HieraAnalyses - Uma ferramenta para análise hierárquica de programas paralelos

Thatyana de F. Piola Seraphim
Instituto de Física de São Carlos
Universidade de São Paulo - Brazil
thatyana.piola@gmail.com

Enzo Seraphim
IESTI
Universidade Federal de Itajubá
seraphim@unifei.edu.br

Gonzalo Travieso
Instituto de Física de São Carlos
Universidade de São Paulo - Brazil
gonzalo@ifsc.usp.br

Resumo

Informações detalhadas para a análise de desempenho de programas paralelos podem ser coletadas através de arquivos de trace. Geralmente, esses arquivos de trace contêm um registro de eventos individuais que ocorrem durante a execução do programa. Considerando que os eventos são geralmente de baixo nível, como operações de comunicação em um sistema paralelo, e que é cada vez mais comum para o programador de aplicações usar abstrações de alto nível (por exemplo, uma rotina paralela de autovalores), existe uma diferença semântica entre a informação coletada e os conceitos usados para o desenvolvimento da aplicação, impedindo o uso eficiente dessa informação. Neste trabalho, é proposta uma nova abordagem para arquivos de trace, onde os arquivos contêm informações sobre os diferentes níveis hierárquicos de uma aplicação. Os arquivos seguem o formato XML, onde as rotinas são tags XML, com rotinas auxiliares chamadas durante sua execução de tags filhos. A abordagem é demonstrada pela sua implementação para o nível da bilbioteca MPI e para o nível do OOPS, sendo este último um framework orientado a objetos com abstrações de alto nível para o desenvolvimento de programas paralelos que usam a biblioteca MPI para sua implementação. Para complementar este trabalho, ferramentas de análise usando o formato de arquivo são apresentadas.

1 Introdução

A computação paralela tem surgido como uma ferramenta indispensável em muitas áreas, onde o desempenho obtido com um único processador não é o suficiente, tornando o processamento paralelo indispensável. Máquinas paralelas que variam de CPUs com vários processadores a grid computacional, de *clusters* de computadores pessoais a sistemas paralelos dedicados estão disponíveis para suprir a necessidade dessas aplicações. Contudo, devido a complexidade do desenvolvimento de *software* paralelo, o uso de sistemas paralelos é restrito a aplicações que podem ser facilmente decompostas em tarefas ou aplicações independentes onde sua importância justifica o alto investimento em recursos necessários.

Uma das razões para esta situação é a importância do desempenho para os programas paralelos, pois o desempenho é geralmente o fator que justifica mudar para uma implementação paralela do programa, apesar da complexidade adicional do código e do custo de *hardware*. Para atingir um bom desempenho, é importante ser capaz de avaliar os fatores que determinam o desempenho de uma aplicação em um dado sistema paralelo. Muitas ferramentas que têm sido propostas ajudam nesta avaliação, como por exemplo, AIMS [9], Pablo [4, 1], Vampir [15], entre outras. A técnica de arquivos de *trace* é usada para registrar eventos que ocorrem durante a execução de programa. Com a análise destes eventos e sua duração, conclusões podem ser obtidas sobre gargalos de desempenho ou seções do código onde otimizações podem ser úteis.

Programas paralelos podem ser desenvolvidos usando bibliotecas de comunicação, como MPI (*Message Passing Interface*) [13] e PVM [2], mas soluções de mais alto nível como ScaLAPACK [6] estão sendo usadas, pois fornecem abstrações que estão mais perto do domínio da aplicação e podem ser otimizadas para alcançar um desempenho melhor para uma ampla escala de plataformas. O uso de

abstrações de mais alto nível cria um problema semântico quando se trabalha com arquivos de trace, pois esses arquivos são geralmente baseados em eventos de baixo nível, como operações de comunicação. Considere o exemplo do usuário da biblioteca POOLALi [19], uma biblioteca orientada a objetos para rotinas de autovalores e autovetores do ScaLAPACK. A biblioteca POOLALi é baseada no ScaLAPACK, que é por sua vez é baseada no PBLAS que é baseada no BLACS. O BLACS é implementado usando MPI (ou PVM) (como mostra a Figura 1). Para o usuário da biblioteca POOLALi, os eventos de *trace* relacionados com as operações de comunicação do MPI são úteis. É importante que os eventos no nível de chamadas de métodos POOLALi sejam registrados. Mas, em alguns casos, análises adicionais requerem acessos a eventos de um nível mais baixo de abstração. Um arquivo de *trace* completo baseado na abordagem para análise de desempenho incluiria informações dos eventos realizados em todos os níveis de abstrações (POOLALi, ScaLAPACK, PBLAS, BLACS, MPI/PVM).

