Controle de Granularidade com *threads* em Programas MPI Dinâmicos

João Vicente F. Lima, Nicolas Maillard
Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil
{joao.lima, nicolas}@inf.ufrgs.br

Resumo

O controle de granularidade é um fator importante no desempenho de programas paralelos. Problemas estáticos adaptam sua granularidade pela decomposição e atribuição de dados a cada tarefa, mas em irregulares não é possível prever a carga de trabalho antes da execução. Os irregulares que utilizam decomposição recursiva, como ordenação, necessitam de dinamismo com suporte a criação de tarefas sob demanda. Alguns ambientes de programação, como Cilk e KAAPI, oferecem dinamismo e trabalham com granularidade através do conceito abstrato de tarefa porém possuem limitações que dificultam seu uso em PAD. O MPI, padrão de fato em PAD, oferece dinamismo de processos e uso de threads mas atribui à implementação especificar o comportamento na criação de uma tarefa. Este trabalho propõe investigar as vantagens no controle de granularidade com threads em programas MPI dinâmicos, através da substituição da criação de processos por tarefas onde um mecanismo (libSpawn) decide entre lançar processo(s) ou thread(s). Os resultados obtidos com o programa de ordenação Cilksort, que segue o modelo Divisão-e-Conquista, demonstram ganhos de até 85% *em criação de tarefas e comunicação.*

1 Introdução

A granularidade é um fator importante para o desempenho de programas paralelos em termos de comunicação e criação de tarefas. Um programa estático controla sua granularidade ao decompor um problema inicial em pedaços menores de modo que cada tarefa receba uma carga de trabalho previamente conhecida. Entretanto, programas irregulares desconhecem sua carga de trabalho antes da execução e decompõem um problema inicial em menores recursivamente até que seja possível chegar a uma solução eficiente. Esta decomposição envolve a criação sob demanda de tarefas e a sincronização correspondente ao envio de parâmetros e retorno de resultados. Este modelo de programação, conhecido como programação dinâmica ou dinamismo, depende de um ambiente de programação que ofereça suporte a criação de tarefas sob demanda.

Alguns ambientes de programação oferecem dinamismo e controle de granularidade através do conceito abstrato de tarefa, que pode ser um processo pesado ou leve (*thread*). Cilk [2] e KAAPI [12] controlam sua granularidade em tempo de compilação ou execução, respectivamente, além de estenderem uma linguagem de programação com primitivas de criação e sincronização de tarefas. Porém, apresentam limitações que dificultam seu uso na programação de alto desempenho (PAD). O MPI [7], atualmente padrão de fato em PAD, incorpora dinamismo de processos e suporte a *threads* concorrentes a partir da segunda versão [13]. Contudo, a implementação MPI é responsável por descrever a forma que as tarefas são criadas e definidas, e as implementações atualmente as definem como processos.

Este trabalho propõe investigar as vantagens no controle de granularidade com *threads* em programas MPI dinâmicos, através da substituição da criação de processos por tarefas onde um mecanismo decide entre lançar processo(s) ou *thread(s)*. A biblioteca libSpawn implementa este mecanismo de controle ao criar *threads* da biblioteca POSIX Threads até um certo limite, que define quando novos processos serão criados. Além da criação de tarefas, a libSpawn manipula a troca de mensagens entre *threads* de forma que a comunicação seja transparente. Os experimentos apresentados são baseados no algoritmo de ordenação Cilksort originalmente implementado em Cilk com o modelo de programação Divisão-e-Conquista (D&C). Os resultados demonstram ganhos significativos de até 85% em relação a versão sem controle de granularidade, devido ao ganho em criação de tarefas e comunicação.

Neste artigo a seção 2 descreve o contexto científico em problemas irregulares e dinâmicos, seguido do estado da arte em ambientes de programação na seção 3. Na seção 4 a proposta da biblioteca libSpawn é descrita, e seus resultados são apresentados na seção 5. Por fim, a seção 6 mostra a conclusão e trabalhos futuros.

