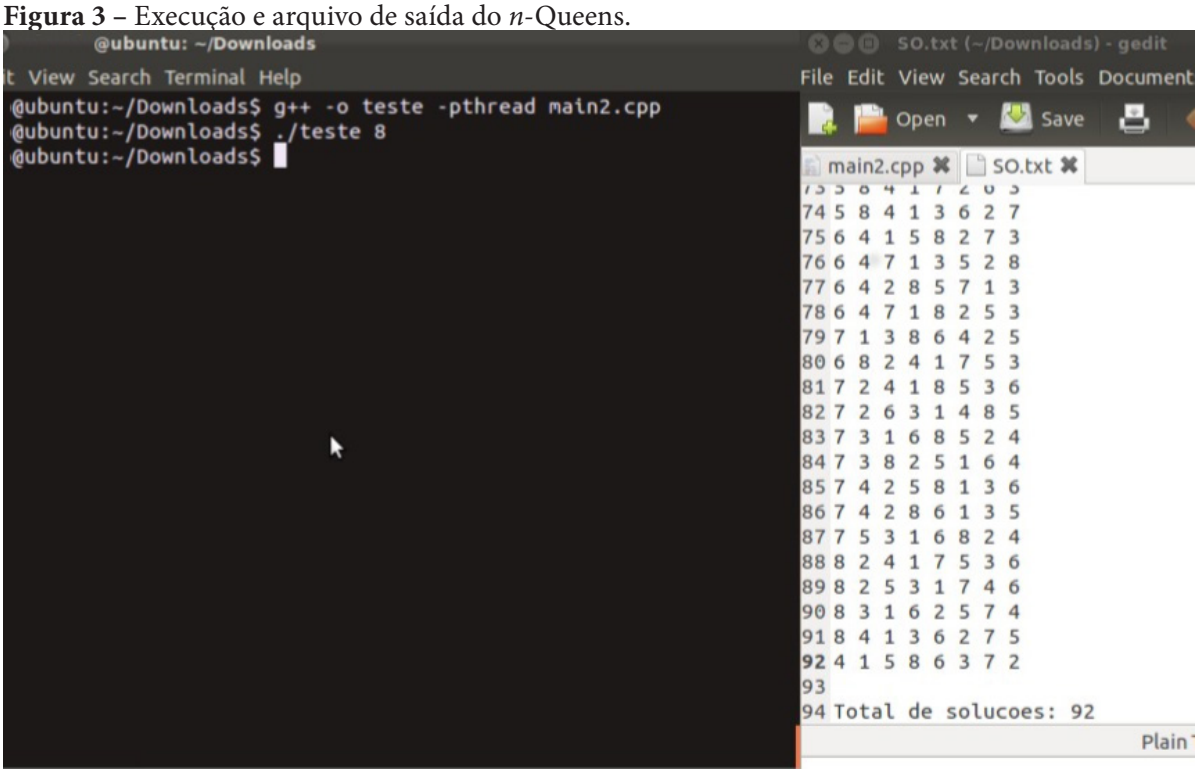
As duas abordagens foram executadas dez vezes para cada valor de *n*. Neste trabalho, os testes foram realizados variando o valor de *n* entre 1 e 12. O tempo de execução foi medido com a ferramenta time.

O computador utilizado nos experimentos contém a seguinte configuração:

- Sistema Operacional: Ubuntu 14.04 LTS 64 bits;
- Processador: Intel i3;
- Modelo: 2310M;
- Frequência: 2.10 GHz x 4; e
- Memória RAM: 7,7 GiB e 800 MHz.

O modelo de processador deste computador (Intel i3) é composto por dois *cores* e quatro *threads*. Isto significa dizer, de forma resumida, que o processador é composto por dois núcleos físicos, acrescidos de dois núcleos virtuais simulados por uma unidade denominada *hyper-threading*. No entanto,

Figura 3 – Execução e arquivo de saída do *n*-Queens.



a solução paralelizada apresentada e os testes realizados consideram apenas o uso de duas *threads*.