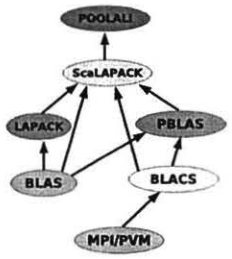

Figura 1. Níveis de Abstração

Este artigo apresenta a ferramenta *Hierarchical Analyses* que permite a avaliação de desempenho em diferentes níveis de abstração. O restante deste artigo está organizado como segue: a seção 2 apresenta ferramentas que são usadas na análise de desempenho. A ferramenta proposta neste trabalho é apresentada na seção 3. Para exemplificar o uso da ferramenta apresentada, é utilizado o programa de dinâmica molecular usando uma versão implementada em MPI e outra no *framework* OOPS, como pode ser visto na seção 4. As conclusões são apresentas na seção 5.

2 Trabalhos Correlatos

A avaliação de desempenho tem por objetivo identificar gargalos. Ferramentas são usadas para ajudar entender o comportamento de programas paralelos, balanceamento de carga, quantidade de comunicações e outras questões relacionadas com o desempenho da aplicação. Algumas ferramentas relacionadas ao desempenho são apresentadas a seguir.

A ferramenta `gprof` ajuda a identificar rotinas ou linhas de código onde o programa gasta mais tempo [10], coletando informações sobre o tempo gasto em cada rotina e o número de chamadas. Esta informação é útil para a identificação de otimizações ou paralelização. Essa ferramenta não tem suporte explícito para paralelismo.

MPE (*Multi-Processing Environment*) está relacionada ao MPICH implementação do MPI e pode ser usada em outras implementações. Suporta facilidades incluindo *profiling* e ferramentas de visualização. A biblioteca de *profiling* trabalha com a interface de *profiling* do MPI [8].

Pablo [7, 18], Paraver [12] e Vampir [7, 15] são ambientes para coleta, análise e visualização de desempenho dos dados de programas paralelos. Os eventos registrados correspondem às operações de comunicação e I/O de MPI. Paraver trabalha com OpenMP e Java. Vampir tem um mecanismo que limita a quantidade de eventos registrados, escolhendo os eventos mais apropriados para a análise desejada.

Paradyn [5, 16] e AIMS (*Automated Instrumentation and Monitoring System*) [21] permitem monitoramento de programas paralelos em tempo real. A instrumentação feita por Paradyn é ajustada dinamicamente durante a execução do programa. O usuário especifica os dados de desempenho a serem coletados (como tempo de CPU, operações de comunicação ou sincronização) e as partes do programa a serem instrumentadas. Não existe a necessidade de recompilar o programa para mudar a instrumentação. Na ferramenta AIMS, o comportamento do programa pode ser visualizado através de animações.

Na ferramenta IPS [11, 14], a instrumentação do código é inserida automaticamente durante a compilação, com a coleta de eventos como chamadas e retornos de procedimentos, operações de sincronização, I/O, criação de processos, entre outros.

3 Ferramenta Hierarchical Analyses

Das ferramentas de avaliação de desempenho apresentadas na seção anterior, nenhuma delas é estruturada e leva em conta os vários níveis de abstração usados no desenvolvimento da aplicação. Esta seção descreve a ferramenta *HieraAnalyses*, desenvolvida para demonstrar a facilidade da abordagem proposta neste artigo. A ferramenta é composta de dois módulos: um módulo de coleta descrito na Seção 3.1 e um módulo de transformação apresentado na Seção 3.2.

3.1 Coleta dos Dados

Os dados a serem usados para a análise de desempenho são coletados e armazenados pelo módulo *hieraCollector*. O formato XML (*eXtensible Markup Language*) [3] é usado pois reflete a organização lógica de chamadas a procedimentos. Essa organização lógica é representada através de uma estrutura de dados em árvore.

Cada rotina de biblioteca que terá sua execução monitorada deverá ser adaptada para a inclusão do código instrumentado. Isto é feito manualmente pelo desenvolvedor da biblioteca ou algum outro usuário com acesso ao código fonte. A coleta foi implementada nas operações da biblioteca do MPI e nas classes do *framework* OOPS (*Object Oriented Parallel System*) [20] por fornecer abstrações de alto nível. Como o OOPS usa MPI para sua implementação, é possível através do sistema de coleta hierárquica analisar o desempenho no nível de chamadas de métodos do OOPS ou operações de comunicação MPI.

