Maximizando os Recursos Computacionais com um Ambiente de Execução Baseado Estritamente em Tarefas

Rodrigo Hübner[1], Juliano Henrique Foleiss[2], André Luiz Tinassi D'Amato[2], Anderson Faustino da Silva[2]

[1]Universidade Tecnológica Federal do Paraná
COINT – BR 369, km 0,5, Bloco A
87301-006, Caixa Postal: 271 – Campo Mourão – Paraná – Brasil

[2]Universidade Estadual de Maringá
Departamento de Informática – Av. Colombo, 5790, Bloco C56
87020-900 – Maringá – Paraná – Brasil

rodrigohubner@utfpr.edu.br, julianofoleiss@gmail.com, altdamato@gmail.com, anderson@din.uem.br

Abstract. *Runtime environments uses strategies for create and manage multiple parallel threads, beyond to provide increase application performance. On the other hand, it doesn't always exploit the resources provided by hardware, and provide mostly programming models where part of management is usually explicit. This paper presents a programming language PLH, a simple language renewing some points in the paradigm of parallel programming and the execution environment PLH-Env which has aimed to explore the most out of multicore architectures. The parallelism in PLH-Env is strictly task-based, so the programming language PLH enhances the use of tasks while much of the management of parallelism is removed from the programmer to the runtime environment.*

Resumo. *Ambientes de execução utilizam estratégias para criar e gerenciar diversos fluxos paralelos, além de prover o aumento do desempenho das aplicações. Por outro lado, nem sempre exploram os recursos disponibilizados pelo hardware, além de fornecerem em sua maioria modelos de programação onde parte da gerência é normalmente explícita. Este artigo apresenta a linguagem de programação PLH, uma linguagem simples que renova alguns pontos no paradigma de programação paralela e o ambiente de execução PLH-Env que possui como objetivo explorar ao máximo os recursos das arquiteturas multinúcleo. O paralelismo em PLH-Env é baseado estritamente em tarefas, desta forma a linguagem de programação PLH potencializa o uso de tarefas enquanto grande parte da gerência do paralelismo é retirada do programador para o ambiente de execução.*

1. Introdução

A evolução do *hardware* ou mais especificamente das arquiteturas computacionais vem se deslocando para UCPs (Unidade Central de Processamento) multinúcleo [Stallings 2009], ou seja, arquiteturas de computador que integram múltiplos processadores. Contudo, para obter um contínuo uso dos recursos disponíveis nas arquiteturas multinúcleo, desenvolvedores precisam explorar o paralelismo do *hardware* subjacente, escrevendo suas

aplicações de forma paralela. Nesse tipo de programação são utilizados mecanismos que permitem especificar unidades ou fluxos paralelos, além do modo de cooperação entre tais fluxos. Atualmente, diferentes mecanismos são utilizados para criar fluxos paralelos e gerenciá-los, com o objetivo de maximizar a utilização dos recursos computacionais [Nichols 1996, Chapman et al. 2007, Planas et al. 2009, El-Ghazawi et al. 2003].

Neste contexto, o recurso mais importante é a UCP que executa efetivamente os diversos fluxos paralelos gerados pelo modelo de execução. Para a efetiva utilização das UCPs disponibilizadas pelo *hardware*, o modelo de execução deve lidar de maneira satisfatória tanto com o gerenciamento dos diversos fluxos paralelos como também dos recursos do *hardware*, além de manter a corretude da aplicação e evitar um forte impacto do ambiente de execução, o que pode acarretar uma perda de desempenho.

Na busca deste ideal diferentes grupos investem em abordagem distintas: seja em linguagem de programação paralela [Chamberlain et al. 2000, Numrich and Reid 1998], bibliotecas paralelas [Blikberg and Sorevik 2001, Tzenakis et al. 2010] ou ainda em ambientes de execução [Planas et al. 2009, Agrawal et al. 2008]. Cada abordagem possui suas vantagens e desvantagens. Bibliotecas ou ambientes de execução para linguagens já existentes possuem a vantagem de possibilitar a portabilidade, mas possuem o potencial de levarem para o modelo de execução estruturas que geralmente ocasionam perda de desempenho, como por exemplo efeitos colaterais funcionais. Novas linguagens de programação possibilitam novas abordagens para solucionar problemas, buscando desta forma viabilizar novos modelos de programação. Contudo, isto interrompe possíveis melhorias em aplicações já estabelecidas.

Embora manter o suporte (inclusive proporcionar melhorias) das aplicações existentes seja necessário, uma questão vital atualmente é a necessidade da mudança de paradigma no desenvolvimento de aplicações, devido ao fato das arquiteturas de *hardware* modernas disponibilizarem vários núcleos de processamento. Portanto, tanto desenvolvedores quanto linguagens/ambientes de execução devem focar esta nova realidade. Se por um lado os desenvolvedores devem escrever códigos paralelos, por outro, as linguagens e seus ambientes de execução devem prover mecanismos para maximizar os recursos disponibilizados.

Idealmente o modelo de programação deveria ser simples de forma a viabilizar uma abordagem não complexa para o desenvolvimento de aplicações paralelas, cujo ambiente de execução seja responsável por gerenciar de forma transparente e eficiente os recursos de *hardware*. Na busca deste ideal, o grupo de pesquisa do Laboratório de Linguagens, Compiladores e Programação Paralela desenvolveu a linguagem de programação PLH [Hübner and da Silva 2011], uma linguagem firmada na premissa de que uma aplicação deve ser uma coleção de unidades funcionais que potencialmente podem ser executadas de forma transparente em paralelo e cujo ambiente de execução (PLH-Env) utiliza estratégias na gerencia de tais unidades paralelas de forma a maximizar os recursos computacionais.

