Avaliação de desempenho de um núcleo de execução multithread

Elvio Vicosa Junior*, Gerson Geraldo H. Cavalheiro

Departamento de Informática
Universidade Federal de Pelotas
Pelotas – RS – Brasil
{evicosa_ifm,gersonc}@ufpel.edu.br

Abstract. Athreads é uma ferramenta de programação multithread que implementa parte da especificação POSIX para threads (Pthreads) e que dispõe de um núcleo de execução que explora um algoritmo de escalonamento baseado no controle do fluxo de dados entre threads. Neste artigo é apresentado um mecanismo de coleta de lixo, que estende a especificação Pthreads em Athreads para permitir que um thread sofra múltiplas sincronizações por *join*. A utilização deste mecanismo é avaliada experimentalmente, sendo avaliado o impacto da utilização deste mecanismo no tempo de execução e o ganho de memória que ele oferece durante a execução de programas.

1 Introdução

Com a popularização de processadores multi-core houve um aumento de ofertas de multi-processadores (tanto arquiteturas UMA quanto NUMA) no mercado. Uma das questões a serem tratadas no desenvolvimento de aplicações para tais arquiteturas é como obter uma implementação que seja ao mesmo tempo eficiente e escalável com o número de processadores disponíveis, sem haver a necessidade de modificações na implementação. Uma alternativa é utilizar mecanismos de escalonamento que explorem eficientemente os recursos do *hardware* disponível [1]. Esta abordagem foi utilizada na concepção do modelo Anahy [2].

Anahy é um modelo de execução concorrente baseado no modelo de fluxo de dados. Este modelo emprega algoritmos de lista [4] para realizar o escalonamento em arquiteturas multi-processadas. Um protótipo operacional de Anahy encontra-se implementado em Athreads [6, 5]. Este protótipo disponibiliza uma interface de programação multithread compatível com um subconjunto de serviços definidos pelo padrão POSIX para threads (IEEE 1003.1c), usualmente identificado como Pthreads. Athreads disponibiliza um núcleo de suporte à execução de programas onde é mantido um grafo descrevendo o relacionamento de troca de dados entre os threads lançados pelo programa. Este grafo é manipulado, em tempo de execução, por um mecanismo de escalonamento aplicando algoritmos

* Bolsista FAPERGS.

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 793-802, 2008.

listas para distribuir o custo computacional das tarefas entre os processadores de uma arquitetura com memória compartilhada.

Neste artigo é apresentada a introdução de um mecanismo de coleta de lixo no núcleo de execução de Athreads. Este mecanismo permite que um thread sofra múltiplas operações de sincronização por *join*. Esta nova funcionalidade estende os recursos de programação definidos por POSIX threads, nos quais um thread pode sofrer no máximo uma (1) sincronização por *join*, permitindo uma maior abrangência de aplicações para Athreads. Este trabalho é ilustrado com uma avaliação de desempenho, sendo comparado o consumo de memória e o tempo de processamento da nova versão de Athreads com os apresentados pela versão sem coletor de lixo.

O restante deste artigo está organizado como segue. A Seção 2 sumariza aspectos relevantes de Athreads neste trabalho e a Seção 3 caracteriza a introdução, na interface de programação, do recurso de coleta de lixo. A Seção 4 apresenta as aplicações desenvolvidas e as análises de desempenho. A Seção 5 conclui este texto.

2 Athreads – Interface de programação para Anahy

Anahy propõe um modelo de programação e execução paralelos. A visão oferecida para o programador é de uma arquitetura paralela com memória compartilhada. O núcleo de execução oferece suporte de escalonamento baseado em algoritmos de lista. A especificação do modelo Anahy, denominado Anahy-Vanilla, encontra-se documentada em [2]. Esta especificação é genérica, podendo ser implementada utilizando diferentes recursos de programação. Uma implementação realizada é Athreads [6, 5], que oferece uma implementação de Anahy com uma interface de programação construída segundo o padrão Pthreads, conforme ilustra a Figura 1. Athreads[1] é disponibilizado como uma biblioteca de serviços para programas em linguagem C/C++.