A gramática do arquivo XML gerado é definida por um arquivo DTD (*Document Type Definition*). O DTD usado para MPI e OOPS é apresentado abaixo.

```
<!ELEMENT processor(hieraMPI | hieraOOPS)*>
<!ATTLIST processor
 rank      ID    #REQUIRED
 init      CDATA #REQUIRED
 finalize CDATA #REQUIRED >
<!ELEMENT hieraMPI EMPTY>
<!ATTLIST hieraMPI
 operation (address|allgather|allgatherv|allreduce|
  alltoall|alltoallv|barrier|broadcast|bsend|
  bsend_init|buffer_attach|buffer_detach|cancel|
  comm_create|comm_dup|comm_split|gather|gatherv|
  get_count|get_elements|ibsend|intercomm_create|
  intercomm_merge|iprobe|irecv|irsend|isend|issend|
  pack|pack_size|probe|receive|recv_init|reduce|
  reduce_scatter|request_free|rsend|rsend_init|scan|
  scatter|scatterv|send|send_init|sendrecv|sendrecv_
  replace|ssend|ssend_init|start|startall|test|testall|
  testany|test_cancelled|testsome|type_commit|type_
  contiguous|type_extent|type_free|type_hindexed|
  type_hvector|type_indexed|type_lb|type_size|
  type_struct|type_ub|type_vector|unpack|wait|waitall|
  waitany|waitsome) #REQUIRED
 file          CDATA #REQUIRED
 line          CDATA #REQUERED
 start_time    CDATA #REQUERED
 finish_time   CDATA #REQUERED
 count         CDATA #IMPLIED
 type          CDATA #IMPLIED
 dest          CDATA #IMPLIED
 tag           CDATA #IMPLIED
 com           CDATA #IMPLIED
 ...
>
<!ELEMENT hieraOOPS (hieraOOPS | hieraMPI | EMPTY)*>
<!ATTLIST hieraOOPS
 class (unknown|distributionBlocked|distributionCyclic|
  distributionNone|matrix|vector|vectorRepl|vectorSequ|
  group|partner|topology|topologyGrid|topologyLinear|
  topologyPlain|topologyPipe|topologyTorus|
  workgroup) #REQUIRED
 method (allreduce|barrier|bcast|col|
  distributionBlocked|distributionCyclic|
  distributionNone|fromEast|fromNext|fromNorth|
  fromSouth|fromWest|fromNE|fromNW|fromSE|fromSW|
  gather|gatherv|globalToLocal|isInGroup|localSize|
  localToGlobal|matrix|max|min|partner|prod|recv|
  reduce|row|scatter|scatterv|send|split|store|
  subGroup|sum|syncGhostsCart|topologyGrid|
  topologyPipe|toEast|toNorth|toPrevious|toSouth|
  toWest|toNE|toNW|toSE|toSW|vector|vectorRepl|
  vectorSequ) #REQUIRED
 file          CDATA #REQUIRED
 line          CDATA #REQUERED
 start_time    CDATA #REQUERED
 finish_time   CDATA #REQUERED
 ...
>
```

O elemento raiz `processor` possue informações sobre o identificador (como *rank* MPI), o tempo inicial e final de execução do processador. O elemento raiz pode ter zero ou mais elementos do tipo *hieraMPI* ou *hieraOOPS*. Todas as operações executadas do tipo *hieraMPI* ou *hieraOOPS*, são filhas deste elemento (`processor`). Essas operações podem ser operações ponto-a-ponto ou coletivas em MPI ou podem ser chamadas de classes e métodos do OOPS. As informações de cada elemento dependem das operações que foram realizadas, podendo conter informações como o nome da operação, nome do arquivo, a linha onde a operação foi executada, os tempos inicial e final da operação. Para elementos OOPS, os nomes das classes e métodos são registrados.

Durante a execução do programa para a coleta de dados são gerados dois tipos de arquivos. Um arquivo de configuração que contém as informações de configuração sobre os arquivos de coleta, e um arquivo contendo as informações referentes a cada processo em execução.

3.2 Análise

O módulo *hieraTransform* lê os dados coletados dos vários níveis hierárquicos e constrói uma representação em memória da qual medidas podem ser computadas para a análise de desempenho da execução do programa. O *hieraTransform* pode ser dividido em duas etapas: transformação e medidas.