Este artigo tem por objetivo detalhar o funcionamento do ambiente de execução da linguagem PLH, como também apresentar uma avaliação empírica para demonstrar a melhor configuração do ambiente para determinadas aplicações. Assim, este artigo possui as seguintes contribuições: (1) descrever a implementação de um ambiente de execução

capaz de retirar do desenvolvedor parte das questões inerentes a programação paralela, (2) apresentar um ambiente de execução parametrizável e (3) apresentar uma avaliação detalhada de tal ambiente, com a intenção de inferir sua melhor configuração.

O restante deste artigo está organizado como segue. A Seção 2 apresenta alguns trabalhos relacionados. A Seção 3 sintetiza a linguagem de programação PLH. A Seção 4 descreve a arquitetura e as estratégias do ambiente de execução PLH-Env. A Seção 5 apresenta uma análise detalhada de PLH-Env. E, por fim, a Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

OpenMP [Smyk et al. 2006] fornece aos desenvolvedores um meio de obter paralelismo em um ambiente com um espaço de endereçamento compartilhado entre diversos processadores. A ideia básica de OpenMP é *anotar* o código sequencial de forma a identificar códigos que possam ser executados em paralelo. O OpenMP usa um modelo *fork/join* e a sincronização entre *threads* ocorre quase sempre implicitamente. PLH gera fluxos paralelos assim como OpenMP, porém não é necessário anotar blocos de código para criar novas tarefas, pois todas as funções são potenciais tarefas paralelas. Além disso a sincronização em PLH é realizada sempre de forma implícita.

Star superescalar (StarSs) [Planas et al. 2009] é um modelo de programação baseada em tarefas. O programador desenvolve os programas inicialmente em C seqüencial e adiciona anotações *pragmas* para marcar tarefas paralelas, identificando suas entradas e saídas. A execução começa em um fluxo principal em um único processador, quando o *thread* atinge uma tarefa, a tarefa é adicionada a um gráfico de dependência em tempo de execução e posteriormente é executada em paralelo. Este modelo apresenta relação com PLH por ser um modelo baseado em tarefas. Contudo, PLH não é baseado em *fork/join* e não possui anotações em código.

Cilk [Frigo 2007] é uma extensão da linguagem C e seus principais recursos são a especificação de diretivas de paralelismo (spaw) e sincronização (sync). Na execução de uma aplicação em Cilk quando são criados novos fluxos de execução paralelos, estes são adicionados a um grafo acíclico direcionado (DAG) gerado pelo ambiente. Quando um DAG possui mais de um fluxo paralelo, o ambiente de execução implementa uma política baseada no roubo de tarefas [Agrawal et al. 2008] como mecanismo para balanceamento de carga entre os trabalhadores. Assim como em Cilk, PLH-Env utiliza um mecanismo para gerenciar o fluxo de execução das tarefas e as dependências geradas entre elas.

Unified Parallel C (UPC) [El-Ghazawi et al. 2003] também é uma extensão da linguagem de programação C. O modelo de execução UPC possui estratégias para resolver o problema da localidade dos dados, minimizando o *overhead* que envolve a comunicação entre os *threads*, mapeando toda comunicação realizada através de passagem de mensagens para acessos em memória compartilhada. Dados compartilhados são criados de forma explícita com a diretiva *shared* e a associação de um *thread* a um dado específico pode ser realizado com a diretiva *affinity*. PLH-Env e UPC utilizam mecanismos de sincronização global que permitem notificar tarefas para bloquear e continuar a execução de acordo com suas dependências na aplicação.

3. A Linguagem de Programação PLH

A linguagem de programação PLH é uma linguagem de propósito geral, cujo ambiente de execução é responsável por gerenciar os fluxos paralelos de forma implícita ao desenvolvedor.

Com o objetivo de relaxar ainda mais o modelo utilizado para o desenvolvimento de aplicações paralelas, a linguagem de programação PLH não possui em sua sintaxe diretivas paralelas, deixando a cargo do ambiente de execução a responsabilidade de gerenciar os fluxos paralelos. Desta forma, a responsabilidade do desenvolvedor é apenas escrever sua aplicação de forma que esta contenha poucas dependências. Embora esta premissa ocorra em praticamente todos os modelos, o diferencial de PLH está no fato de remover questões gerenciais que tradicionalmente são impostas ao desenvolvedor.

As principais características de PLH são listadas a seguir:

- Uma linguagem estritamente baseada em tarefas. Uma tarefa em PLH é toda porção de código que potencialmente pode ser executada em paralelo.
- Não possui efeitos colaterais funcionais. Atualizações de variáveis globais dentro de subprogramas e/ou passagem de parâmetros por referência, são exemplos de efeitos colaterais que limitam o paralelismo. PLH evita tais construções com o objetivo de maximizar a utilização dos recursos disponibilizados pelo *hardware*, a medida que diminui a necessidade de exclusão mútua [Lin and Snyder 2009].
- Tarefas são executadas de forma assíncrona ou síncrona. O ambiente de execução verifica a necessidade de sincronização entre as tarefas, para determinar a sua forma de execução.
- Tarefas podem armazenar resultados. Isto torna possível executar uma tarefa assíncronamente dentro de um laço de repetição e obter os resultados somente depois que todas as tarefas finalizarem sua execução, postergando os resultados gerado pelas tarefas.

Estas características da linguagem de programação PLH abstraem características específicas do ambiente de execução, que por sua vez, gerencia os fluxos de execução paralelos e a sincronização entre tais fluxos.

Tradicionalmente, quando uma aplicação sequencial é construída, o compilador ou interpretador da linguagem de programação gerencia e executa o código da aplicação também de forma sequencial [Scott 2008]. O modelo de execução de PLH segue uma abordagem diferente, executando porções de código da aplicação fora de ordem [Damm and Pnueli 1997], de acordo com o conjunto de dependências que é descoberto em tempo de compilação, contudo sem infringir a semântica da aplicação.