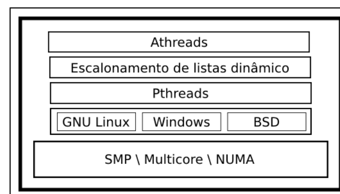
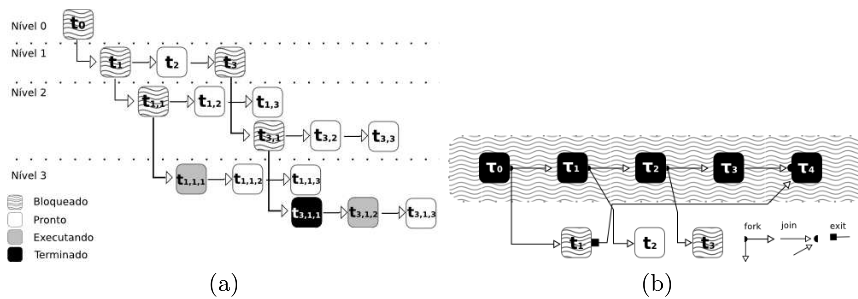


Fig. 1. Organização em camadas de Anahy Vanilla.

O modelo de programação de Anahy permite que o programador descreva a concorrência da aplicação com operações do tipo *split/join*. Estas operações determinam, respectivamente, a criação e sincronização de/entre tarefas ao mesmo

[1] Implementação disponível em www.anahy.org

tempo em que definem a comunicação de dados entre elas. Esta comunicação define a ordem de execução entre as tarefas e permite identificar os trechos de código concorrentes. Estas informações são utilizadas para manter, em tempo de execução um grafo descrevendo o fluxo de dados entre as tarefas. A Figura 2.a ilustra um grafo de criação de threads durante a execução de um programa Athreads – os threads são representados em diferentes estágios de seus ciclos de vida. Nesta figura, um arco vertical indica uma operação de criação de thread, arcos horizontais representam threads criadas por uma mesma origem. A Figura 2.b destaca um nível do grafo da Figura 2.a apresentando as relações de dependência de dados entre tarefas, segundo o modelo de programação empregado em Anahy. A transposição deste modelo de programação à Athreads é realizada pelo uso de serviços de criação e sincronização de threads:
`athread_create(...)/athread_join(...)`. A sintaxe destes serviços é apresentada na seqüência, destaca-se que ambos serviços contêm informações sobre troca de dados entre threads. Conforme definido no modelo, é construído um grafo representando a ordem de execução das tarefas definidas pelo programa. No entanto, no protótipo Athreads, *tarefas* são encapsuladas no contexto de threads, unidades de maior granulosidade que representam uma seqüência ordenada de tarefas.



(a) (b)

Fig. 2. *(a) Grafo de criação de threads em Athreads. (b) Grafo de dependência de dados entre tarefas em Anahy.*

Na Figura 2.b podem ser observados arcos dirigidos no sentido horizontal. Estes arcos representam as dependências entre tarefas contidas em um único thread. Arcos dirigidos entre diferentes níveis representam dependências entre threads. O ponto de partida de um arco representa uma operação de criação de um novo thread (invocação ao serviço `athread_create`), resultando na criação de duas novas tarefas: uma sendo a tarefa sucessora no contexto do thread atual, outra sendo a primeira tarefa no novo thread. O ponto de chegada, representado pela união de dois arcos, indica a invocação de um serviço de `athread_join` como ponto inicial de uma nova tarefa.

O modelo de execução implementado em Athreads preconiza a existência de um suporte de escalonamento baseado em algoritmos de lista. Este escalonador

tem a função de considerar o grafo de dependências para ativar o lançamento de threads sobre os processadores disponíveis. O núcleo de execução é composto por um conjunto de processadores virtuais (PVs), em número igual ou superior ao número de processadores reais da arquitetura utilizada, e consome (executa) os threads descritos no grafo de dependências, segundo alguma política de busca [2]. Desta forma, o grafo de dependência e tratado como a lista de tarefas necessária à operação dos algoritmos de escalonamento de lista, sendo a prioridade entre os threads definida pela seleção da política de busca. A execução do programa é concluída quando o thread identificado com a função `main` do programa principal terminar.

Sintaxe dos serviços Athreads A criação e sincronização de threads em Athreads é realizada com invocação aos serviços `athread_create` e `athread_join` que possuem sintaxe idêntica às operações análogas definidas em Pthreads:

```
int athread_create( athread_t *th, athread_attr_t *attr,
                    *(void *) func, void *in );
int athread_join( athread_t th, (void **) result );
```

O retorno de ambos serviços é um código de erro, indicando sucesso ou falha na execução do serviço, o valor 0 (zero) representa sucesso. O parâmetro `th` representa o identificador de um thread. No caso da operação de criação, `th` é atualizado com um valor único, identificando o novo thread criado. Na operação de sincronização, `th` indica qual o thread que deve ser sincronizado. O parâmetro `func` identifica o nome da função que contém o corpo do novo thread a ser executado. Esta função define a seqüência de tarefas a ser executado pelo novo thread. O parâmetro `in` indica a posição de memória que contém os dados de entrada para a primeira tarefa do novo thread e o parâmetro `result`, no serviço de sincronização, indica o endereço de memória que deve conter o registro do endereço dos dados de retorno oferecidos por um thread sincronizado.

