Computação de Alto Desempenho

Relatório Trabalho Prático 2

Alunos:

Marco Simões, msimoes@student.dei.uc.pt

Nuno Lourenço, naml@student.dei.uc.pt

Coimbra, Maio de 2010

Índice

Introdução

No âmbito da disciplina de Computação de Alto Desempenho integrante do plano curricular do Mestrado em Engenharia Informática foi desenvolvido o seguinte trabalho com o objectivo de avaliar o desempenho de algoritmos paralelos para a coloração de grafos.

A estrutura do relatório apresenta inicialmente uma descrição do problema a analisar, seguida de uma descrição das implementações com a respectiva justificação das tecnologias usadas. Apresentamos os testes efectuados, os resultados obtidos e, finalmente, as conclusões extrapoladas.

Este trabalho permitiu-nos desenvolver as nossas capacidades de paralelismo de algoritmos, quer a nível do problema quer ao nível da implementação.

Descrição do problema

O problema passa por, dado um grafo G, composto por vértices e arestas, simbolicamente representado por G=(V,E), atribuir a cada vértice uma cor, de forma a que esta cor seja diferente da dos vizinhos desse vértice (Figura 1).

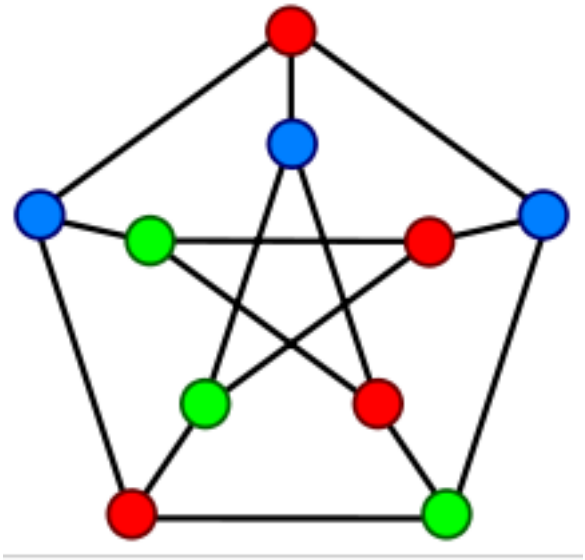

Figura 1 - Grafo de Petersen

É importante a introdução de número cromático, que é o número mínimo de cores necessário para colorir todos os nós de um grafo. Por exemplo, o número cromático da figura 1 é 3.

Implementações

Neste projecto fizemos 4 implementações paralelas. Utilizamos ainda uma implementação sequencial fornecida com o enunciado do projecto para comparação de resultados.

Tecnologia utilizada

Para a implementação de algoritmo paralelos para a coloração de grafos utilizados a tecnologia MPI (Message Passing Interface), para a transmissão de informações entre os diferentes processos.

Implementação Sequencial

Greedy

O funcionamento deste algoritmo é muito simples: vamos percorrendo o grafo por uma ordem sequencial, atribuindo a cada nó a menor cor disponível, baseado nas cores já atribuídas aos seus vizinhos. No fim retornamos o conjunto de cores resultante.

Como referido acima, este algoritmo foi-nos fornecido com a publicação do enunciado, pelo que não vamos entrar em mais detalhes sobre o seu funcionamento.

Implementações paralelas

Em todas as implementações paralelas optámos por ter um processo responsável pela leitura e apresentação dos dados ao utilizador. Este processo é ainda responsável por distribuir as tarefas por todos os outros processos que vão fazer todo o trabalho. Introduzimos assim o nosso conceito de processo Master (executado com o rank 0), coordenador dos trabalhos executados pelos diferentes processos, os Workers (com valores de rank [1, n[, sendo n o número de processos usados na execução dos algoritmos).