2 Dinamismo e Aplicações Paralelas Irregulares

A irregularidade de uma aplicação paralela significa a imprevisibilidade de carga de trabalho até a sua execução [14]. Essa imprecisão está relacionada à plataforma de execução ou ao tipo de aplicação tratada. A irregularidade de plataforma corresponde aos ambientes heterogêneos dos quais o compartilhamento de recursos com outros programas e o *hardware* exercem influência no desempenho das aplicações. Grid, NOW (*Networks of Workstations*) e *clusters* são exemplos de plataformas com recursos heterogêneos e compartilhados [9, 1, 3]. Aplicações paralelas que utilizam plataformas como a descrita contornam a irregularidade através do balanceamento de carga ou da alocação de recursos sob demanda, que consiste em dividir o problema inicial entre tarefas criadas dinamicamente. Estes dois métodos citados dependem de um ambiente de programação que proporcione primitivas para este fim.

Problemas irregulares, por sua vez, apresentam algoritmos com um comportamento irregular orientado à tarefa ou à entrada de dados [19]. Algumas das principais propriedades relacionadas à irregularidade de um algoritmo incluem:

- entrada irregular, como em uma matriz esparsa;
- estruturas internas que crescem ou mudam em tempo de execução;
- localidade das dependências de dados;

Um modelo de programação que aborda entradas irregulares é o divisão e conquista (D&C) onde um problema maior é subdividido em menores recursivamente até se chegar em uma solução trivial, que termina com a combinação dos resultados [14]. A implementação de programas com este modelo requer dinamismo de tarefas na medida que estas são criadas sob demanda para resolver um conjunto menor do problema em paralelo. Implementações deste modelo são possibilitadas por ambientes de programação, apresentados na seção 3, que suportam dinamismo de tarefas.

3 Trabalhos Relacionados: Ambientes de Programação Dinâmicos

Os ambientes de programação que oferecem dinamismo atualmente podem ser específicos para arquiteturas SMP (*Symmetric multiprocessor*) ou *clusters*. Cilk [2] é um ambiente de programação para arquiteturas SMP que estende a linguagem de programação C através de 3 primitivas, ou palavras-chave, das quais é possível definir paralelismo e sincronização. As tarefas criadas dinamicamente consistem em *threads* com execução assíncrona e escalonamento de acordo com a dependência entre tarefas através do roubo de tarefas (*Work-Stealing*). O ambiente controla a granularidade de acordo com a dependência de dados em tempo de compilação e o número de processadores disponíveis em tempo de execução. Os trabalhos relacionados ao Cilk [2, 10] mostram a eficiência em expressar paralelismo e dinamismo com simplicidade. Entretanto, a falta de uma implementação otimizada para ambientes distribuídos é a sua principal limitação.

KAAPI [12] é um ambiente para *clusters* SMP e *multicore* que estende a linguagem C++ por *macros* e possibilita expressar paralelismo e sincronização entre tarefas. O ambiente descreve uma tarefa como uma *thread* em modo usuário onde uma ou mais *threads* são mapeadas em processadores virtuais (VT's), que são *threads* em modo núcleo encarregadas de executar as tarefas conforme as decisões de escalonamento. Este escalonamento segue o mesmo mecanismo do Cilk, com roubo de tarefas, e atua no controle de granularidade entre os nós disponíveis à medida que os VT's ficam ociosos. Outros trabalhos em KAAPI [12, 5] relatam a eficiência e estabilidade satisfatória do ambiente, no entanto, nenhum estudo avalia a complexidade de código necessária na programação de aplicações paralelas.

O MPI (*Message-Passing Interface*) [7] é uma interface de programação padrão amplamente utilizada em aplicações com troca de mensagens. A versão MPI-2 incorpora a criação dinâmica de processos e o suporte a chamadas concorrentes entre *threads*, de forma que programas MPI se beneficiem de dinamismo e paralelismo multinível no caso de *clusters multicore*. A criação de tarefas de acordo com o padrão é descrita como assíncrona e coletiva, se mais de um processo estiver envolvido. Por outro lado, a especificação omite detalhes de implementação sobre o gerenciamento e criação das tarefas. Trabalhos recentes abordam o escalonamento [4] e roubo de tarefas hierárquico [17] em aplicações dinâmicas com o propósito de distribuir a carga de trabalho entre os recursos disponíveis.