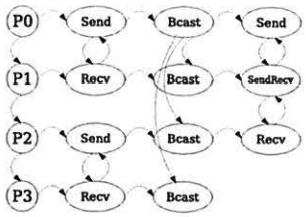

Figura 2. Ligações do Grafo

A fase de transformação lê os arquivos XML de cada processo gerados pelo módulo de coleta *hieraCollector* e constrói um grafo cujos vértices representam as operações (os elementos do arquivo XML) e as arestas representam as relações entre elas. A Figura 2 mostra um exemplo com 4 processadores (P0, ..., P3), onde por exemplo, P0 executou as operações *send, bcast* e *send*. As operações de comunicação têm seus respectivos vértices ligados pelas arestas.

As arestas do grafo possibilitam vários procedimentos de navegação a serem utilizados para a avaliação de desempenho. Uma possibilidade como mostra a Figura 3, é visitar todas as operações de cada processo apenas uma vez obtendo informações detalhadas sobre o processador e os tipos

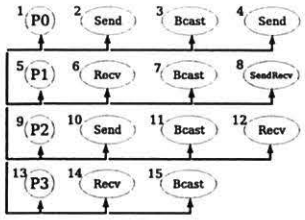

Figura 3. Percurso por Operações

de operações. A navegação é realizada na seguinte ordem: 1, 2, 3, 4, 5,. . ., 15, como mostra a figura.

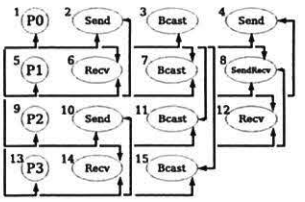

Figura 4. Percurso entre Operações Parceiras

Outra possibilidade é, a partir de uma operação navegar para as operações parceiras (como o *receive* correspondente de um *send*), antes de navegar para a próxima operação do mesmo processo, como mostra a Figura 4. Por exemplo na figura, a navegação é feita na seguinte ordem: 1, 2, 6, 3, 7, 11, 15, 4, 8, 5, 6, 2, 7, 8, 4, 12, 9, 10, 14, 11, 12, 8, 13, 14, 10, 15.

Na fase de medidas, a primeira tarefa é escolher qual a medida. Devido à grande quantidade de dados coletados durante a execução de programas paralelos, as medidas são geralmente de natureza estatística, como número de operações, médias, desvio padrão ou histogramas. As medidas são avaliadas navegando o grafo de uma maneira apropriada, ou seja, usando a navegação por operações da Figura 3 para contar o número de cada tipo de operação executada.

4 Experimentos

O *hieraCollector* foi implementado para MPI e OOPS. MPI (*Message Passing Interface*) foi escolhido devido ao seu amplo uso pela comunidade de programação paralela e sua disponibilidade para uma grande escala de máquinas, permitindo portabilidade de código e comunicação eficiente.

O *framework* OOPS (*Object-Oriented Parallel System*) é uma biblioteca de classes com o objetivo de suportar o desenvolvimento de aplicações científicas com uso extensivo de matrizes e vetores distribuídos. Suporta abstrações de alto nível para o desenvolvimento de códigos paralelos, sem ocultar completamente o paralelismo. Sua implementação é baseada no MPI. Por essa razão, ele se adequa aos testes para a ferramenta proposta neste trabalho, pois o usuário de OOPS desenvolverá o código baseado nas abstrações OOPS em vez das abstrações MPI.

Para testar a ferramenta em uma aplicação real, foi utilizado o programa de dinâmica molecular que implementa o algoritmo de decomposição de forças de Plimpton [17] para dinâmica molecular de partículas *Lennard-Jones*. Um dado número de partículas é distribuído em uma caixa tridimensional (com condições periódicas de contorno) de acordo com algumas condições iniciais para posição e velocidade que são especificadas. A evolução da posição e da velocidade das partículas é avaliada interativamente levando em consideração as interações entre as partículas e, essas interações são computadas usando o potencial de interação *Lennard-Jones* entre elas. A computação da interação de forças entre cada par de partículas é decomposta entre os processadores disponíveis, com as partículas distribuídas em blocos entre os processadores. Os processadores são organizados em uma grade bidimensional, e cada processador é responsável pela interação da partícula da mesma linha com a partícula da mesma coluna. Veja [17] para maiores explicações e outros detalhes do algoritmo. Esse algoritmo foi implementado na versão MPI e na versão OOPS.

Esse algoritmo foi executado em um cluster composto por oito nós de máquinas Pentium 4, 3.0 GHz, executando GNU/Linux.

