Paralelização do Phyml 3.0, Programa para Reconstrução de Árvores Filogenéticas, usando OpenMP

Martha Torres[1], Ayran Oliveira Soares Vieira[1]

[1]Departamento de Ciências Exatas e Tecnológicas – Universidade Estadual de Santa Cruz (UESC)
CEP – Ilhéus – BA – Brasil

{mxtd2000,ayran_br}@yahoo.com.br

Resumo. *Este artigo descreve a paralelização do programa PhyML 3.0 usando o paradigma de memória compartilhada através de OpenMP. Apresentam-se medidas de desempenho usando um servidor com dois processadores, cada um com quatro cores. O PhyMl é um dos principais programas utilizado para realizar reconstrução de árvores filogenéticas.*

1.Introdução

A diversidade biológica, ou seja, as diferenças entre os grupos de organismos é entendida hoje como o resultado do processo de evolução. Os seres vivos não são entidades estáticas, mas se transformam ao longo de gerações sob influência do meio. O objetivo da análise filogenética é inferir a relação correta entre três ou mais espécies contemporâneas de dados de sequência (pode ser de aminoácidos, nucleotídeos, códons, etc) de membros representativos de cada espécie [Schad et al. 1998]. A representação da história evolutiva dessas espécies ou grupos de espécies é feita através das árvores filogenéticas, onde as sequências são agrupadas e são estabelecidos os relacionamentos entre estas. Tem-se desenvolvido muitos programas para reconstrução de árvores filogenéticas (RAF), baseados em métodos de distância, máxima verossimilhança, parcimônia e analise bayesiana. Atualmente as pesquisas nesta área estão direcionadas a melhorar o desempenho, exatidão e escalabilidade dos métodos.

O programa PhyML [Guindon and Gascuel 2003], baseado no método de máxima verossimilhança, tem sido amplamente utilizado (com mais de 2300 citações no ISI Web of Science), por conta de sua simplicidade, e de uma boa relação entre eficiência e exatidão. Desde a versão 2.45 o PhyML apresenta uma versão paralela a qual foi desenvolvida utilizando MPI (Messages Passing Interface), essa paralelização foi realizada sobre a opção de bootstrap (técnica estatística clássica que dá um suporte realistico às árvores obtidas). Esta opção, quando habilitada para um número natural N, realiza N iterações de busca da árvore com maior máxima verossimilhança, a paralelização desta opção, teoricamente, permite a redução do tempo de execução de NxT1 para (NxT1) /P, onde N é o número de iterações, T1 o tempo de execução de uma iteração, e P é o número de processadores. Portanto, faz-se necessária uma paralelização que diminua o tempo de execução T1 de uma iteração. Neste trabalho apresenta-se uma versão paralela do PhyML 3.0 para diminuir este tempo de execução usando o paradigma de memória compartilhada através do OpenMP. A caraterística fundamental dos programas de RAF é seu consumo de CPU na realização de operações de ponto flutuante especificamente quando se usam métodos baseados em máxima

verossimilhança e analise bayesiana. Os cálculos são feitos de maneira acumulativa de sitio a sitio.

O resto do artigo está organizado da seguinte maneira: na seção 2. será explicada como foi feita a paralelização, na seção 4 apresenta-se os testes e resultados e finalmente na seção 5 descreve-se as conclusõe e trabalhos futuros.

2.Paralelização do PhyML

Para realizar a paralelização do PhyML, foi traçado um caminho que tornasse mais simples e eficiente o cumprimento desta tarefa. Foi feita a escolha de uma IDE de desenvolvimento que facilitasse a análise de desempenho do programa sequencial do PhyML. Para desempenhar esse papel foi escolhida a IDE (Integrated Development Enviroment) NetBeans 6.8 associada às ferramentas do Sunstudio 12.1. E através de um mecanismo de *profiling* (delineamento de perfil), proporcionado por esta associação Netbeans-Suntudio, foi delineado o perfil comportamental do programa possibilitando o reconhecimento das funções que utilizavam maior tempo de processamento.