A linguagem de programação PLH elimina a necessidade de utilizar mecanismos explícitos para o desenvolvimento e gerenciamento de aplicações paralelas, como *threads*, OpenMP ou outros modelos. Por outro lado, nem todas as porções de uma aplicação poderão ser executadas em paralelo, a medida que o desenvolvedor não esteja comprometido em desenvolver código paralelo.

O processo de geração de código além de traduzir PLH para código da arquitetura alvo, instrumenta o código gerado com o objetivo de manter a semântica da aplicação durante a execução fora de ordem. Este processo consiste em duas tarefas: adicionar rotinas do ambiente de execução PLH-Env; e adicionar primitivas de sincronização.

As rotinas necessárias a PLH-Env são aquelas que instanciam a sua arquitetura básica, a qual será descrita em maiores detalhes na próxima seção. Além disto, a cooperação entre as tarefas podem necessitar de sincronização. Neste caso, o código gerado deve ser instrumentado com a inserção de pontos de sincronização a medida que estes sejam necessários.

Um ponto de sincronização é necessário a medida que uma determinada tarefa necessite de resultados que serão produzidos por outra tarefa. Para estes casos, a instrumentação de código insere uma primitiva de *wait* imediatamente antes do ponto cujo resultado seja necessário. A execução de um *wait* altera o estado de uma tarefa para *bloqueada*. E no momento que o resultado que espera pela tarefa esteja pronto, PLH-Env altera o estado da tarefa para *pronta* indicando que está apta para execução.

O Programa 1 apresenta o código PLH da Aplicação SOR.

Programa 1 Aplicação SOR em PLH

```
01 calc(grid,first_row,last_row,n,iters,p) {
02   loop i, [first_row:(last_row-1)] {
03     p == 1 ? {
04       i % 2 == 1 ? j_start = 2;
05       : j_start = 1;
06     } : {
07       i % 2 == 1 ? j_start = 1;
08       : j_start = 2; }
09     loop j, [j_start:(n-1)], 2 {
10       grid[i][j] = (grid[i-1][j] +
11         grid[i][j-1] + grid[i+1][j] +
12         grid[i][j+1]) * 0.25; }
13   }
14   return(grid);
15 }
16 worker(grid, f_row, l_row, n, iters) {
17   import(io);
18   loop i, [0:iters] {
19     grid=calc(grid,f_row,l_row,n,iters,1);
20     grid=calc(grid,f_row,l_row,n,iters,2);
21   }
22 }
23 main() {
24   import(random);
25   import(sys);
26   import(io);
27   n = 10;
28   grid = [];
29   loop l, [0:(n+1)] {
30     temp = [];
31     loop c, [0:(n+1)] {
32       temp = temp+[random.range(1000)];
33     }
34     grid = grid + [temp];
35   }
36   max_iters = 20;
37   part = n / sys.ncpu();
38   loop i, [0:(sys.ncpu()-1)] {
39     f_row = i * part + 1;
41     l_row = (f_row + part) - 1;
42     worker(grid,f_row,l_row,n,max_iters);
43   }
44 }
```

Neste código serão inseridas rotinas de sincronização na linha 19 e 20, pois nesses pontos são esperados o resultado da tarefa `calc` para que a execução possa continuar. Em casos como esse, são acrescentadas dependências na aplicação, que podem resultar em um baixo desempenho. Casos como esse da aplicação SOR podem ser evitados se a aplicação for reescrita para calcular cada parte da matriz separadamente a partir de alguns cálculos a mais durante o desenvolvimento. O ideal é que o programador se comprometa a desenvolver aplicações com o mínimo de dependência para explorar ao máximo o paralelismo disponível no *hardware* utilizado.

4. O Ambiente de Execução PLH-Env

Modelos de programação paralela como *threads*, OpenMP e StarSs requerem que o programador gerencie quase que por completo o ambiente de execução. Embora OpenMP e StarSs sejam modelos menos rígidos que *threads*, estes ainda podem levar a sérios erros de programação, devido aos diversos mecanismos disponibilizados aos desenvolvedores.

PLH-Env abstrai a aplicação desenvolvida como sendo composta por um conjunto de tarefas que cooperam entre si e podem ser executadas em qualquer ordem, desde que a semântica da aplicação não seja modificada e as dependências entre tarefas sejam respeitadas. Desta forma, o ambiente é capaz de manter a semântica da aplicação identificando uma determinada ordem para executar as tarefas que compõem a aplicação.

4.1. Arquitetura do Ambiente

O projeto de PLH-Env foi norteado pela especificação de um ambiente de execução que fosse robusto e facilmente extensível. Desta forma, o ambiente de execução além de gerenciar as questões inerentes à linguagem de programação, também é responsável por aquelas inerentes à programação paralela. Um atrativo de PLH-Env é a sua extensabilidade, facilitando a adição de novas políticas. A arquitetura de PLH-Env é apresentada na Figura 1.

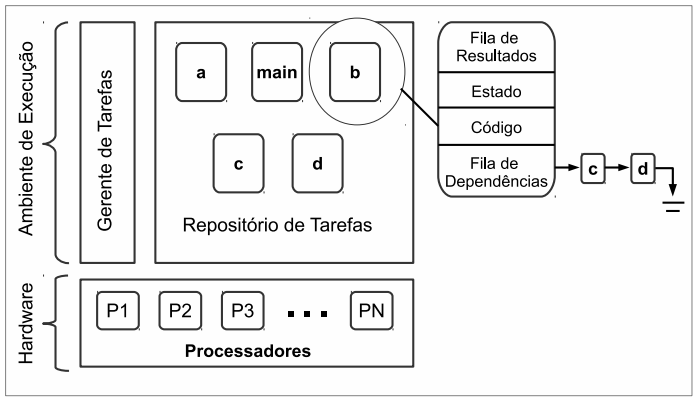

Figura 1. Arquitetura de PLH-Env

PLH-Env é composto basicamente por dois componentes: **Gerente de tarefas**; e **Repositório de tarefas**.