A descrição do suporte a *threads* no MPI proporciona explorar um paralelismo intra-nó em memória compartilhada, principalmente em SMP's *multicore*. Programas que usam *threads* devem ser inicializados pela chamada `MPI_Init_thread` onde o nível de concorrência suportado pela implementação da interface é consultado. O nível `MPI_THREAD_MULTIPLE` possibilita que múltiplas *threads* façam chamadas MPI e obtenham resultados semelhantes em qualquer ordem, desde que o programa esteja correto e sem ambigüidades. Outros trabalhos discutem especificamente as necessidades e o custo de exclusão mútua em implementações MPI com suporte a *threads* [16, 20].

As implementações com maior suporte ao MPI-2 atualmente são o MPICH2 [15] e Open MPI [11]. Entretanto, apenas o MPICH2 apresenta estabilidade na criação dinâmica de processos e uso de *threads*. No Open MPI, o suporte é previsto para a próxima versão (1.3.0).

4 libSpawn: Controle de Granularidade com *threads* no MPI

O controle de granularidade está entre um dos parâmetros importantes para o bom desempenho de programas paralelos, principalmente nos irregulares que dependem da localidade e custo na criação de tarefas [8]. No contexto do MPI-2, o controle de granularidade envolve a escolha entre criar processos pesados ou *threads* com a finalidade de reduzir custos em criação de tarefas, além de reduzir custos em comunicação. Dessa forma, este trabalho investiga as vantagens no controle de granularidade com *threads* em programas MPI dinâmicos através da biblioteca libSpawn.

A libSpawn consiste em 10 chamadas MPI sobrescritas que estão relacionadas ao controle de granularidade e implementam o mecanismo de criação de tarefas. Este mecanismo proporciona ao processo de um dado executável criar um número de *threads* menor ou igual ao limite estabelecido em tempo de compilação. A criação de processos se inicia no momento que o número de *threads* ultrapassar o limite descrito. Além da criação de tarefas, a libSpawn trata da troca de mensagens entre *threads* devido ao fato do MPI designar identificadores somente para processos dentro de um comunicador. A tabela 1 lista as funções que a libSpawn sobrescreve atualmente.

Tabela 1. Funções MPI implementadas na libSpawn

MPI_Init	MPI_Finalize
MPI_Send	MPI_Recv
MPI_Comm_rank	MPI_Comm_size
MPI_Comm_free	MPI_Comm_spawn
MPI_Comm_get_parent	MPI_Comm_disconnect

4.1 Controle de Granularidade

A chamada `MPI_Comm_spawn` concentra o mecanismo de controle implementado pela biblioteca em que dois testes decidem se a tarefa será uma *thread* ou um processo. O primeiro compara o comando da nova tarefa com o nome do executável que deu origem ao processo atual, e se diferentes um novo processo é criado. O próximo teste valida o número de *threads* concorrentes no processo em relação ao limite estabelecido. Se ultrapassado, um novo processo deve ser criado. A partir destes testes a estrutura `task_t`, descrita na seção 4.2, é configurada para cada nova *thread* e passada como parâmetro na função `pthread_create` da biblioteca POSIX Threads.

A configuração do limite de *threads* é definida em tempo de compilação pela *macro* `MAX_THREADS` e utilizada na chamada `MPI_Comm_spawn` através de um contador global que contém o número de *threads* que estão em execução no processo. A figura 1 mostra de forma simplificada os principais passos do algoritmo descrito. A criação de um processo padrão MPI está em uma região crítica devido ao fato da chamada `MPI_Comm_spawn` original não permitir sua execução concorrente.