O parâmetro `attr`, utilizado na criação de um novo threads, descreve o conjunto de atributos de execução do novo thread. O padrão Pthreads define uma variedade de atributos que podem ser implementados. Entre os atributos definidos pelo padrão Pthreads, existe um que permite identificar se o thread sofrerá ou não uma operação de sincronização, sendo definido como *joinable* ou *detached*, conforme o caso. Este atributo será considerado para a extensão proposta no prosente trabalho. Em [5] é apresentada outra extensão introduzida em Athreads para uso dos atributos, onde permite-se definir um esquema para realizar o empacotamento e desempacotamento de threads no caso de migração de threads em máquinas com memória distribuída.

A Figura 3 exemplifica a criação e sincronização de threads em Athreads. Nesta figura, a função `inc` contém o trecho de código a ser executado pelo novo thread, o qual retorna o valor recebido em entrada de uma unidade. Pode-se notar que o parâmetro responsável pela configuração de atributos foi definido como `NULL`. Neste caso, a exemplo do que ocorre em Pthreads, as configurações são inicializadas utilizando valores padrões.

```
#include <stdio.h>
#include <athreads.h>

void *inc(void *in) {
 int *v = malloc(sizeof(int));
 v = (int*) in;
 *v += 1;
 return v;
}
```

```
int main(int argc, char *argv[]) {
 athread_t th;
 int *result;
 int value = atoi(argv[1]);
 athread_create(&th, NULL, inc, &value);
 athread_join(th, &result);
 printf("%s + 1 = %d", value, *result);
 return 0;
}
```

Fig. 3. Exemplo de código em Athreads.

3 Inclusão da variação `max_joins` ao padrão POSIX

O padrão POSIX define que threads podem ser executados um de dois modos: *detached* e *joinable*, sendo o modo de execução definido como um *atributo de thread*. Conforme o atributo selecionado, o thread pode ou não sofrer uma, e apenas uma, operação de sincronização por `pthread_join`. Uma operação de sincronização em um *thread joinable* permite que o fluxo seja sincronizado no ponto de chamada desta operação e também que o resultado de retorno do thread seja obtido. Após isto, o thread é finalizado, seu descritor desalocado e o seu identificador torna-se inválido, causando erro caso aconteça tentativa de nova sincronização sobre ele.

Os threads declarados com o atributo *detached*, por sua vez, não podem sofrer nenhuma sincronização por `pthread_join`. Este atributo é associado a um thread quando deseja-se que, ao término de sua execução, todo espaço de memória a ele alocado seja liberado, não havendo, portanto, a expectativa de retorno de resultado.

Em Athreads todos threads criados são do tipo *joinable*, isto é, todos threads necessariamente retornam valor e permitem sincronização. O não retorno de resultado é incompatível com o modelo de execução baseado no fluxo de dados. Para permitir um espectro maior de algoritmos paralelos, Athreads inclui uma variante no atributo *joinable*: a possibilidade de indicar quantas operações de sincronização por *join* o thread pode receber. Este atributo é identificado por *max_joins*. Sua operação permite que o descritor de um thread seja mantido no grafo até que o número de sincronizações especificado ocorra. Quando for atingido o número máximo de sincronizações configurado no atributo `max_joins` de um determinado thread, o nodo que lhe representa no grafo é removido, liberando a área de memória que ele havia alocado para armazenar informações sobre o thread.

4 Aplicações

Para verificar o impacto da utilização do coletor de lixo sobre o grafo de fluxo de dados em Athreads, foram implementadas três aplicações que geram difer-

entes estruturas de dependências. A Figura 4 ilustra os grafos gerados para estas aplicações. Todos experimentos documentados nesta seção foram realizados em uma máquina utilizando sistema operacional Linux, dotada de 8 processadores Intel Itanium 1.5 GHZ e 16 GB de memória; os processadores encontram-se fisicamente localizados em 2 Quad Brick Blocks (QBB), onde o fator NUMA (non-uniform memory access) é 2 entre 2 QBB.