O *gerente de tarefas* é responsável pela alocação de tarefas aos processadores (ou núcleos) físicos, como também da gerência do *respositório de tarefas*. O *gerente de tarefas* garante que a quantidade de tarefas (T) ativas seja no mínimo, igual a quantidade de processadores (P) disponíveis no *hardware* utilizado, com o objetivo de maximizar a utilização dos recursos disponíveis.

Embora a configuração padrão do *gerente de tarefas* garanta que P tarefas estejam ativas simultaneamente, para $T > P$, é possível parametrizar o ambiente de execução para ajustar a quantidade de tarefas ativas. Isto proporciona uma maior flexibilidade ao desenvolvedor, por exemplo para analisar sua aplicação. Contudo, não existem garantias que sempre existirá P tarefas ativas, devido a:

1. Quantidade de fluxos potencialmente paralelos ser menor que P; e/ou
2. Existência de dependências entre os fluxos paralelos.

O *repositório de tarefas* é composto por T instâncias da estrutura que representa uma determinada tarefa, onde cada instância contém: o fluxo de execução da tarefa; o estado corrente de execução; uma fila de resultados; e uma lista contendo a identificação

de todas as tarefas que serão acordadas quando o processamento da tarefa representada estiver terminado.

Uma característica do *repositório de tarefas* é sua capacidade de expandir e contrarir durante a execução da aplicação. Esta flexibilidade garante que PLH-Env fará um bom uso da memória física. Portanto, é de responsabilidade do *gerente de tarefas* instanciar novas tarefas no *repositório de tarefas*, como também liberar as regiões de memória das tarefas finalizadas. Esta funcionalidade é similar a um coletor de lixo [Appel and Ginsburg 1998].

4.2. Modelo de Execução

O modelo de execução de PLH se assemelha ao modelo de execução de C, pelo fato da execução iniciar pela chamada da função cuja assinatura é composta pelo identificador *main*, além dos parâmetros e tipo de retorno. A diferença está no fato de que em PLH, cada chamada de função inicia uma nova tarefa em potencial. Portanto, a execução da aplicação inicia com uma chamada a função *main* e a medida que novas funções são chamadas, novas tarefas são instanciadas.

Para PLH-Env uma tarefa pode estar em um dos seguintes estados: Pronta; Ativa; Bloqueada; ou Terminada.

Uma tarefa está no estado *pronta* se ela está apta para execução e no estado *ativa* se ela está alocada a algum processador físico. Caso esta não possa continuar sua execução, pelo fato de depender de resultados produzidos por outras tarefas ela é retirada do processador e seu estado passa a ser *bloqueada*. Quando o fluxo de instruções de uma tarefa termina ela vai para o estado *terminada*.

A Aplicação 2 será utilizada para exemplificar o modelo de execução utilizado em PLH. No início da execução da aplicação, PHL-Env inicia seus componentes e chama a função *main*. Esta chamada irá instanciar a primeira tarefa do *repositório de tarefas*. Por sua vez, o *gerente de tarefas*, ao identificar que existe tarefa no *repositório* e que existem processadores ociosos, retira *main* do repositório e a aloca a um processador. A medida que a execução de *main* chama novas funções, novas tarefas são instanciadas no *repositório de tarefas* pelo *gerente de tarefas*.

Programa 2 Aplicação PLH exemplo

```
01 d(value) {...}
02 c(value) {...}
03 b(value) { return value**2; }
04 a() {
05     x = io.read();
06     Y = b(x);
07     return c(y); }
08 main() {
09     x = a();
10     y = b(x);
11     d(y); }
```

As tarefas instanciadas se comunicam com o *gerente de tarefas* em duas situações, a saber:

1. Imediatamente antes de necessitar de resultados produzidos por outras tarefas; e
2. Imediatamente antes de terminar sua execução.

Ambos os casos, notificam o *gerente de tarefas* da necessidade de alocar uma nova tarefa a um processador. No primeiro caso, o *gerente de tarefas* retira a tarefa do processador, armazena o seu estado corrente de execução (composto pela pilha de execução e os

registradores) no repositório, altera o estado da tarefa para *bloqueada* e aloca uma nova tarefa ao processador ocioso, nesta respectiva ordem. No segundo, o *gerente de tarefas* retira a tarefa do processador, aloca uma tarefa cujo estado seja *pronta* ao processador ocioso e libera o espaço de memória até então consumido pela tarefa em estado *terminada*, também nesta respectiva ordem. Além destas atividades, o *gerente de tarefas* é também responsável por notificar a uma tarefa que os resultados, que ela está esperando, estão prontos. Neste caso, o *gerente de tarefas* armazena os resultados na *fila de resultados* da respectiva tarefa e altera o seu estado para *pronta*. Este processo continua, até que não existam mais tarefas para executar.

5. Análise de Desempenho e Discussão

Para avaliar o potencial de PLH-Env foram realizados experimentos em um computador Intel(R) Xeon E5504, 2,00GHz de frequência de UCP, 8 núcleos divididos em 2 processadores, 32KB de *cache* L1 separadas em cache de dados e *cache* de instruções, 256KB de *cache* L2 unificada, 4MB de *cache* L3 unificada e 24GB de memória RAM executando o sistema operacional Ubuntu com kernel 2.6.32-38-server.

Nestes experimentos foram utilizados três grupos de aplicações, a saber:

1. Aplicações totalmente paralelas:
   Mandelbrot gera um conjunto contendo 5000 pontos em um plano complexo.
   SparseMatMul multiplica duas matrizes esparsas de tamanho 2000x2000.
   Series calcula os 50000 primeiros coeficientes da série de Fourier.
2. Aplicações que tradicionalmente são implementadas com *locks*:
   RWPathology soma e incrementa elementos de um vetor de tamanho 150000.
   SharedCounter incrementa um contador 600000000 vezes.
   StridePathology soma elementos intercalados de um vetor de tamanho 2000000.
3. Aplicações que tradicionalmente são implementadas com *barreiras*:
   LU aplica fatoração LU em uma matriz 1100x1100.
   N-body simula a colisão de 1000 corpos.
   SOR aplica o método *Red/Black* sucessivo em uma matriz 300x300.