Greedy Paralelo

Begin

Para i = 1 até p **Faz** em paralelo

Recebe matriz de adjacências

Enquanto houverem vértices sem cor

Escolher aleatoriamente um vértice por colorir

Colorir esse vértice, tendo em base as cores já conhecidas dos vizinhos

Fim_Enquanto

Fim_Para

Verificar qual das soluções paralelas tem o menor número cromático

End

Figura 2 - Algoritmo Greedy Paralelo

Notação Utilizada:

p -> número de processadores disponíveis

Neste algoritmo o Master apenas distribui a matriz de adjacências pelos Workers e aguarda os seus resultados. Depois de os receber escolhe qual a melhor solução.

Os Workers, por sua vez, apenas recebem a matriz de adjacência e aplicam uma variação do algoritmo greedy, em que a ordem em que os nós são escolhidos é obtida aleatoriamente.

Plassman com V processos

Este algorimto está descrito no enunciado em pseudo-código, pelo que não vamos estar a detalhar o seu funcionamento.

Plassman com p processos

```
Begin
        // parte 1
        Para  i = 1 até p Faz em paralelo
                Gerar os pesos aleatórios de cada vértice
                Enviar pesos para outros processos e receber os deles
        Fim_Para


        // parte 2
        Para  i = 1 até p Faz em paralelo
                Para j = 1 até b Faz
                        Gerar lista de espera e de envio para vizinhos de outros processos
                        Se espera(j) == 0 Faz
                                Gera cor de j
                                Para destino em envia(j) Faz
                                        Envia cor de j para destino
                                Fim_Para
                        Fim_Se
                Fim_Para
                Enquanto total de espera > 0 Faz
                        Espera mensagem // traz informação de cor e vértice original
                        Guarda em cores[origem_mensagem] a cor_mensagem
```

```
                    Para j = 1 até b Faz
                        Se origem de mensagem em espera(j) Faz
                            Remove de espera(j)
                            Se espera(j) == 0 Faz
                                Gera cor de j
                                Para destino em envia(j) Faz
                                    Envia cor de j para destino
                                Fim_Para
                            Fim_Se
                        Fim_Se
                    Fim_Para
                Fim_Enquanto
                Colorir nós locais (sem vizinhos em outros processos) como no greedy
            Fim_Para
End
```

Notação Utilizada:

p -> número de processadores disponíveis

b -> número de vértices por processador (block)

Este algoritmo introduz o conceito de vértices locais e vértices globais. Os V_{locais} são vértices cujas arestas apenas ligam a vértices pertencentes ao mesmo processo. Os $V_{globais}$ são os restantes vértices, isto é, vértices que contêm arestas que ligam a vértices de outros processos.

Na primeira parte do algoritmo cada processo gera pesos aleatórios para os seus vértices e partilha-os com os restantes processos. Para isto, fazemos Broadcast para um grupo de utilizadores composto apenas pelos Workers.

Depois de termos os pesos de todos os nós, podemos verificar, à semelhança da versão anterior do mesmo algoritmo, os vértices dos quais teremos que esperar e dos quais teremos que enviar. Os vértices que não tiverem que aguardar procedem de imediato aos seus envios.

De seguida, o algoritmo aguarda a chegada de mensagens. Quando uma mensagem chega, esta traz informação sobre o vértice de origem e a sua cor. Essa informação é guardada pelo processo, que posteriormente verifica se algum dos vértices estava à espera deste. Se sim, deixa de esperar por este. Se já não tiver que esperar por mais nenhum vértice, calcula a sua cor e envia-a para os vértices na fila para envio.

No fim, as cores do bloco processado por cada processo são enviados para o Master.

Block Partition Algorithm

Este algoritmo pretende-se ser uma versão paralela, em que cada processador colora um conjunto de vértices. Assim podemos ter um número de processadores inferior ao número de vértices do grafo, que conseguimos fazer a coloração sem problemas nenhuns.