A Figura 1 apresenta a tabela que mostrou ser a fonte de informação mais importante para detecção de gargalos dentre as tabelas e gráficos gerados com a IDE. As informações principais da tabela são o tempo de CPU exclusivo e inclusivo. Por exemplo, ao observar-se a coluna de tempo de CPU exclusivo pode-se constatar se uma determinada função concentra muitos cálculos dentro de si própria ou não, comparando o valor referente a esta função na tabela com os valores referente às outras funções na própria tabela. Se o valor do tempo de CPU exclusivo de uma função destoa muito para cima em relação a outras, isso provavelmente significa que esta função deve conter instruções que demandem muito tempo de CPU ou pode ser uma função que recebe muitas chamadas. E em ambos os casos essa função pode ser considerada um gargalo e uma paralelização de tal função é possivelmente desejável.

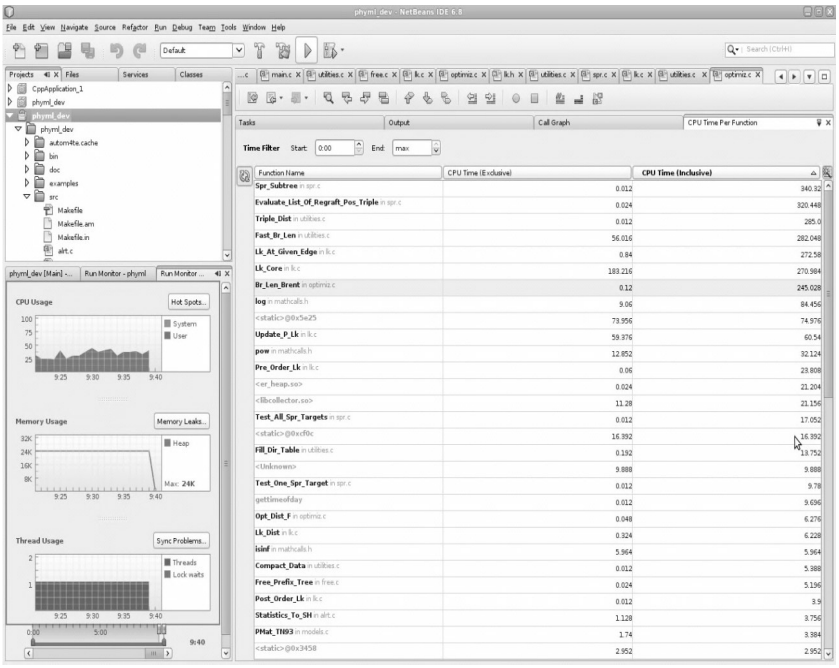

Figure 1. Profiling da SunStudio, Tempo Exclusivo é igual ao tempo total de processamento em uma determinada função. Tempo inclusivo é o tempo exclusivo mais o tempo de processamento das funções chamadas na determinada função.

O tempo de CPU inclusivo está relacionado às chamadas que uma função faz a outras funções, um valor alto de uma certa função nesta coluna em comparação com o de outras funções pode significar três coisas: Primeira, a função tem dentro de si própria instruções que tomam muito tempo de processamento. Segunda, a função chama funções que exigem muito tempo de CPU. Terceira, simplesmente a função recebe muitas chamadas durante a execução do programa. Também é interessante fazer a comparação dos valores dessas duas colunas para poder inferir conclusões com mais precisão. Há três possibilidades para os valores comparados dessas duas colunas: a coluna de tempo exclusivo ter o mesmo valor da coluna de tempo inclusivo; a coluna de tempo exclusivo ter valor ligeiramente menor que o da coluna de tempo inclusivo; a coluna de tempo exclusivo ter valor bastante inferior ao da coluna de tempo inclusivo. No primeiro caso, supõe-se que a função não efetua chamadas a outras funções; no segundo caso, admite-se que a função efetua chamadas a outra funções as quais são leves (exigem pouco tempo de processamento); finalmente no terceiro caso, conclui-se que a função efetua chamadas a funções pesadas (exigem muito tempo de processamento). Por exemplo: Lk_Core apresenta um tempo exclusivo de 183,216 segundos e tempo inclusivo de 270,84 segundos (significando que efetua chamadas a outras funções as quais consomem tempo de execução) . Update_P_Lk Len apresenta um tempo exclusivo e inclusivo de 59,36 e 60,24 segundos respectivamente (significa que praticamente não efetua chamadas a outras funções). Fast_Br_Len apresenta um tempo exclusivo e inclusivo de 56,82 e 282,32 segundos respectivamente (significa que está função consome muito tempo chamando outras funções).