Embora o objetivo de PLH-Env seja maximizar os recursos computacionais disponíveis, o ambiente de execução foi parametrizado para utilizar uma quantidade específica de processadores. Desta forma, será possível avaliar mais precisamente a escalabilidade de PLH-Env, como também o uso dos recursos computacionais disponibilizados. Nos experimentos cada aplicação do *benchmark* foi executada seis vezes, variando a quantidade de processos em 1, 2, 4, 8, 16, 32 e 64. Assim, os dados apresentados representam o tempo médio entre seis execuções.

5.1. *Speedup*

Como mostrado na Figura 2 cada grupo de aplicações possui um comportamento diferente. Nesta figura os resultados apresentados correspondem a quantidade de processadores físicos existente.

Tradicionalmente era esperado que as aplicações que não necessitam de comunicação entre as tarefas (MANDELBROT, SPARSEMATMULT e SERIES) obtivessem uma curva mais próxima do ideal. Contudo, os resultados demonstram uma situação um

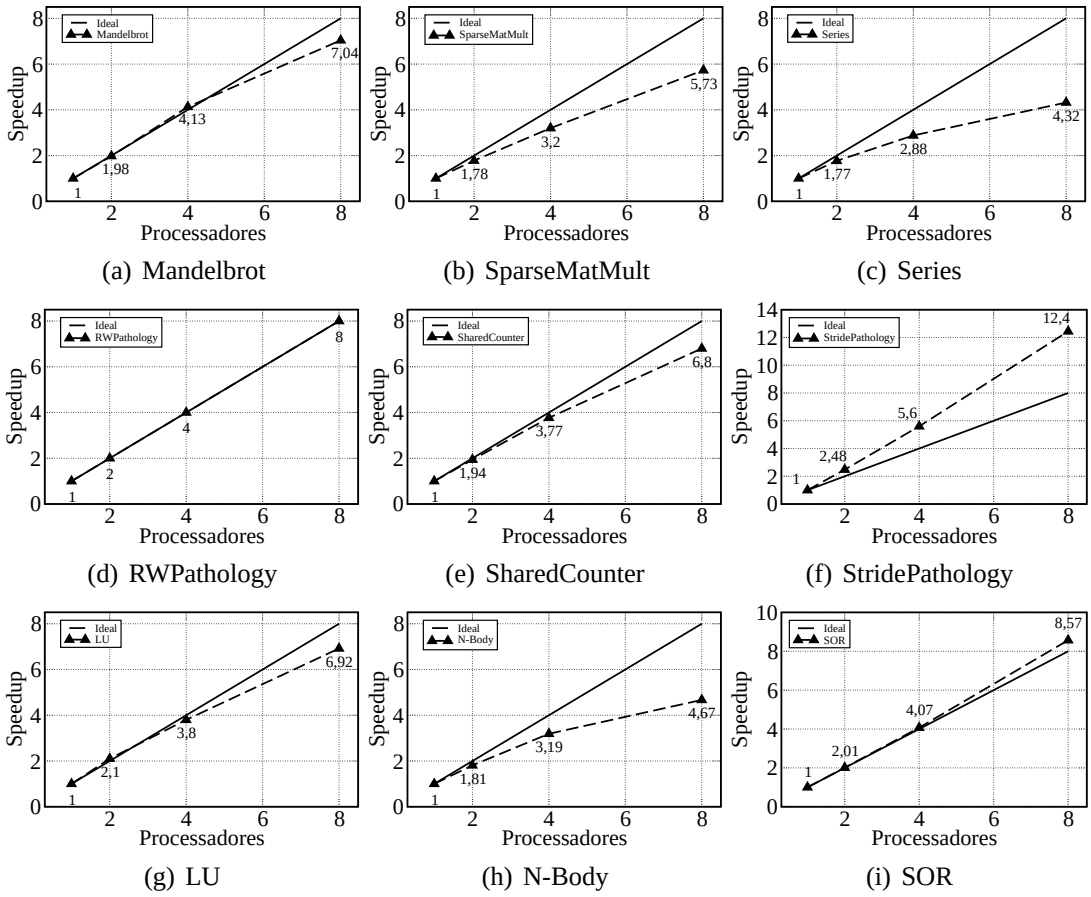

Figura 2. *Speedup* do *benchmark*

pouco diferente. As aplicações que obtiveram uma melhor desempenho são aquelas cujas tarefas cooperam entre si para obter o resultado final. No contexto de PLH, isto significa que existe uma forte dependência entre as tarefas.

Os resultados apresentados na Figura 2 demonstram que as aplicações cujas tarefas não necessitam de comunicação possuem um *speedup* médio de 1,8/3,4/5,7 para dois, quatro e oito processadores respectivamente. Por outro lado, as aplicações cujas tarefas interagem possuem um *speedup* médio de 2,0/4,0/7,9.

Um outro dado importante não mostrado nestes gráficos, é que para a aplicação N-Body, ao aumentar a quantidade de tarefas para mais de oito, o *speedup* se mantém crescendo até uma configuração com 16 unidades paralelas com um *speedup* de 1,8/3,1/4,6/9,3 para 2, 4, 8, e 16 tarefas respectivamente. Isto se dá pelo motivo de que N-BODY possui poucas dependências, o que possibilita explorar melhor os recursos físicos a medida que a quantidade de tarefas aumenta. Vale ressaltar novamente que a arquitetura utilizada possui apenas oito núcleos físicos. Para outras três aplicações (MANDELBROT, SPARSEMATMULT, SOR) o ganho para mais de oito tarefas foi menor, com uma média de 1,9/3,8/7,1/7,5/8,1/9,2 para 2/4/8/16/32/64 tarefas respectivamente. Já o restante das aplicações não obtiveram um bom desempenho com mais de oito tarefas por serem desenvolvidas de acordo com a quantidade de núcleos físicos existentes.