4.1 HieraCollector para MPI e OOPS

O programa de dinâmica molecular na versão para o MPI e para o OOPS, foi executado com 4 processos. O arquivo de configuração XML gerado para MPI e OOPS, pode ser visto no quadro abaixo que tem `hieraCollector` como elemento raiz, o nome da aplicação indicado por `application` e o número de processos `count_proc`. O elemento `collect_file` possui os dados que são referentes a cada processo em execução. Por exemplo, a linha 4 mostra que o processo 0 (`rank=P0`) irá armazenar os seus dados coletados no arquivo chamado `trace0.xml`.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hieraCollector SYSTEM "hieraCollector.dtd">
3 <hieraCollector application="dinamica" count_proc="4">
4   <collect_file rank="P0">trace0.xml</collect_file>
5   <collect_file rank="P1">trace1.xml</collect_file>
6   <collect_file rank="P2">trace2.xml</collect_file>
7   <collect_file rank="P3">trace3.xml</collect_file>
8 </hieraCollector>
```

Parte do conteúdo do arquivo `trace0.xml` para a versão do programa em MPI é mostrado abaixo. O arquivo mostra o elemento raiz `processor` com o identi-

ficador do processo *rank=P0*, o tempo inicial e o tempo final da execução do processo 0. Os filhos do elemento raiz `processor` são várias operações MPI (todas do tipo *hieraMPI*) que correspondem aos campos das operações *broadcast*, *send*, *receive*, e aos campos para informações sobre a operação, como nome do arquivo, número da linha, tempo inicial e final, entre outros.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE processor SYSTEM "trace.dtd">
3 <?xml-stylesheet type="text/xsl" href="visual.xsl"?>
4 <processor rank="P0" init="0.01229" finalize="1.96906">
5  <hieraMPI operation="broadcast" file="md.c" line="37"
6   start_time="0.01992" finish_time="0.02001" count="1"
7   type="MPI_INT" root="0" comm="MPI_COMM_WORLD"/>
8  <hieraMPI operation="receive" file="auxfmd.c" line="51"
9   start_time="0.02540" finish_time="0.02542" count="1"
10  type="MPI_INT" rem="1" tag="0" comm="MPI_COMM_WORLD"/>
11  <hieraMPI operation="send" file="auxfmd.c" line="58"
12   start_time="0.02645" finish_time="0.02649"
13   count="1536" type="MPI_FLOAT" dest="1" tag="1"
14   comm="MPI_COMM_WORLD"/>
15  <hieraMPI operation="type_contiguous" file="md.c"
16   line="92" start_time="0.04312" finish_time="0.04313"
17   count="3" oldtype="MPI_FLOAT" newtype="P6_dtype"/>
18  <hieraMPI operation="type_commit" file="md.c" line="93"
19   start_time="0.04316" finish_time="0.04316"
20   type="P6_dtype"/>
21  <hieraMPI operation="scan" file="md.c" line="95"
22   start_time="0.04319" finish_time="0.04321" count="1"
23   type="MPI_INT" op="MPI_SUM" comm="MPI_COMM_WORLD"/>
24  <hieraMPI operation="allgatherv" file="md.c" line="137"
25   start_time="0.05551" finish_time="0.05568" scount="512"
26   stype="MPI_INT" rcount="512" rtype="MPI_INT"
27   comm="P5_comm"/>
28  <hieraMPI operation="reduce" file="md.c" line="171"
29   start_time="0.07606" finish_time="0.07609" count="1536"
30   type="MPI_FLOAT" op="MPI_SUM" root="1" comm="P5_comm"/>
31 ...
32 </processor>
```

O quadro abaixo mostra parte do arquivo `trace1.xml` gerado pela execução do processo de rank 1 da versão do código de dinâmica molecular do OOPS. O elemento raiz é `processor`, com as informações do *rank=P1* e o tempo inicial e final para o processo. Os elementos filhos são *hieraOOPS* para os métodos chamados durante a execução, com informações sobre as classes e métodos. Por exemplo, a linha 6 mostra a chamada para o construtor da classe `TopologyPipe`, implementado na linha 17 do arquivo `TopologyPipe.cc`, chamado com os argumentos `previous` e `next` como especificado.

A estrutura hierárquica do arquivo pode ser observada da linha 28 à linha 34. A chamada para o método `bcast` da classe `TopologyGrid` (linha 28) resulta na chamada do método `bcast` da classe `Group` (linha 31) que chama o elemento `hieraMPI` que é a execução da operação `broadcast` (linha 34).