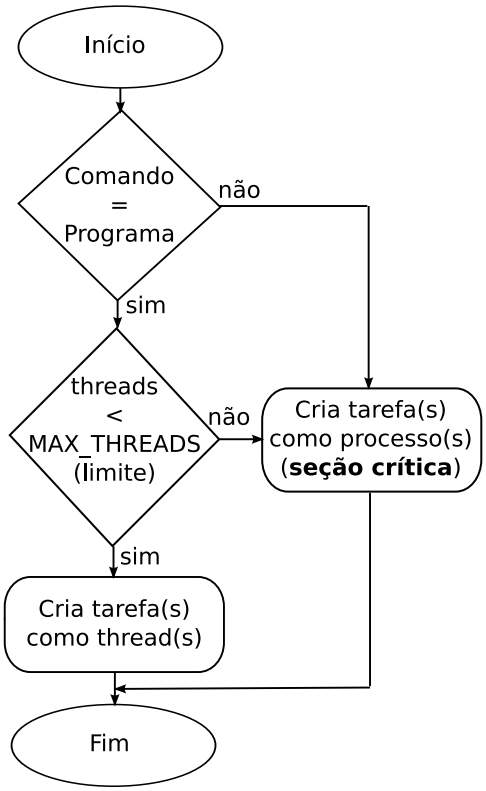


Figura 1. Fluxograma da função `MPI_Comm_spawn` na libSpawn

4.2 Estrutura de uma *thread*

A estrutura `task_t` do código 1 descreve uma *thread* do processo e seus campos, juntamente com a variável `task` responsável por armazenar as informações sobre a *thread* em execução. O atributo de compilação `__thread` declara a variável como global somente para cada *thread*, o que elimina o uso de exclusão mútua [6]. O campo `id` da estrutura `task_t` identifica a *thread* dentro do mesmo comunicador e caracteriza o seu *rank* nele. A seguir, `argc` e `argv` informam os parâmetros da função `main` da nova *thread*, e `thread` representa a *thread* atual pelo valor recebido da biblioteca POSIX Threads. O espaço de

memória para a estrutura é alocado e configurado na função MPI_Comm_spawn que em seguida passa como parâmetro à função pthread_create.

```
typedef struct
{
 int        id;
 int        *argc;
 char       **argv;
 pthread_t  thread;
 MPI_Comm   parent;
 int        *ref_comm;
} task_t;

static task_t __thread *task = NULL;
```

Código 1. Estrutura task_t que descreve uma *thread*

O comunicador parent da estrutura task_t representa as *threads* que pertencem ao mesmo grupo e cada uma delas recebe o mesmo valor, devido ao tipo MPI_Comm ser um identificador e não um endereço de memória. A função MPI_Comm_spawn da biblioteca adquiri um novo identificador ao duplicar o comunicador recebido como parâmetro, o que permite a libSpawn diferenciar contextos, como definido no padrão MPI, mesmo que a *thread* não o utilize diretamente para a comunicação através de chamadas MPI. A razão de se usar identificadores do MPI está relacionada ao uso indireto do comunicador nas funções de troca de mensagens, como explicado na seção 4.3.

O campo ref_comm indica o número de *threads* em execução que fazem referência ao comunicador parent. Dessa forma, quando ref_comm chegar a zero o comunicador é destruído. O propósito da contagem de referência se deve à possível liberação do comunicador antes que as *threads* finalizem suas referências a ele. Mesmo que o padrão MPI descreva que a estrutura de um comunicador é destruída somente após o término da última referência, uma das *threads* do mesmo comunicador pode tentar liberar ele no momento que não há referências a parent.

4.3 Troca de Mensagens entre *threads*

A libSpawn faz um controle simplificado da comunicação entre *threads* do mesmo processo nas chamadas MPI_Send e MPI_Recv. Duas listas são mantidas para o controle das requisições de envio e recebimento de mensagens, respectivamente, com exclusão mútua a fim de evitar acessos indevidos.

O MPI define a identificação de uma mensagem pelo nome **message envelope** [7] que consiste em 4 campos:

- *source* - origem da mensagem
- *destination* - destino da mensagem
- *tag* - identificador de mensagens
- *communicator* - comunicador utilizado

Estes campos de identificação são utilizados na comparação de requisições correspondentes e realização da transferência de dados.