Fazendo esta análise foram identificados três grandes gargalos sendo o principal a função Lk_Core que é o cerne dos cálculos de verossimilhança. Descobriu-se que esta função é chamada exaustivamente durante a execução do programa, em parte devido ao grande número de funções que indiretamente a chamam. Esta função é chamada pelas funções Lk e Lk_At_Given_Edge. E estas na sua vez são chamadas por muitas outras funções durante a execução do programa.

A função Lk_Core foi modificada de modo a permitir paralelização, primeiro tentou-se fazer uma paralelização interna que acabou não mostrando resultados satisfatórios chegando até a piorar o desempenho do programa. Analisando a tentativa frustrada da paralelização interna, percebeu-se tratar-se de um problema de granularidade. Ou seja percebeu-se que a função Lk_Core não era tão pesada, o que pesava era o fato de ser chamada muitas vezes. Partiu-se então para uma paralelização externa. E esta se deu paralelizando as chamadas à Lk_Core, estas chamadas eram feitas dentro de um laço *for* controlado por uma variável correspondente ao sítio atual. Dentre outras coisas foi aplicada a diretiva *#pragma parallel for*.

Outro importante gargalo encontrado foi a função Fast_Br_Len. Sua paralelização consistiu, entre outras adaptações, na utilização da diretiva *#pragma omp parallel* para criação e duas diretivas *#pragma omp for* dentro desta região paralela, a primeira das diretiva *#pragma omp for* foi aplicada num laço que facilmente se detecta a complexidade já que continha ainda outros três laços internos ao paralelizado. Já a segunda foi aplicado sobre um laço criado para compensar alterações necessárias à adaptação. E o último importante gargalo encontrado foi a função Update_P_Lk Len

Para esta função de modo similar às outras foi utilizada a diretiva #pragma omp parallel for em um laço controlado por variável representando sítio. Porém para essa

função houve mais trabalho para definir quais variáveis eram shared ou private.

Em resumo, foi feita a análise do código fonte e percebendo-se por que a determinada função consome tanto tempo de CPU, buscava-se possíveis trechos da função a paralelizar. Ou seja eram procurados trechos de código que fossem independentes entre si, ou por laços *for* que tivessem iterações independentes entre si. Aplicava-se então uma tentativa de paralelização, sempre buscando testes que comprovasse a viabilidade da tal tentativa, tendo como viável uma paralelização que não comprometesse a lógica global do código sequencial e que apresentasse um avanço no sentido do desempenho e economia de tempo.

3.Teste e Resultados

Os testes foram realizados em um computador multicore, com dois processadores de quatro cores e tecnologia HT (Hyperthreading) com 2.4 GHz de velocidade de clock e 24159 MB de memória RAM. Foram testadas 12 arquivos de sequências de entrada de dimensões (número de taxa e número de sítios) variadas, todas as sequências eram de nucleotídeos e foram executadas com as opções mostradas na Tabela 1.

Tabela 1. Opções usadas nos testes realizados

Opção	*Significado*
"-d nt"	*Entrada é de nucleotídeo*
"-m GTR"	*Modelo de substituição a ser usado é o GTR*
"-f 0.19, 0.34, 0.31, 0.36"	*São as frequências respectivas de A,C,G e T*
"-t 2.32"	*Proporção de transição/transversão é 2.32*
"-v e"	*Proporção de sites invariáveis é estimada*
"-a 0.89"	*Parâmetro de forma de distribuição gama é 0.89*
"-o tlr"	*A árvore,os comprimentos, e os parâmetros de proporção são otimizados*
"-s SPR"	*A opção de busca de topologia de árvore é SPR.*