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE processor SYSTEM "trace.dtd">
3 <?xml-stylesheet type="text/xsl" href="visual.xsl"?>
4 <processor rank="P1" init="0.000064"
5  finalize="134.766716">
6  <hieraOOPS file="TopologyPipe.cc" line="17"
7   class="OOPS::TopologyPipe" method="TopologyPipe"
8   previous="0" next="2" start_time="0.014255"
9   finish_time="0.014377"/>
10 <hieraOOPS file="TopologyGrid.cc" line="4273"
11  class="OOPS::TopologyGrid" method="split" color="0"
12  key="1" start_time="0.014401" finish_time="0.023591">
13   <hieraOOPS file="Topology.cc" line="2221"
14    class="OOPS::Group" method="split" color="0" key="1"
15    start_time="0.014440" finish_time="0.023584"/>
16 </hieraOOPS>
17 <hieraOOPS file="TopologyGrid.cc" line="4274"
18  class="OOPS::TopologyGrid" method="split" color="1"
19  key="1" start_time="0.023638" finish_time="0.031642">
20   <hieraOOPS file="Topology.cc" line="2221"
21    class="OOPS::Group" method="split" color="1" key="1"
22    start_time="0.023672" finish_time="0.031634"/>
23 </hieraOOPS>
24 <hieraOOPS file="molecularDynamicsOOPS.cc" line="0"
25  class="OOPS::TopologyGrid" method="TopologyGrid"
26  gsize="4" cols="2" start_time="0.014385"
27  finish_time="0.031741"/>
28 <hieraOOPS file="molecularDynamicsOOPS.cc" line="45"
29  class="OOPS::TopologyGrid" method="bcast" root="0"
30  start_time="0.031749" finish_time="0.031878">
31   <hieraOOPS file="Topology.cc" line="2335"
32    class="OOPS::Group" method="bcast" root="0"
33    start_time="0.031788" finish_time="0.031872">
34     <hieraMPI operation="broadcast" file="Topology.cc"
35      line="2335" count="1" type="MPI_INT" root="0"
36      comm="OOPS_Comm1" start_time="0.031788"
37      finish_time="0.031866"/>
38   </hieraOOPS>
39 </hieraOOPS>
40 <hieraOOPS file="molecularDynamicsOOPS.cc" line="46"
41  class="OOPS::TopologyGrid" method="bcast" root="0"
42  start_time="0.031885" finish_time="0.032002">
43   <hieraOOPS file="Topology.cc" line="2335"
44    class="OOPS::Group" method="bcast" root="0"
45    start_time="0.031918" finish_time="0.031995">
46     <hieraMPI operation="broadcast" file="Topology.cc"
47      line="2335" count="1" type="MPI_INT" root="0"
48      comm="OOPS_Comm1" start_time="0.031918"
49      finish_time="0.031989"/>
50   </hieraOOPS>
51 </hieraOOPS>
52 <hieraOOPS file="molecularDynamicsOOPS.cc" line="47"
53  class="OOPS::TopologyGrid" method="bcast" root="0"
54  start_time="0.032008" finish_time="0.032123">
55   <hieraOOPS file="Topology.cc" line="2335"
56    class="OOPS::Group" method="bcast" root="0"
57    start_time="0.032043" finish_time="0.032118">
58     <hieraMPI operation="broadcast" file="Topology.cc"
59      line="2335" count="1" type="MPI_INT" root="0"
60      comm="OOPS_Comm1" start_time="0.032043"
61      finish_time="0.032112"/>
62   </hieraOOPS>
63 </hieraOOPS>
64 ...
65 </processor>
```

4.2 HieraTransform para MPI e OOPS

Para demonstrar ferramentas de análise baseadas na estrutura de grafo gerada dos arquivos coletados descritos, algumas aplicações foram desenvolvidas para extrair alguns dados de desempenho estatísticos. Uma delas conta o número de operações de cada tipo em cada nível hierárquico; outra computa o número de operações de cada tipo discriminado pelo processador e entre operações MPI e OOPS; a terceira aplicação simplesmente coleta o número total de cada operação executada; e por último uma outra aplicação computa o tempo total de comunicação e

computação para cada processador.

Avaliação do Programa de Dinâmica Molecular MPI

A tabela 1 mostra a contagem do número de operações do nível hierárquico 0 da aplicação MPI. A coluna `Processos` mostra os processos envolvidos na execução da aplicação. A coluna `Nível` mostra o nível na hierarquia, neste caso existe apenas o nível 0. A coluna `Contador` mostra o número de operações contidas no nível 0. Todos os processos executaram o mesmo número de operações, com o processo 0 executando cerca de 10% mais operações que os demais processos.