De acordo com uma análise das características das aplicações é possível indicar

alguns fatores que influenciaram o desempenho de PLH, a saber: a criação de uma quantidade excessiva de tarefas pequenas, uma distribuição desproporcional entre o tamanho das tarefas e uma política de escalonamento de tarefas ineficiente.

Estes três fatores explicam a perda de desempenho das aplicações MALDELBROT, SPARSEMATMUL e SERIES. MALDELBROT é a aplicação que obteve o melhor e mais estável desempenho deste grupo. Enquanto as aplicações SERIES e SPARSEMATMUL possuem um desvio padrão máximo de 1,26 e 5,76, respectivamente nesta ordem.

SERIES é uma aplicação que cria uma quantidade excessiva de tarefas, o que está impactando negativamente na gerência do modelo de execução. Além disso, a dependência gerada entre as tarefas da aplicação SERIES é muito alta, resultando na má utilização dos recursos computacionais. Na aplicação MANDELBROT que não possui dependência entre as tarefas o desempenho foi melhor do que as outras aplicações deste grupo, pois toda chamada de tarefa é assíncrona e existe um balanceamento de carga quase igualado entre as tarefas que estão em execução. Já a aplicação SPARSEMATMUL tem o problema de potencialmente gerar tarefas com tamanho diferentes, o que ocasiona um desbalanceamento de carga, contudo, mesmo com este problema possui uma melhor escalabilidade que SERIES.

As tarefas que obtiveram um melhor desempenho são aquelas cuja quantidade de tarefas é proporcional a quantidade de processadores e/ou possuem uma distribuição proporcional em sua carga. SOR, LU e N-BODY, RWPATHOLOGY, SHAREDCOUNTER e STRIDEPATHOLOGY possuem estas características, com uma ligeira variação na maneira do ambiente gerenciar as dependências entre as tarefas.

As aplicações que tradicionalmente são implementadas com o uso de barreiras, escalam menos do que aquelas implementadas tradicionalmente com *locks* (RWPATHOLOGY, SHAREDCOUNTER e STRIDEPATHOLOGY). O mapeamento das aplicações em PLH pode ser realizado da seguinte forma: Toda vez em que é encontrado uma sincronização por barreira, a tarefa é dividida em duas partes, gerando uma dependência da primeira parte para iniciar a segunda parte. Já em aplicações que utilizam *locks* como mecanismo de sincronização, cada tarefa que possuía um *lock* para alterar um dado com exclusividade, em PLH vai atualizar este mesmo dado independente em cada tarefa e ao receber os resultados de cada tarefa que atualizou este mesmo dado, um cálculo deve ser realizado para manter a mesma integridade que existia anteriormente ao utilizar *locks*. Isto demonstra que o *overhead* ocasionado pelo gerenciamento das tarefas, quanto a trocar o estado de uma tarefa, é baixo. Contudo, o aumento das trocas de estados tende a aumentar o *overhead* do ambiente, consequentemente ocasionando uma perda de desempenho, como é o caso das aplicações cujo algoritmo possuem diversas iterações e cada iteração precisa de resultados da anterior.

Um ponto importante a ser destacado é o fato das aplicações RWPATHOLOGY, SHAREDCOUNTER e STRIDEPATHOLOGY serem beneficiadas com a semântica de PLH, pelo fato de possibilitarem minimizar a quantidade de trocas de estado de suas tarefas. Esta situação não ocorre em uma implementação tradicional que utiliza *locks*, pois a cada acesso a um dado compartilhado existe uma exclusão mútua. PLH utiliza instâncias diferentes de dados e ao final do processamento coleta os resultados produzidos pelas diversas tarefas para gerar o resultado final, ocasionando uma redução na quantidade de interações

entre as tarefas, em outras palavras, uma redução na quantidade de dependências entre as tarefas.

5.2. Uso de UCP

Um dos objetivos com PLH é o desenvolvimento de uma linguagem de programação que maximize o uso dos recursos disponíveis no *hardware*, mas precisamente o uso da UCP. Como mostrado na Figura 3 apenas duas aplicações do *benchmark* não alcançaram este objetivo, SERIES e N-BODY. Os gráficos de utilização de UCP descrevem as configurações de 1, 2, 4 e 8 processadores.