Begin

Partir V em V_1, ...V_p, onde floor(n/p)<=|V_i|<=ceil(n/p)

//Passo 1 – Atribuição de cores aos vértices

Para vértice v em V_i **Faz**

    Para i = 1 até p, **Faz** em paralelo

        Atribuir a v a menor cor possível

        Enviar cor de v para todos

    Fim_Para

Fim_Para

//Passo 2 – Resolução de conflitos para os vértices que foram coloridos no mesmo passo

Para vértice v em V_i **Faz**

    Para i = 1 até p, **Faz** em paralelo

        Para cada vizinho u de v que foi colorido no mesmo passo **Faz**

            Se color(v) == color(u) **Então**

                Guardar o min{u,v}

            Fim_Se

        Fim_Para

    Fim_Para

Fim_Para

//Passo 3 – Colorir os nós que estão em conflito sequencialmente

End

Figura 3 - Block Partition Algorithm

A notação utilizada no algoritmo é:

V -> conjunto de todos os vértices do grafo

p -> número de processadores

n -> número total de vértices

$|V_i|$ -> número total de vértices no bloco V_i

Testes

Para testar a performance dos nossos algoritmos decidimos montar uma experiência onde variamos o número de vértices que compõem os grafos. Gerámos desde grafos com valores muito pequenos (e.g. 10), até valores grandes (e.g. 4096). Os valores do número de vértices são incrementados em potências de dois, à excepção dos iniciais.

12,16,64,128,256...

Para estes números de vértices gerámos grafos de três tipos: Ring, Grid, Crown.

A análise que efectuámos para cada tipo de implementação está relacionada não só com o tempo que cada um dos algoritmos demora a executar, mas também com o número de cores que cada algoritmo necessita para colorir o grafo.

Os testes foram efectuados usando 6 dos computados da sala G56.

Resultados

Performance

As figuras 4, 5 e 6 apresentam os resultados de performance para cada algoritmo. De salientar que o algoritmo plassman_v, que usa n_processos == n_vertices, apenas corremos até aos inputs de tamanho 128. Mais que isso o algoritmo demorava demasiado tempo a correr, pois utiliza demasiados recursos.

Ainda que referir que para os grafos do tipo crown, que para cada vértice têm (v/2 -1) arestas, os algoritmos plassman com n processos e o greedy sequencial não conseguiram computar em tempo útil soluções para grafos com 4096 vértices.

De resto, verificamos que o algoritmo Greedy sequencial obtém os tempos mais rápidos para grafos ring e grid. No entanto, para grafos crown, este algoritmo têm muito má performance, ganhando significativamente o algoritmo block partition. O algoritmo plassman obriga a muitas trocas de mensagens, principalmente para grafos com muitas arestas que ligam a processos diferentes, como é o caso dos crown. Desta forma, também piora bastante a sua prestação nos

grafos crown. Para estes, a melhor solução é o block partition, obtendo melhores tempos que qualquer um dos restantes.