Tabela 1. Número de operações na versão MPI em cada nível da hierarquia

Processos	Nível	Contador
0	0	2396
1	0	2144
2	0	2144
3	0	2144

A tabela 2 mostra mais informações, listando o nível de abstração que a coleta foi realizada (coluna `Coleta`), as operações realizadas (coluna `Operação`), o nome do arquivo onde a operação foi executada (coluna `Arquivo`), a linha onde a operação foi executada no código fonte (coluna `Linha`) e o número de vezes que cada operação foi executada (coluna `Cont`). Devido a quantidade de informações coletadas, somente as operações do processo 0 são mostradas na tabela. As operações *reduce* nas linhas 171 e 175 do arquivo `md.c` são as operações mais executadas pelo processo 0.

A tabela 3 mostra os tipos de operações executadas (por todos os processos). A coluna `Operação` apresenta a lista com as operações que foram executadas na aplicação e a coluna `Contador` mostra o número de vezes que cada operação foi realizada. Nesta tabela, pode ser visto a quantidade de operações de *reduce* que foram executadas por todos os processos.

A tabela 4 apresenta os tempos total de comunicação e o tempo total gasto pelos diferentes processos durante a execução da aplicação de dinâmica molecular. A coluna `Processos` mostra os processos envolvidos na execução da aplicação. A coluna `Tempo de Comunicação` mostra o tempo de comunicação gasto por cada processo durante a execução e a coluna `Tempo Total` apresenta o tempo total gasto pelos processos na execução da aplicação. É possível observar que o tempo de comunicação é uma fração importante do tempo total.

Para se ter uma idéia da influência da instrumentação no tempo de execução, a tabela 5 apresenta os tempos totais

Tabela 2. Número de operações listadas por operações individuais na versão MPI do código dinâmica molecular

Coleta	Operação	Arquivo	Linha	Cont
mpi	allgather	md.c	107	1
mpi	allgather	md.c	108	1
mpi	allgatherv	md.c	137	1
mpi	allgatherv	md.c	139	1
mpi	allgatherv	md.c	150	200
mpi	allgatherv	md.c	156	200
mpi	broadcast	md.c	37	1
mpi	broadcast	md.c	38	1
mpi	broadcast	md.c	39	1
mpi	broadcast	md.c	40	1
mpi	broadcast	md.c	41	1
mpi	broadcast	md.c	42	1
mpi	broadcast	md.c	43	1
mpi	broadcast	md.c	44	1
mpi	broadcast	md.c	45	1
mpi	comm_split	md.c	101	1
mpi	comm_split	md.c	102	1
mpi	receive	auxfmd.c	51	9
mpi	receive	auxfmd.c	90	180
mpi	receive	auxfmd.c	92	180
mpi	reduce	md.c	171	400
mpi	reduce	md.c	175	400
mpi	reduce	md.c	183	200
mpi	reduce	md.c	184	200
mpi	reduce	md.c	185	200
mpi	reduce	md.c	186	200
mpi	scan	md.c	95	1
mpi	send	auxfmd.c	58	9
mpi	type_commit	md.c	93	1
mpi	type_contiguous	md.c	92	1

Tabela 3. Número de operações listadas por operação individual na versão MPI do código de dinâmica molecular

Operação	Contador
allgather	8
allgatherv	1608
broadcast	36
comm_split	8
receive	378
reduce	6400
scan	4
send	378
type_commit	4
type_contiguous	4

obtidos com a execução da aplicação de dinâmica molecular usando a versão do código com e sem a instrumentação. A diferença entre os tempos obtidos usando as duas versões foi de 6%.

Tabela 4. Tempo total de execução e tempo de comunicação (em segundos) para os quatro processos do código de dinâmica molecular MPI

Processos	Comunicação (seg)	Tempo Total (seg)
0	0.495700	1.784300
1	0.915030	1.753440
2	0.991050	1.753510
3	0.972300	1.753490

Tabela 5. Influência do código instrumentado no tempo de execução da versão MPI do código de dinâmica molecular (tempo em segundos)

Processos	Com Trace (seg)	Sem Trace (seg)
0	1.784300	1.676900
1	1.753440	1.655580
2	1.753510	1.655580
3	1.753490	1.655630

Avaliação do Programa de Dinâmica Molecular OOPS