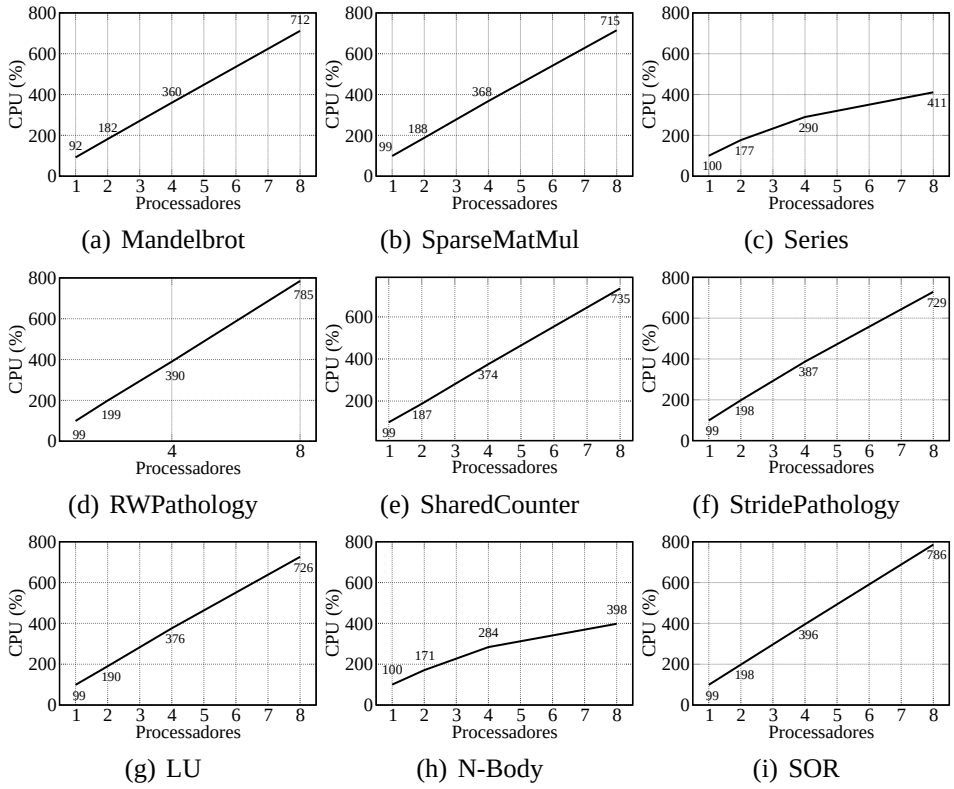

Figura 3. Ocupação da CPU do *benchmark*

Os problemas descritos na seção anterior contribuiram para que SERIES não alcançasse o objetivo proposto para o desenvolvimento de PLH. Uma análise detalhada dos dados coletados durante a execução desta aplicação demonstram uma estabilidade apenas para uma execução com um único processador. Para esta configuração, o percentual de uso de CPU é de 99,66%, com um desvio padrão de apenas 0,51. Por outro lado, esta situação é bem diferente para uma configuração com uma quantidade maior de processadores.

A medida que a quantidade de processadores aumenta, SERIES continua estável porém reduz a utilização da UCP. Para uma configuração com dois processadores o uso da UCP possui uma média de 177%, com um desvio padrão de apenas 0,5. Em uma configuração com quatro processadores, a média sobe apenas para 290%, com um desvio padrão maior de 0,6. Por fim, para uma configuração com oito processadores, a média é de 411%, com um desvio padrão de 1,3.

Esta situação é atípica, por exemplo, para uma configuração com oito processadores, SERIES chega a utilizar apenas 407% da utilização da UCP, sendo que o ideal seria uma valor próximo a 800%. Uma análise futura mais detalhada de SERIES responderia essas questões, identificando se o problema está no código da aplicação ou no ambiente de execução.

N-BODY é outra aplicação que necessita de uma análise mais detalhada para identificar o problema de seu baixo uso de UCP. Para uma configuração com um ou dois processadores, N-BODY possui um bom desempenho. Contudo, após esta quantidade de processadores existe uma perda de desempenho, o qual é mais acentuado para oito processadores. A utilização de UCP por N-BODY é bem próxima aos resultados obtidos por SERIES, mesmo que em N-BODY era de se esperar uma menor utilização UCP pois quando comparado com SERIES, existe uma maior dependência entre as tarefas na aplicação. Para N-BODY o desvio padrão do percentual de uso de UCP é de 0/2,06/7,68/13,30 para 1, 2, 4, e 8 processos/processadores respectivamente.

Estes resultados indicam um possível problema no ambiente de execução, durante o aumento da quantidade de processadores até oito. O estudo detalhada das aplicações SERIES e N-BODY possivelmente ajudará a descobrir as possíveis causas deste problema e consequentemente, a melhora do ambiente de execução.

5.3. Consumo de Memória

A Figura 4 apresenta o consumo de memória de cada aplicação do *benchmark* para 1, 2, 4, 8 e 16 tarefas.

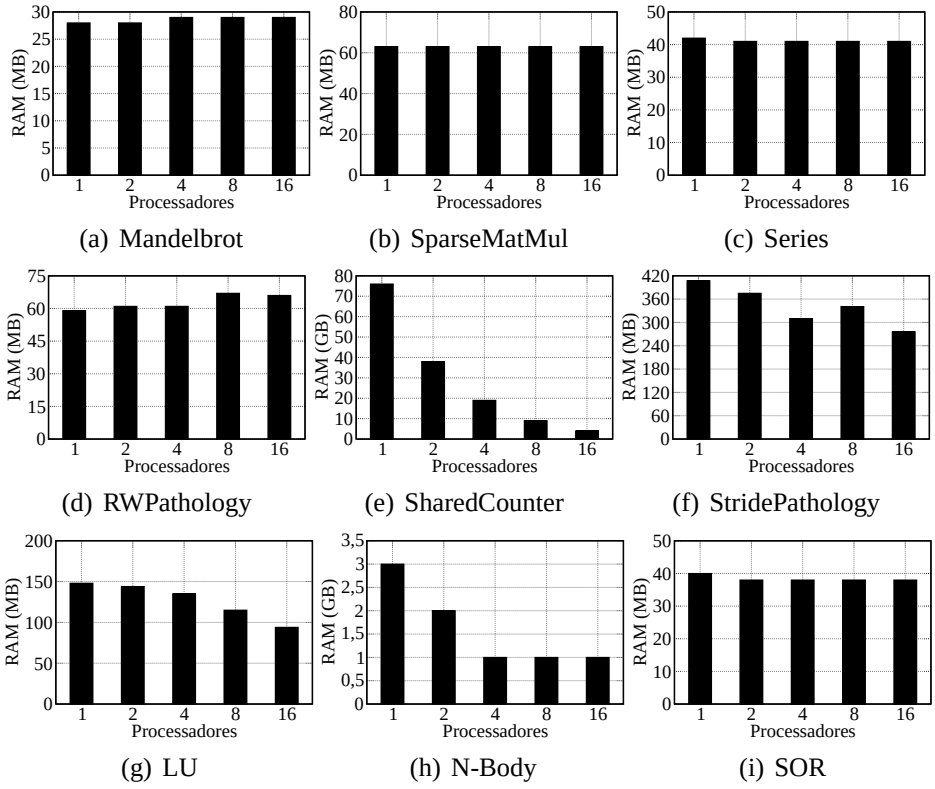

Figura 4. Consumo de memória do *benchmark*

O consumo de memória é dependente de cada aplicação e está relacionado basicamente ao tamanho dos dados da aplicação e a quantidade de tarefas criadas. Como a quantidade de tarefas existentes em uma aplicação não depende da configuração de *hardware*, a tendência é que a quantidade de consumo de memória seja constante.