(a) (b) (c)

Fig. 4. Grafos para três aplicações em Athreds (a) Miner, (b) Strassen e (c) Cálculo do número Pi.

Nos resultados de desempenho, cada medida de desempenho apresentada corresponde a uma média de 10 execuções, não tendo sido observado desvio padrão maior do que 5%. Os gráficos identificam no eixo Y a quantidade de memória consumida no tempo de execução dado no eixo X de cada execução. Cada gráfico registra dois pares de curva, cada par representa o consumo de memória RAM e da memória mapeada (swap) para a execução das aplicações com e sem o suporte à coleta de lixo. O ponto final de cada par de curvas identifica o tempo total de execução do caso correspondente. O objetivo da análise é verificar se há redução do consumo de memória com o uso do coletor de lixo e, também, se ocorre penalização no tempo de execução devido a sua introdução no núcleo de execução.

4.1 Miner

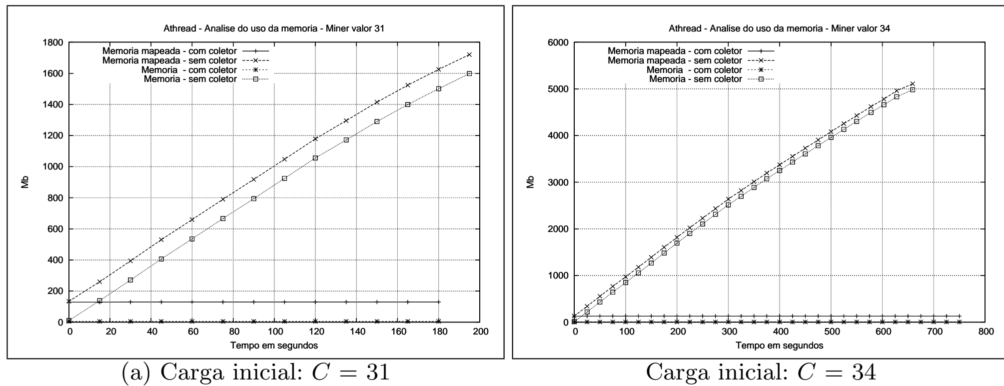
O Miner é uma aplicação recursiva, gerando um grafo de tarefas desbalanceado. Nesta aplicação, cada mineiro representa um thread. Como parâmetro de entrada, cada mineiro recebe uma carga de trabalho C, devendo produzir o resultado da mineração sobre este valor. Cada mineiro pode criar, para auxiliá-lo, outros m mineiros, com cargas de trabalhos $C-1, C-2, \ldots, C-m$. No presente estudo de caso, cada mineiro cria dois ajudantes ($m = 2$) quando o parâmetro C de entrada for maior ou igual a 2. O resultado da computação de um mineiro é a combinação dos resultados dos mineiros auxiliares com o resultado de sua própria mineração. O algoritmo de um mineiro é apresentado na Figura 5.

A Figura 4.1 apresenta os resultados de desempenho obtidos com a execução desta aplicação com diferentes cargas computacionais iniciais, $C = 31$ e $C =$

```
int miner(var carga)
        if not precisa_ajuda
                return carga
        else
                athread_create( thread1, miner( carga - 1 ) )
                athread_create( thread2, miner( carga - 2 ) )
        end
        part1 = athread_join(thread1)
        part2 = athread_join(thread2)
        return (part1 + part2)
```

Fig. 5. Algoritmo de execução de um mineiro.

34, representando dois tamanhos para o problema. Observa-se, a partir destes gráficos, que a necessidade de memória para execução do problema é reduzida quando o mecanismo de coleta de lixo é utilizado. Documenta-se também que o limite para execução deste problema na máquina utilizada para experimentos com a versão sem coletor de lixo foi $C = 34$, limite não encontrado na versão utilizando o coletor de lixo nos experimentos realizados. Outro aspecto verificado nos gráficos, é que é possível observar que sobrecustos de execução são inseridos com o uso do coletor de lixo e que este custo está associado ao tamanho do grafo gerado. Outros experimentos não documentados nesta seção corroboram com esta afirmação.



(a) Carga inicial: $C = 31$ Carga inicial: $C = 34$

Fig. 6. Consumo de memória na execução da aplicação Miner.

4.2 Strassen: Multiplicação de matrizes

O algoritmo de Strassen é um algoritmo de baixo custo computacional para multiplicação de matrizes. Enquanto o método tradicional de multiplicação de

matrizes possui n^3 multiplicações, o algoritmo de Strassen requer $n^{2.87}$ operações, realizando o particionamento das matrizes de entrada em blocos de 2×2 elementos, conforme Eq. 1, sendo as matrizes A, B, e C de tamanho $n \times n$, com n sendo um valor par, gerando $n/2^2$ blocos.