Resultados Obtidos

Todas as soluções testadas foram obtidas em menos de dois minutos para tamanhos de tabuleiro variando de 1 até 11. A tabela de avaliação de desempenho (Tabela 1) apresenta tempo médio e desvio padrão das duas soluções e o *speedup* da solução paralela em relação à sequencial.

O tempo de execução com duas *threads* foi consideravelmente menor se comparado ao sequencial, à medida que o tamanho do tabuleiro cresceu. No entanto, a aceleração da solução paralela apresentada não é linear e o tempo de execução cresce exponencialmente. Para valores de *n* pequenos, o custo computacional de utilizar *threads* foi maior em alguns casos. Considerando tabuleiros de xadrez de tamanho usual ($n = 8$), o tempo de execução médio da versão paralelizada foi cerca de 6,4% inferior ao da versão sequencial.

É possível notar que a variabilidade dos valores cresce mais na solução sequencial, exceto para $n = 10$, onde a solução paralelizada apresentou maior variabilidade nos testes executados. Os zeros constantes nas colunas de Desvio Padrão na Tabela 1 ocorreram quando não houve variabilidade, ou seja, os dez testes apresentaram o mesmo tempo de execução.

O campo preenchido com hífen (-) na Tabela 1 indica que o valor não foi obtido em menos de uma hora, no caso da execução do algoritmo sequencial. Os campos preenchidos com asterisco (*) indica que os valores não puderam ser calculados.

Conclusões e Trabalhos Futuros

Este trabalho apresentou e comparou duas soluções do problema das *n*-Rainhas, sendo uma sequencial e outra paralelizada. Observou-se que nos testes realizados a versão paralelizada obteve ganho de desempenho a partir de valores de *n* maiores que 7, se comparada à solução de força bruta sequencial. Para tabuleiros de tamanho usual, isto é, com $n = 8$, o tempo de execução foi cerca de 6,4% menor tomando como base o tempo médio de execução. No entanto, faz-se necessário realizar experimentos em outras arquiteturas, incluindo desde computadores com um núcleo a computadores com vários núcleos.

Tabela 1 - *Tabela de avaliação de desempenho*

n	Tempo médio sequencial	Desvio padrão sequencial	Tempo médio com 2 processadores	Desvio padrão com 2 processadores	*Speedup*
1	0,005s	0,000s	0,005s	0,000s	1,000
2	0,005s	0,000s	0,005s	0,000s	1,000
3	0,005s	0,000s	0,005s	0,000s	1,000
4	0,005s	0,001s	0,006s	0,001s	0,833
5	0,006s	0,002s	0,007s	0,001s	0,857
6	0,006s	0,002s	0,007s	0,001s	0,857
7	0,016s	0,004s	0,016s	0,003s	1,000
8	0,047s	0,007s	0,044s	0,006s	1,068
9	0,242s	0,022s	0,164s	0,031s	1,476
10	2,874s	0,386s	1,589s	0,401s	1,809
11	102,985s	4,286s	59,724s	2,111s	1,724
12	-	*	1769,028s	54,083s	*

Entende-se que a execução de mais testes deve ser realizada para validar o experimento, além de ser importante executar os algoritmos para valores de *n* maiores. Estes experimentos são ideias de trabalhos futuros, bem como a execução do algoritmo das *n*-Rainhas paralelizado utilizando mais *threads*, testando este em cenários com outras dimensões. Pretende-se ainda aplicar os mesmos conceitos utilizados neste trabalho para a execução de outros problemas clássicos da Computação, tais como o *problema do cavalo* (*knight's tour problem*), o *8-puzzle* e o *problema do caixeiro viajante* (*travelling salesman problem*).

Referências

Blelloch, G. (2010). *Algorithms for Parallel Cache Hierarchies*. Carnegie Mellon University. MPI, 2010.

Bridge, D. (s.d). *Lecture 25: Algorithms with Exponential Complexity*. Recuperado de http://www.cs.ucc.ie/~dgb/courses/toc/handout25.pdf.