Apenas para as aplicações SHAREDCOUNTER e N-BODY o consumo de memória não se manteve constante. É interessante observar que para estas aplicação, o aumento da quantidade de processadores ocasiona uma redução do consumo. Em SHAREDCOUNTER a cada aumento na quantidade de processadores ocorre uma redução de 50% no consumo de memória. Em N-BODY esta redução é mais discreta sendo de 30%, 19% e 12%, progressivamente quando a quantidade de processadores dobra. Além disto, não era esperado que SHAREDCOUNTER tivesse um consumo de memória tão elevado. Esta também é uma situação atípica e necessita de um estudo mais detalhada para uma compreensão maior desta situação.

6. Conclusões e Trabalhos Futuros

Programação paralela tem sido um tópico de pesquisa durante os últimos anos. Diversas pesquisas têm focado no desenvolvimento de sistemas que utilizem diferentes mecanismos para prover escalabilidade com desempenho. Contudo, muitas pesquisas embora tenham demonstrado um ganho de desempenho ainda ocasionam uma alta carga ao desenvolvedor, pelo fato deste ter que lidar com diversas questões gerenciais.

PLH é uma linguagem de paradigma imperativo de propósito geral, que busca executar de maneira implícita as unidades funcionais da aplicação, sem contudo adicionar construções paralelas a linguagem de programação. Sua implementação foi norteada para prover o uso mais efetivo dos recursos computacionais pelo ambiente de execução PLH-Env.

O objetivo principal de PLH-Env é ser um ambiente de execução que maximiza a utilização dos recursos disponíveis no *hardware*. Para isto, PLH retira do programador a maior parte da gerência de questões paralelas, como por exemplo o uso de primitivas de sincronização. Questões gerenciais ficam a cargo do ambiente de execução. Desta forma, a linguagem com o ambiente de execução é diferente de outros modelos de programação paralela apresentados neste artigo.

O desenvolvimento do ambiente de execução PLH-Env demonstrou que é possível maximizar a utilização dos recursos computacionais disponíveis, ocasionando escalabilidade a medida que os processadores aumentam, porém os desenvolvedores devem se comprometer em desenvolver aplicações paralelas.

Por outro lado, a avaliação de PLH demonstrou que existem problemas que devem ser resolvidos, como também lacunas que devem ser preenchidas para que o ganho de desempenho, com o uso de PLH, seja maior.

O próximo desafio no desenvolvimento de PLH-Env será disponibilizar um mecanismo de comunicação síncrona e acrescentar um escalonador de tarefas mais eficiente.

Referências

Agrawal, K., Leiserson, C. E., He, Y., and Hsu, W. J. (2008). Adaptive work-stealing with parallelism feedback. *ACM Trans. Comput. Syst.*, 26(3):1–32.

Appel, A. W. and Ginsburg, M. (1998). *Modern Compiler Implementation in C*. Cambridge University Press, New York, Cambridge.

Blikberg, R. and Sorevik, T. (2001). Nested parallelism: Allocation of threads to tasks and openMP implementation. *Scientific Programming*, 9(2-3):185–194.

Chamberlain, B. L., Choi, S.-E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. (2000). Zpl: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26:197–211.

Chapman, B., Jost, G., and Vanderpas, R. (2007). *Using OpenMP*. MIT Press, New York, NY, USA.

Damm, W. and Pnueli, A. (1997). Verifying out-of-order executions. Technical report, Jerusalem, Israel, Israel.

El-Ghazawi, T., Carlson, W., Sterling, T., and Yelick, K. (2003). *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, USA.

Frigo, M. (2007). Multithreaded programming in cilk. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 13–14, New York, NY, USA. ACM.

Hübner, R. and da Silva, A. F. (2011). PLH: A General Purpose Parallel Language for Humans. In *Proceedings of the IADIS International Conference on Applied Computing*, pages 1–8, Rio de Janeiro, Brasil. IADIS.

Lin, C. and Snyder, L. (2009). *Principle of Parallel Programming*. Addison Wesley, California, USA, 1 edition.

Nichols, B. (1996). *Pthreads Programming*. Oreilly and Associates, New York.

Numrich, R. W. and Reid, J. (1998). Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31.

Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. (2009). Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23:284–299.

Scott, M. L. (2008). *Programming Language Pragmatics*. Elsevier, USA, 3 edition.

Smyk, A., Tudruj, M., and Masko, L. (2006). Open mp extension for multithreaded computing with dynamic smp processor clusters with communication on the fly. In *Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, pages 83–88, Washington, DC, USA. IEEE Computer Society.

Stallings, W. (2009). *Computer Organization and Architecture: Design and Performance*. Prentice Hall, USA, 8th edition.

Tzenakis, G., Kapelonis, K., Alvanos, M., Koukos, K., Nikolopoulos, D. S., and Bilas, A. (2010). Tagged procedure calls (tpc): Efficient runtime support for task-based parallelism on the cell processor. In Patt, Y. N., Foglia, P., Duesterwald, E., Faraboschi, P., and Martorell, X., editors, *HiPEAC*, volume 5952, pages 307–321, USA. Springer.