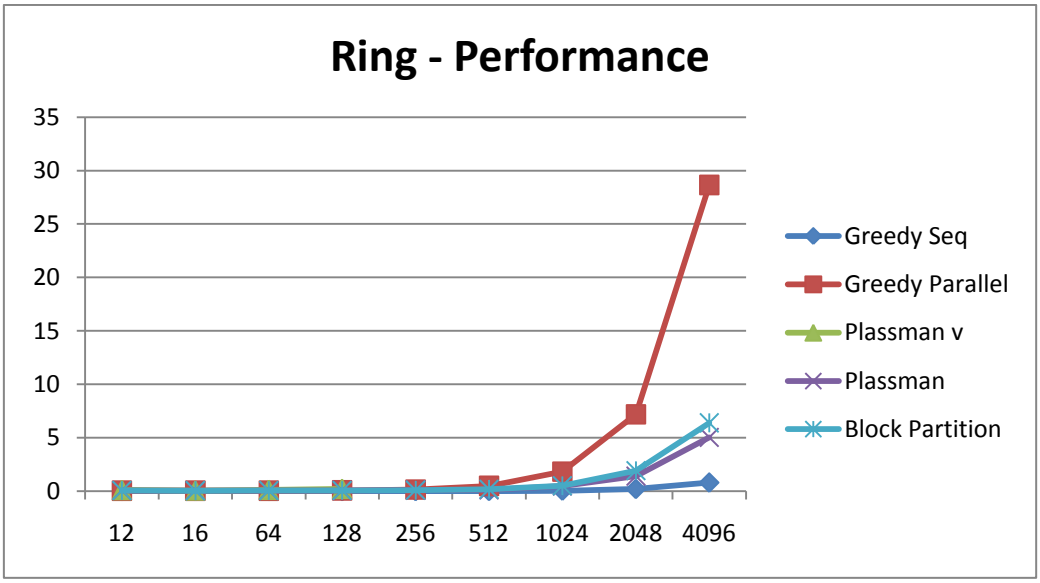

Figura 4 - Resultados para um grafo Ring

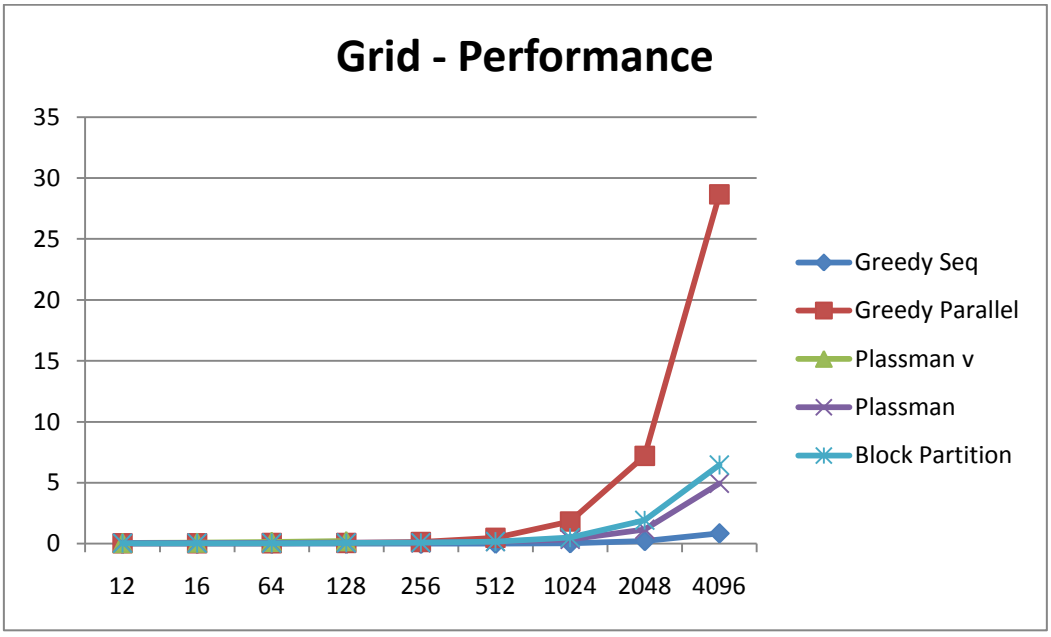

Figura 5 - Resultados para um grafo Grid

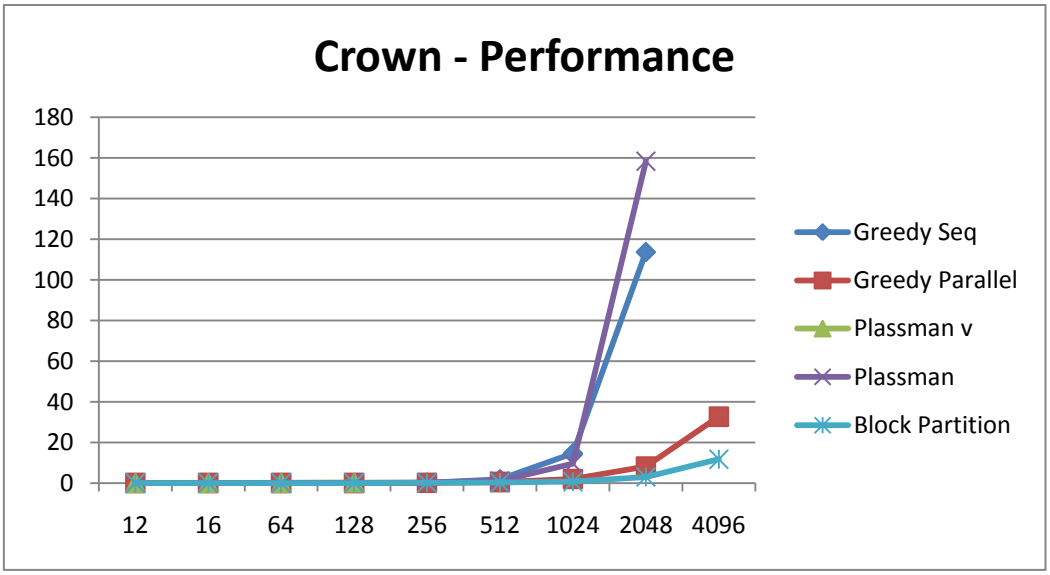

Figura 6 - Resultados para um grafo Crown

Número de Cores

O número de cores aparece como forma de medida para a qualidade dos algoritmos. No entanto, é preciso salientar que os grafos ring e grid beneficiam significativamente o algoritmo greedy sequencial. Isto porque no ring a sua aplicação resulta sempre no valor óptimo, pois todos têm um número par de vértices e começando em qualquer nó, seguindo sempre para um vizinho, este consegue o valor óptimo. Para os grids a situação é semelhante. Estes são do tipo 2^x * 2^y * 2^z = n_vertices, pelo que podem ser coloridos apenas com duas cores calculadas facilmente pela aplicação do algoritmo greedy sequencial. Podemos, no entanto, avaliar as prestações dos restantes algoritmos nestes grafos, e fazer a avaliação mais justa nos grafos tipo crown.

Podemos ver que para os grafos ring os resultados são todos muito idênticos, obtendo-se um desvio do resultado óptimo de apenas uma cor na maioria das soluções.

Para os grafos grid verificamos que os algoritmos de plassman e o greedy_parallel divergem mais do resultado com o aumento das instâncias. No entanto, esse desvio não se apresenta muito significativo. O block partition mantém resultados mais constantes com um desvio menor do resultado óptimo.

Para os grafos crown, os mais interessantes a nível de complexidade, observamos que o greedy_sequencial cresce demasiadamente com o aumento das instâncias. O block partition também piora a qualidade dos resultados ao longo das instâncias. Os melhores resultados são alcançados pelo greedy_parallel e pelo plassman. Salientar apenas que para o caso maior o algoritmo de plassman não termina.