Erickson, J. (2014). *Lecture 3: Backtracking.* Algorithm's Class Notes. Recuperado de http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/03-backtracking.pdf.

Heroux, M. A., Raghavan, P., & Simon, H. D. (Eds.). (2006). Parallel processing for scientific computing (Vol. 20). SIAM.

Martinjak I. & Golub M. (2007). Comparison of Heuristic Algorithms for the N-Queen Problem. *29th Int. Conf. on Information Technology Interfaces*, Cavtat, Croatia. Recuperado de http://www.zemris.fer.hr/~golub/clanci/iti2007.pdf.

Galante G. (2013). *Computação Paralela: uma visão geral.* Recuperado de http://www.inf.unioeste.br/pet-comp/wp-content/uploads/2013/08/parallel-pdf.pdf.

Ekanayake J., Qiu X., Gunarathne T., Beason S. & Fox G. (2010). High Performance Parallel Computing with Cloud and Cloud Technologies. In *Cloud Computing and Software Services.* doi: 10.1201/EBK1439803158-c12, pp. 275-308. Recuperado de http://grids.ucs.indiana.edu/ptliupages/publications/cloud_handbook_final-with-diagrams.pdf.

Facchinei F., Sagratella S., & Scutari, G. (2014). *Parallel Algorithms for Big Data Optimization*. Recuperado de http://www.optimization-online.org/DB_FILE/2014/04/4304.pdf.

Janjic V., Brown C., MacKenzie K., Danelutto K., Aldinucci M. & Garcia J. D. (2016). RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications. *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* pp. 288-295. Heraklion: IEEE. doi: http://dx.doi.org/10.1109/PDP.2016.122. Recuperado de http://calvados.di.unipi.it/storage/paper_files/2016_pdp_rpl.pdf.

Letavec, C., & Ruggiero, J. (2002). The n-queens problem. *INFORMS Transactions on Education*, 2(3), 101-103.

Magalhães, B. R., Tauheed, F., Heinis, T., Ailamaki, A., & Schürmann, F. (2016). An Efficient Parallel Load-Balancing Framework for Orthogonal Decomposition of Geometrical Data. In: *International Conference on High Performance Computing* (pp. 81-97). Springer International Publishing.

Moreno, E. (s.d.). *Programação Concorrente: sincronização entre processos*. Sistemas Operacionais. Slide. Pontifícia Universidade Católica do Rio Grande do Sul.

Olmedo E., de la Calleja J., Benitez A. & Medina M. A. (2012). Point to point processing of digital images using parallel computing. *IJCSI International Journal of Computer Science Issues*, Vol. 9, Issue 3, No 3. Recuperado de http://ijcsi.org/papers/IJCSI-9-3-3-1-10.pdf.

Sosič R., & Gu J. (1990). A Polynomial Time Algorithm for the N-Queens Problem. *SIGART Bulletin*, Vol. 1, 3, pp. 7-11. Recuperado de http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=6ABB14FA91C79BEE6CEEAB0859DE51E1?doi=10.1.1.57.4685&rep=rep1&type=pdf.

Wang, K., Zhou, X., Li, T., Zhao, D., Lang, M., & Raicu, I. (2014). Optimizing load balancing and data-locality with data-aware scheduling. In Big Data (Big Data), 2014 IEEE International Conference on (pp. 119-128). IEEE.

Yelick, K. (2013). *CS 194 Parallel Programming Why Program for Parallelism?* CS194 Lecure. 5/12/2013. Recuperado de http://facultymembers.sbu.ac.ir/fazlali/old/Downloads/Course4/slides/Multicore01.pdf.

Abstract

The *n*-queens problem is well-known for its computationally costly solutions, especially with regard to the execution time. Thinking about it, this paper presents a simple way of solving it with parallel programming, in order to introduce this matter and demonstrate the possibility of finding solutions to this problem in a shorter time. The paper also describes problem definition, strategies used to solve it and compares the results obtained with the sequential solution, demonstrating that there is an improvement in execution time as the size of the board increases.
Keywords: *n*-Queens, Parallel Computing, Computational performance evaluation.