A libSpawn executa as operações de comparação e transferência de dados através de uma *thread* que faz buscas nas listas de requisições a cada chamada MPI_Send ou MPI_Recv. O código 2 descreve os passos na busca, sinalização e espera da troca de mensagens. A *thread* procura por requisições correspondentes de acordo com o **message envelope** de cada uma e realiza a transferência de dados entre elas. As *threads* responsáveis pelo par de requisições são sinalizadas em uma variável condicional contida na requisição de forma que a tarefa continue sua execução. Por fim, *wait_and_unlock* libera o *mutex* e bloqueia a *thread* até que outra chamada comunicante a sinalize.

```
mutex_lock(mutex);
while (1) {
  send = head_list_send;
  receive = head_list_receive;
  while (send and receive) {
    if (match(send, receive)) {
      complete (send, receive);
      signal_threads (send, receive);
    }
    send = next (send);
    receive = next (receive);
  }
  wait_and_unlock (conditional, mutex);
}
```

Código 2. Algoritmo para completar comunicações

A operação de transferência não faz com que os endereços de memória recebidos como parâmetro sejam compartilhados pelas diferentes *threads*, e ao invés disso os dados de envio são copiados diretamente ao endereço de destino. O padrão MPI prevê este comportamento com operações comunicantes e utilizar compartilhamento requer a definição de atributos aos endereços em questão.

5 Resultados

5.1 Ordenação Paralela por Cilksort

O Cilksort consiste na implementação de um algoritmo de ordenação no ambiente Cilk baseada em Mergesort [10]. Este algoritmo se caracteriza pela abordagem D&C no particionamento dos números recursivamente até um limite pré-determinado para a ordenação seqüencial. O código 3 demonstra a estrutura básica em notação Cilk onde in é a entrada de n números.

O vetor de entrada in, assim como o auxiliar tmp, é divido em 4 partes de tamanho $n/4$ e estas são ordenadas em paralelo pelas tarefas criadas na chamada `spawn`. Logo em seguida, o `sync` faz com que a tarefa atual espere o término das tarefas lançadas anteriormente e inicie a união das partes ordenadas com o `cilkmerge`.

O `cilkmerge` inicia com `median` que encontra as medianas de A e B onde o elemento médio de A (ma) é $n/2$ e de B (mb) é o maior elemento tal que seja menor ou igual a ma. As medianas formam um par (ma, mb) de forma que $ma + mb = (n + m)/2$ e todos os elementos em $A[1..ma]$ são menores que $B[mb..m]$, e todos de $B[1..mb]$ são menores que $A[ma..n]$. Por fim, novas tarefas são lançadas para a união entre as diferentes partes.

A implementação em MPI trabalha com a ordenação da mesma forma mas impõe limitações na sincronização. O modelo recursivo D&C permite limitar a comunicação entre tarefas criadas (tarefas filhas) e seu criador (tarefa pai) [18]. Dessa forma, a implementação em questão exige duas sincronizações: envio das entradas aos filhos e retorno dos resultados ao pai. O envio das entradas se faz a partir de duas comunicações por filho em que a primeira informa o número de elementos para ordenação, e a segunda os elementos de entrada. No retorno dos resultados, os filhos enviam as partes ordenadas ao pai através de uma comunicação por filho.

Além disso, o paralelismo em Cilk é MIMD (*Multiple Instruction Multiple Data*) que em MPI deve ser expresso em MPMD (*Multiple Programs Multiple Data*). A implementação MPMD do algoritmo consiste em 3 programas: **cilksort-start**, **cilksort-sort** e **cilksort-merge**. **cilksort-start** lê o vetor de entrada de um arquivo e inicia a ordenação ao lançar o primeiro **cilksort-sort**. **cilksort-sort** e **cilksort-merge** implementam as funções `cilksort` e `cilkmerge`, respectivamente, além das operações de sincronização descritas.

A ordenação seqüencial em cada caso é iniciada quando o número de elementos é menor ou igual a 2048. As ordenações seqüenciais correspondentes são *quicksort* para o `cilksort` e *mergesort* para `cilkmerge`.

```
cilksort(in[1..n]) =
 decl tmp[1..n];
 A = in;  B = A + n/4;
 C = B + n/4; D = C + n/4;
 tmpA = tmp;  tmpB = tmpA + n/4;
 tmpC = tmpB + n/4; tmpD = tmpC + n/4;
 spawn cilksort(A, tmpA);
 spawn cilksort(B, tmpB);
 spawn cilksort(C, tmpC);
 spawn cilksort(D, tmpD);
 sync;
 spawn cilkmerge(A, B, tmpA);
 spawn cilkmerge(C, D, tmpC);
 sync;
 spawn cilkmerge(tmpA, tmpC, A);
 sync;

cilkmerge(A[1..n], B[1..m], C[1..(n+m)]) =
 median (A[1..n], B[1..m], ma, mb);
 spawn cilkmerge(A[1..ma], B[1..mb],
   C[1..(n+m)/2]);
 spawn cilkmerge(A[ma..n], B[mb..m],
   C[(n+m)/2..(n+m)]);
 sync;
```