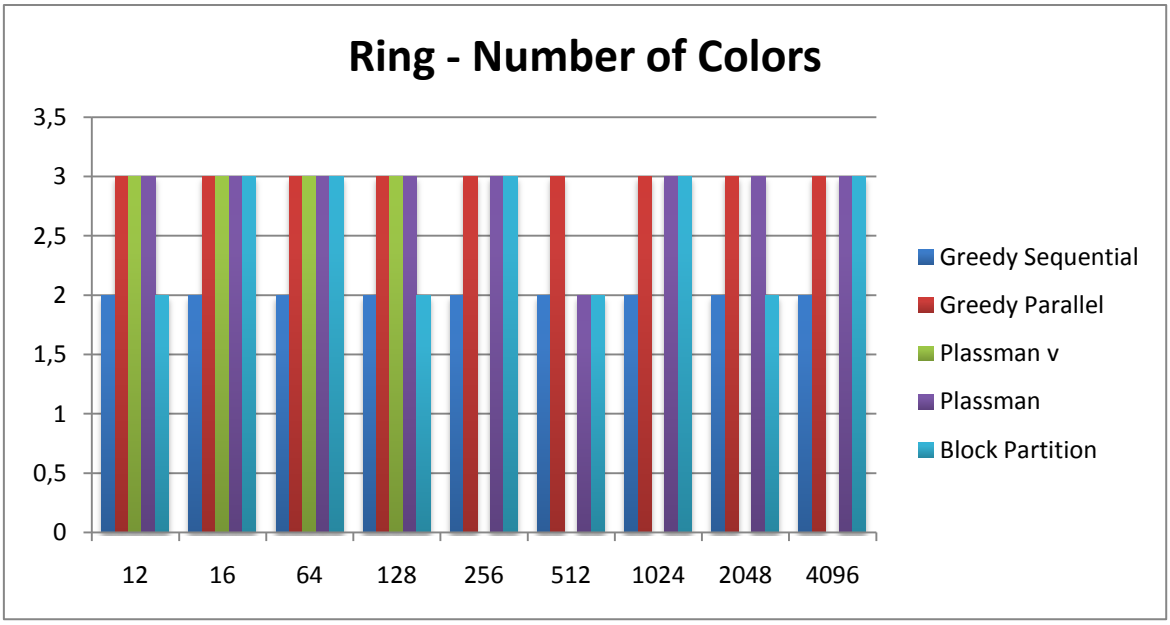

Figura 7 - Número cromático obtido por cada um dos algoritmos para um grafo Ring

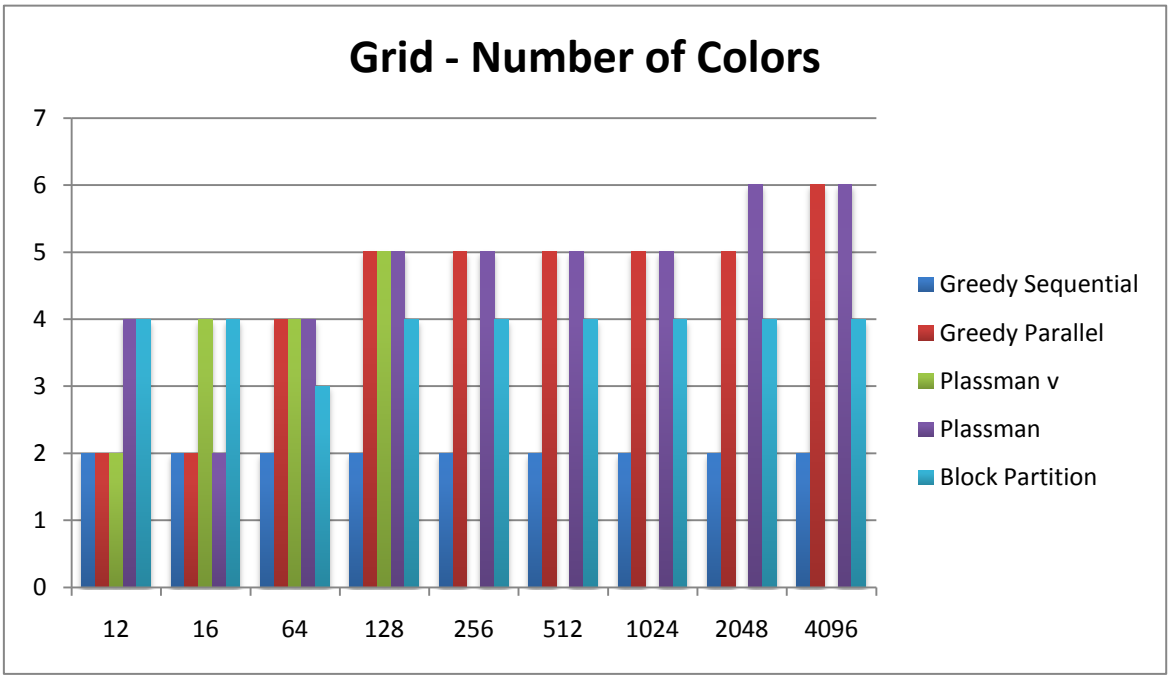

Figura 8 - Número cromático obtido para cada um dos algoritmos para um grafo Grid