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} * \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \tag{1}$$

Partindo da Eq. 1 são obtidos 7 termos temporários para multiplicação de cada um dos blocos 2×2 construídos:

$T_1 = (A_{1,1} + A_{2,2}) * (B_{1,1} + B_{2,2})$; $T_2 = (A_{2,1} + A_{2,2}) * B_{1,1}$
$T_3 = A_{1,1} * (B_{1,2} - B_{2,2})$; $T_4 = A_{2,2} * (B_{2,1} - B_{1,1})$
$T_5 = (A_{1,1} + A_{1,2}) * B_{2,2}$; $T_6 = (A_{2,1} + A_{1,1}) * (B_{1,1} + B_{1,2})$
$T_7 = (A_{1,2} - A_{2,2}) * (B_{2,1} + B_{2,2})$

A partir dos termos temporários são obtido os valores da multiplicação das sub-matrizes A e B:

$C_{1,1} = T_1 + T_4 - T_5 + T_7$; $C_{1,2} = T_3 + T_5$
$C_{2,1} = T_2 + T_4$; $C_{2,2} = T_1 - T_2 + T_3 + T_6$

A concorrência nesta aplicação encontra-se no cálculo dos termos temporários $T_{1...7}$ para cada um dos $n/2^2$ blocos. A combinação dos resultados das multiplicações produz o resultado final na matriz de saída.

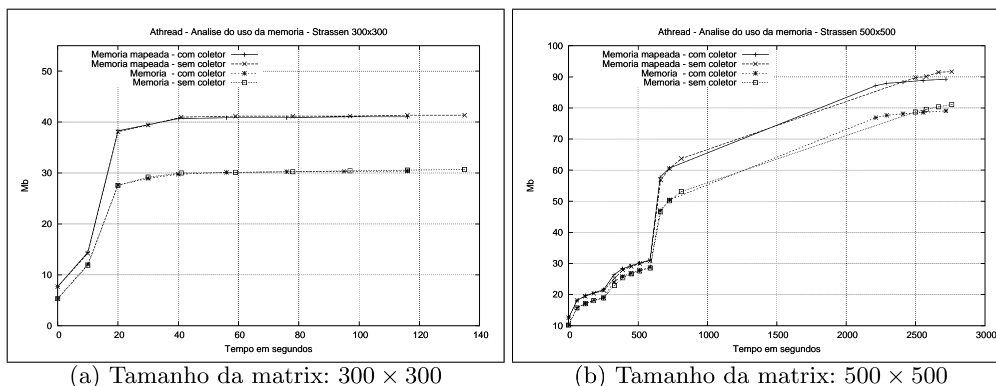
Os resultados de desempenho da aplicação Strassen encontram-se na Figura 7. Como mostram as curvas, os requisitos de memória em ambas versões são equivalentes. Este comportamento é esperado uma vez que os threads para computação dos blocos são criados no início do algoritmo e sincronizados no seu final. O desempenho coletado também possibilitou verificar que a utilização do coletor de lixo permite melhorar o desempenho final de execução nesta aplicação. A razão para tal é que, a medida em que threads são retiradas do grafo, novas pesquisas neste grafo são realizadas em menor tempo.

4.3 Cálculo do número π

O número π é uma das constantes matemáticas mais antigas que se conhece. Diversos trabalhos propõe métodos para realizar o cálculo deste número. Neste trabalho foi empregado um método estatístico, o qual utiliza um quadrado de lado 2 e um círculo de raio 1, sendo que a área do círculo e do quadrado são dadas por: $A_{cir} = \pi * r^2$ $A_{qud} = l^2$

Logo $r = 1$ e $l = 4$. A partir da relação entre as áreas do círculo e do quadrado, obtém-se que:

$$\pi = 4 * \frac{A_{cir}}{A_{qud}} \tag{2}$$



(a) Tamanho da matrix: 300×300 (b) Tamanho da matrix: 500×500

Fig. 7. Consumo de memória na execução da aplicação de multiplicação de matrizes com algoritmo de Strassen

Assim, o algoritmo para obter o número π considera um círculo circunscrito por um quadrado e o opera da seguinte forma:

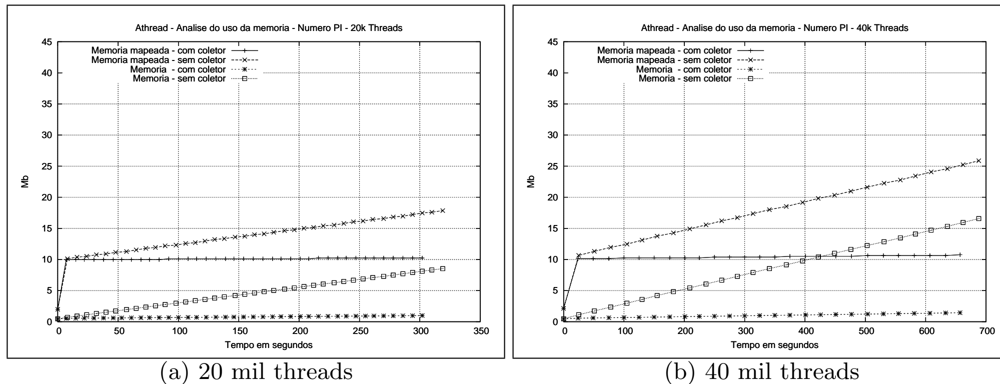
1. São gerados aleatoriamente pontos dentro do quadrado;
2. Para cada ponto gerado, contabiliza-se os que também encontram-se dentro da área do círculo;
3. Toma-se np como o número de pontos no círculo dividido pelo número de pontos no quadrado;
4. Como visto a relação na Equação 2, π $4 * np$.

A implementação da aplicação para calcular o número π de forma paralela, denominada ParPi, foi desenvolvida de forma a explorar ao máximo o número de sincronizações entre os threads. Para isto, foram utilizadas a criação e sincronização de um grande número de threads. Os gráficos na Figura 4.3 mostram a criação e sincronização de 20 e 40 mil threads em intervalos de 200, ou seja, é criado um número de 200 threads e é feita a sincronização destas para acumular resultados parciais.

As curvas de desempenho mostram que o uso da memória da aplicação ParPI é menor quando o coletor de lixo é utilizado. Pode-se notar que no início da execução nas duas situações, com e sem coletor de lixo, o consumo de memória é semelhante. O uso do coletor de lixo, no entanto, permite que as necessidades de memória estabilizem rapidamente.

5 Conclusão

Este trabalho apresentou uma extensão ao suporte executivo de Athreads para suportar uma nova funcionalidade na sua interface aplicativa: a sincronização múltipla de threads. A avaliação de desempenho apontou que este mecanismo permitiu ganhos de desempenho em termos de economia de memória em todos



(a) 20 mil threads (b) 40 mil threads

Fig. 8. Consumo de memória na execução da aplicação para cálculo do número π.

os casos, sendo esta economia melhor visualizada nas aplicações que possuem um maior número de sincronizações. Como efeito colateral positivo, também observou-se redução do custo computacional das operações de escalonamento, resultando assim tanto em economia de memória como em redução no tempo total de processamento.

Como trabalhos futuros, pretende-se desenvolver novas aplicações sobre Athreads, que possuam um grafo de fluxo de dados diferente dos das aplicações mostradas neste trabalho, com o objetivo de obter novos resultados do comportamento do dispositivo de coleta de lixo e desta forma otimizá-lo.

References

1. Kwok, Y.K., Ahmad. I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors ACM Comput. Surv., 31(4):406-471 In VII High Performance Computing for Computational Science, Berlin 2006 Springer-Verlag
2. Cordeiro, O. C. et al.: Exploiting Multithreaded Programming on Cluster Architectures HPCS (2005).
3. Cavalheiro, G. G. H.: Anahy: A programming environment for cluster computing. VII High Performance Computing for Computational Science, Berlin 2006 Springer-Verlag(LNCS4395)
4. Graham, R. L.: Bounds on Multiprocessing Timing Anomalies SIAM Journal of Applied Mathematics, 17(2):416-429, 1969
5. Cavalheiro, G. G. H. ; Gaspary, L. P. ; Cardozo Júnior, M. A.; Cordeiro, O. C.: Anahy: A programming Environment for cluster computing. In: High Performance Computing for Computational Science - VecPar 2006, 2007, Rio de Janeiro. LNCS High Performance Computing for Computational Science - VecPar 2006. Heildelberg : Springer-Velag, 2006. p. 198-211.
6. Cavalheiro, G. G. H. ; Dinis, E. B. ; Saccol, D. P; Moschetta, E. M.: Dynamic List Scheduling of Threads on Clusters CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006