Código 3. Algoritmo do Cilksort em notação Cilk

5.2 Análise de Ganhos em Função da Granularidade

A avaliação de desempenho utilizou 3 entradas de dados com 3 milhões de elementos em ordem aleatória, que geram 13.313 tarefas. Os testes foram realizados em 10 nós do Grid'5000 com 2 processadores Dual Core, em um total de 40 *cores*, e rede Gigabit Ethernet. As médias foram obtidas a partir de 20 execuções independentes em cada configuração.

Os gráficos da figura 2 demonstram os tempos da solução proposta em relação aos tempos sem controle de granularidade. Por motivos de legibilidade, estes gráficos se encontram na página 7.

O histograma da figura 2(a) demonstra os tempos de acordo com a entrada de dados e a granularidade utilizada. O tempo de execução com 1 *thread* supera o tempo sem a libSpawn devido ao custo da biblioteca em execuções sem criação de *threads*. Por outro lado, os ganhos obtidos com este controle de granularidade a partir de 2 *threads* por processo são significativos. O caso com 2 *threads* reduz o tempo de execução em aproximadamente 37% e o melhor tempo corresponde a granularidade com 100 *threads* e 85% de ganho.

O bom desempenho demonstrado se deve ao aumento da granularidade que proporcionou reduzir o custo em criação de tarefas com *threads*, além de reduzir o custo em comunicação. A característica D&C do algoritmo contribui para os bons resultados em troca de mensagens com menor custo devido as primeiras tarefas serem *threads* e utilizarem mensagens maiores para enviar as entradas e receber os resultados.

Os tempos com 1, 4 e 20 *threads* demonstram o aumento do custo em criar tarefas como processos na implementação atual, mesmo nos dois últimos casos onde o ganho em comunicação e criação de processos leves foi reduzido. A granularidade com 2 *threads* implica na criação contínua de processos em fase de ordenação (*cilksort*) devido ao número pretendido de tarefas, no caso 4, ser maior que o limite estabelecido. Todavia, a fase de união (*cilkmerge*) cria *threads* e processos alternadamente conforme a execução, o que proporciona melhorias de desempenho. Por outro lado, a granularidade de 4 *threads* permite a fase de ordenação criar 4 tarefas como *threads* na primeira chamada recursiva. Porém, na próxima chamada o processo lança 4 tarefas como processos. Dessa forma, o custo do mecanismo implementado na libSpawn se apresenta maior para alguns casos em conseqüência do maior número de processos criados.

Os gráficos 2(b), 2(c) e 2(d) confirmam o custo inicial e os ganhos significativos da solução proposta. Todavia, estes gráficos demonstram outra característica que é a sobrecarga com relação ao aumento na granularidade. Os tempos obtidos a partir de 80 *threads* são semelhantes ou maiores que anteriores devido a contenção de memória reduzir os ganhos citados anteriormente.

As medições também consideram o pior caso onde os números de entrada estão em ordem decrescente, e o melhor caso quando os números estão ordenados. O número de tarefas criadas por ambos é 7.154 e os resultados apresentam tendências semelhantes as discutidas para números em ordem aleatória. Os tempos obtidos são semelhantes entre os dois casos e o melhor resultado é de $14,28$ segundos com desvio padrão de $1,18$.