A Tabela 2 mostra os tempos em segundos das execuções do programa com número variado de threads. Da análise da tabela percebe-se um pequeno slow down (desaceleração) ao passar de quatro para cinco threads, este se deve muito provavelmente ao aumento do tempo de comunicação entre as threads, já que com quatro threads garante a execução num mesmo processador, mas ao se usar cinco threads obrigatoriamente há de se usar os dois processadores. Resumindo: o ganho no acréscimo de uma thread, no caso especial de quatro para cinco threads, não foi suficiente para compensar o aumento do tempo necessário para comunicação das threads em processadores diferentes. A partir da sexta thread já volta a ser vantajoso mesmo com o aumento no tempo de comunicação devido ao uso de dois processadores diferentes, entretanto o aumento de desempenho é pouco significativo depois desta thread.

Tabela 2. Opções usadas nos testes realizados

	Seq12_3768	Seq20_3768	Seq29_3768	Seq42_3768	Seq55_3768	Seq62_0600	Seq62_1200	Seq62_1800	Seq62_2400	Seq62_3000	Seq62_3768
Sequencial	31,20	90,20	194,60	296,60	403,60	148,60	306,20	394,60	358,40	429,60	911,80
2 Threads	23,60	71,20	155,80	233,40	310,80	117,60	237,00	306,40	278,60	341,20	684,00
3 Threads	17,00	50,60	111,00	166,00	219,00	83,80	168,40	216,80	198,60	244,80	496,80
4 Threads	14,50	45,25	89,50	141,75	177,25	68,00	136,25	175,00	161,25	195,67	396,00
5 Threads	15,00	42,25	94,00	141,00	184,75	72,75	143,25	184,00	169,00	205,75	409,50
6 Threads	13,00	37,00	81,75	123,25	163,75	64,25	127,50	161,75	150,50	178,75	363,75
7 Threads	12,50	35,50	77,00	120,25	156,75	61,75	121,25	153,50	140,75	168,75	344,50
8 Threads	12,25	35,25	76,50	114,75	151,75	61,50	120,50	150,00	137,75	168,00	342,20

Também foram feitos testes com bootstrap. Estes testes foram realizados com as opções padrão exceto pela opção de bootstrap “-b 5” utilizando 5 réplicas. Os resultados para uma sequência de nucleotídeo com 62 taxa e 3768 sítios são mostrados na Tabela 3. Neste caso percebe-se que o ganho em desempenho é maior que na situação anterior o que é um comportamento desejável dado que os resultados das árvores filogenética sempre vem acompanhadas da análise de bootstrap.

Tabela 3. Resultados. Tempos (min) significa tempos em minutos

	Tempo (horas)	Speedup	Eficiência	Tempos(min)
Serial	0,832	1,000	1,000	49,933
2 Threads	0,522	1,594	0,797	31,333
3 Threads	0,411	2,026	0,675	24,650
4 Threads	0,351	2,370	0,593	21,067
5 Threads	0,311	2,680	0,536	18,633
6 Threads	0,281	2,966	0,494	16,833
7 Threads	0,260	3,204	0,458	15,583
8 Threads	0,238	3,504	0,438	14,25

4.Conclusões e Trabalhos Futuros

Foi implementada uma versão paralela, utilizando o paradigma de memória compartilhada, através do OpenMP do programa PhyMl 3.0 o qual é de grande relevância na área de inferência filogenética. Pela análise de desempenho conclui-se que o tempo de execução diminuiu ao aumentar o número de threads embora percebeu-se aumento no overhead de comunicação. Isto nós leva a propor novas alternativas de paralelização como usar GPU para explorar a paralelização de granularidade fina. Atualmente estão sendo preparados testes para avaliar o desempenho da versão hibrida MPI e OpenMP na qual a parte MPI é usada para implementar a paralelização do bootstrap.

References

Schadt, E., Sinsheimer, J. and Lange, K. (1998) “Computational Advances in Maximum Likelihood Methods for Molecular Phylogeny”. In: Genome Research. v. 8, p. 222-233

Guindon, S. and Gascuel, O. (2003) “A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood” . In: Syst. Biol., v. 52, n.5, p.696-704