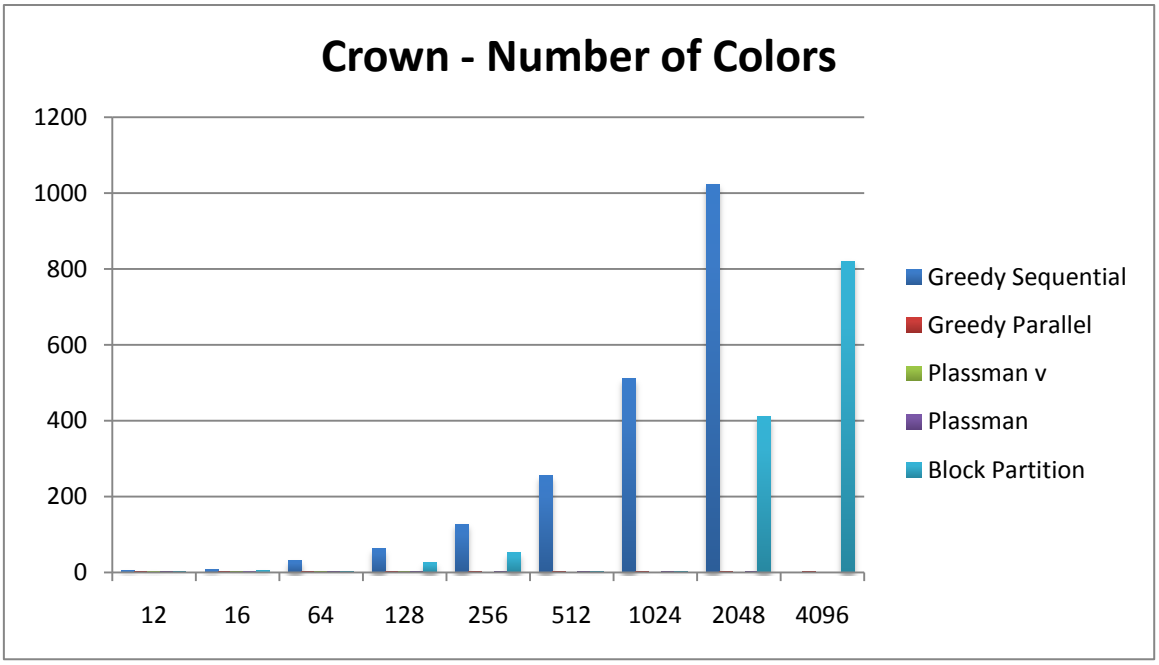

Figura 9 - Número cromático obtido para cada um dos algoritmos para um grafo do tipo Crown

Conclusão

Analisando os resultados, e fazendo uma correlação entre o tempo de execução e o número de cores obtidas, chegamos à conclusão que o algoritmo que mostra melhores indicadores é o greedy parallel, que tem tempos razoáveis e resultados bons mesmo para o caso dos crown. O block partition apresenta os melhores tempos mas para os crown gera muitos conflitos e leva a maus resultados.

Faz sentido também correlacionar as performances com as mensagens de mpi trocadas pelos algoritmos: O plassman tem a maior troca de mensagens, enquanto que o block partition tem uma troca mais leve. O greedy_parallel tem que enviar toda a matriz de adjacências no inicio, mas depois daí não tem mais trocas até devolver os dados para o master. Isto reforça os resultados apresentados para o crown, em que as trocas do plassman são demasiadas para que este tenha uma performance aceitável.

NOTA: Foram efectuados também testes com 12 processos em vez de 6, para tentar tirar partido do context switch nos casos em que os processos ficariam parados a ler da rede, por exemplo. No entanto, esta experiência revelou piores resultados para praticamente todos os testes, pelo que decidimos não incluir esses dados no relatório.