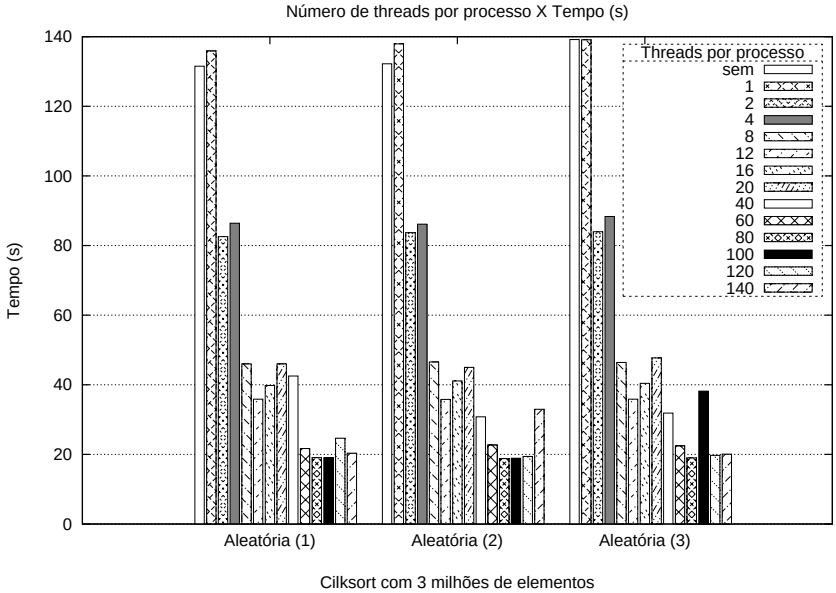
6 Conclusão

A imprecisão da carga de trabalho até a execução caracteriza aplicações paralelas irregulares que se originam da irregularidade de plataforma ou aplicação. As implementações de problemas irregulares contornam esta imprecisão através da criação de tarefas sob demanda, ou dinamicamente, o que dificulta um controle de granularidade adequado. Um ambiente de programação com suporte a dinamismo e controle de granularidade é fundamental para o desempenho de aplicações com estas características.

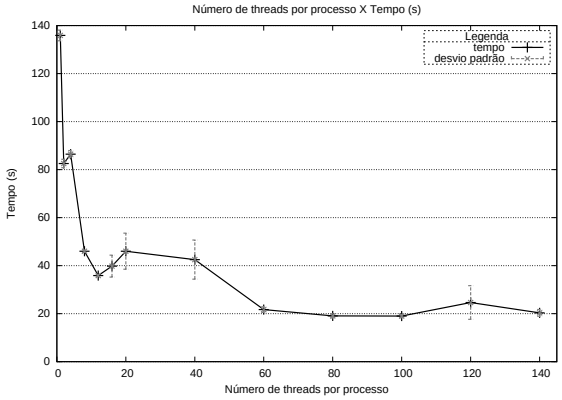
Ambientes como Cilk e KAAPI suportam dinamismo, mas suas limitações dificultam o uso em PAD. O MPI, padrão de fato em PAD, oferece na versão MPI-2 a criação dinâmica de processos e utilização concorrente de *threads*. Todavia, as implementações são responsáveis pelo comportamento na criação de tarefas. Este trabalho investigou as vantagens do controle de granularidade com *threads* em programas MPI dinâmicos através de um mecanismo que decide entre a criação de processo ou *thread* chamado libSpawn. A implementação controla a granularidade de tarefas ao permitir a criação de várias *threads* por processo.

A avaliação de desempenho utilizou o algoritmo de ordenação Cilksort que segue a abordagem de programação D&C. Os resultados demonstram ganhos significativos de até 85% em relação aos tempos sem controle de granularidade. Os principais fatores responsáveis pelos bons resultados estão relacionados a criação e comunicação de tarefas localmente, mesmo com uma perda de desempenho devido ao custo da biblioteca em situações com maior número de processos.

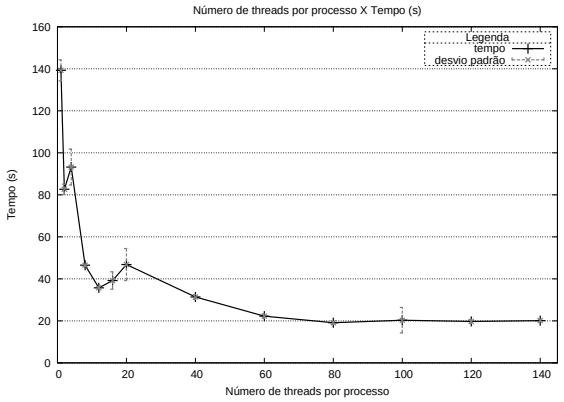
Como trabalhos futuros, o mecanismo de controle deverá ser alterado a fim de incluir um limite de granularidade configurável em tempo de execução através de parâmetros como CPU's disponíveis ou carga de trabalho do sistema. Outra melhoria prevista é incluir maior suporte ao MPI, com comunicações não bloqueantes e coletivas dentro de um grupo de *threads*. Além disso, o mecanismo proposto será validado frente a outros programas paralelos com diferentes irregularidades e características. Entre estas características, deverão ser considerados custos de comunicação, processamento e localidade.



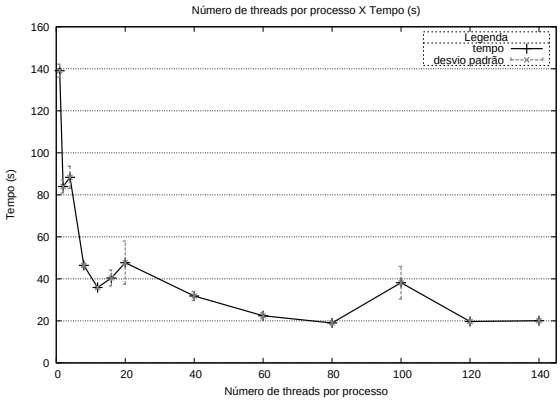
(a) Histograma com as 3 entradas aleatórias



(b) Entrada aleatória (1)



(c) Entrada aleatória (2)



(d) Entrada aleatória (3)

Figura 2. Resultados e devio padrão do Cilksort com entradas de 3 milhões de elementos aleatórios

Referências

[1] M. A. Bender and M. O. Rabin. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems Special Issue on SPAA00*, 35:289–304, 2002.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, July 1995. ACM Press.

[3] N. Capit, G. D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[4] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the Dynamic Creation of Processes in MPI-2. *Lecture Notes in Computer Science - 13h European PVM/MPI Users Group Meeting*, 4192/2006:247–255, Sept. 2006.

[5] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive Loops with Kaapi on Multicore and Grid: Applications in Symmetric Cryptography. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 33–42, New York, NY, USA, 2007. ACM.

[6] U. Drepper. ELF Handling For Thread-Local Storage. http://people.redhat.com/drepper/tls.pdf, December 2005.

[7] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report CDA-9115428, University of Tennessee, Knoxville, Tennessee, July 1997.

[8] I. Foster. *Designing and Building Paralllel Programs*. Addison-Wesley, 1995.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, jun 1998.

[11] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Lecture Notes in Computer Science - 13h European PVM/MPI Users Group Meeting*, 3241/2004:97–104, Nov. 2004.

[12] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007.

[13] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. L. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par, Vol. I*, volume 1123, Lyon, France, aug 1996. Springer.

[14] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2th edition, 2003.

[15] W. Gropp. MPICH2: A New Start for MPI Implementations. *Lecture Notes in Computer Science - 9th European PVM/MPI Users Group Meeting*, 2474/2002:37–42, Sept. 2002.

[16] W. Gropp and R. Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, Sept. 2007.

[17] G. P. Pezzi, M. C. Cera, E. Mathias, N. Maillard, and P. O. A. Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. *SBAC-PAD 2007: 19th International Symposium on Computer Architecture and High Performance Computing, 2007.*, pages 247–254, 2007.

[18] G. P. Pezzi, M. C. Cera, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Escalonamento Dinâmico de programas MPI-2 utilizando Divisão e Conquista. *WSCAD'06 - Workshop em Sistemas Computacionais de Alto Desempenho*, 7:71–78, 2006.

[19] G. Rünger. Parallel Programming Models for Irregular Algorithms. *Lecture Notes in Computational Science and Engineering - Parallel Algorithms and Cluster Computing*, 52:3–23, 2006.

[20] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. *Lecture Notes in Computer Science - 14h European PVM/MPI Users Group Meeting*, 4757:46–55, Sept. 2007.