A mesma análise é feita para a versão do código de dinâmica molecular usando o OOPS. É utilizado o mesmo algoritmo só que implementado usando as primitivas do OOPS.

A tabela 6 mostra o número de operações executadas em cada um dos níveis de abstração 0, 1 e 2. É possível notar um número muito maior de operações realizadas do que as operações no código MPI. Isto se deve ao fato de que a versão atual do OOPS não tem nenhuma operação de redução em seções de *arrays*, como apresentado no MPI, e portanto, as reduções devem ser executadas em um *loop* para cada partícula.

Tabela 6. Número de operações em cada nível hierárquico para o código OOPS de dinâmica molecular

Processos	Nível 0	Nível 1	Nível 2
0	101756	165476	100811
1	101756	100948	100811
2	101756	100948	100811
3	101756	100948	100811

A tabela 7 discrimina o número de operações executadas e o nível de abstração em que a coleta foi realizada (MPI e OOPS) para o processo 2. Pôde ser observado nesta tabela que, a maioria das operações executadas são operações MPI de redução (um total de 100000) requisitadas a partir das operações *sum* nas linhas 181 e 188 do arquivo fonte `molecularDynamicOOPS.cc`.

Tabela 7. Discriminação do número de operações para o processo 2 da versão OOPS do códigode dinâmica molecular

Coleta	Operação	Arquivo	Linha	Cont
mpi	allgather	Topology.cc	2655	2
mpi	allgatherv	Topology.cc	2622	2
mpi	allreduce	Topology.cc	3672	200
mpi	allreduce	Topology.cc	3837	600
mpi	broadcast	Topology.cc	2335	5
mpi	broadcast	Topology.cc	2423	3
mpi	broadcast	Topology.cc	2434	1
mpi	reduce	Topology.cc	3892	100000
mpi	scan	Topology.cc	4202	1
oops	broadcast	molecularDynamicsOOPS.cc	45	1
oops	broadcast	molecularDynamicsOOPS.cc	46	1
oops	broadcast	Topology.cc	2423	3
oops	broadcast	Topology.cc	2434	1
oops	distributionBlocked	Application	0	1
oops	gather	Topology.cc	2622	2
oops	gather	Topology.cc	2655	2
oops	gather	Topology.h	3019	60
oops	load	Vector.h	1606	4
oops	localSize	Vector.h	1285	4
oops	localSize	Vector.h	1531	4
oops	scan	molecularDynamicsOOPS.cc	105	1
oops	scan	Topology.cc	4202	1
oops	scatter	Vector.h	1531	4
oops	scatter	Topology.h	3045	4
oops	split	Topology.cc	2221	2
oops	split	TopologyGrid.cc	4273	1
oops	store	Vector.h	1613	60
oops	sum	molecularDynamicsOOPS.cc	181	50000
oops	sum	molecularDynamicsOOPS.cc	188	50000

O resultado resumido por tipo de operação para todos os processos são mostrados na tabela 8. A maior parte da contagem das operações é denominada por operações de *sum* (e a correspondente *reduce*), seguida pelas operações *localSize* e *localToGlobal*, responsáveis pela verificação do tamanho da parte local do *array* e a conversão do *array* de índices de local para global.

Tabela 8. Contagem das operações para todos os processos da versão OOPS do código de dinâmica molecular

Operação	Contador	Operação	Contador
allgather	8	load	16
allgatherv	8	localSize	32800
allreduce	3200	localToGlobal	32016
broadcast	36	scatter	32
reduce	400000	split	16
scan	12	store	240
broadcast	72	sum	806400
distribBlocked	4	topologyGrid	4
gather	3712	topologyPipe	12

Finalmente, o efeito da instrumentação no tempo de execução do código é apresentada na Tabela 9. Pode ser visto que a sobrecarga é de aproximadamente 0.8%, o que não é um valor pequeno, devido ao alto número de operações registradas.

5 Conclusões

Devido ao crescente uso de níveis mais alto de abstração para o desenvolvimento de códigos paralelos, é impor-

Tabela 9. Tempo de execução (em segundos) com e sem instrumentação para a versão OOPS do código de dinâmica molecular

Processos	Com Trace (seg)	Sem Trace (seg)
0	239.407456	237.457
1	239.288433	237.458
2	239.288600	237.459
3	239.288144	237.471

tante o uso de ferramentas de desempenho que levam em consideração estes níveis e que são capazes de gerar informações no nível de abstração entendido pelo desenvolvedor da aplicação.

Este artigo apresentou a ferramenta *HieraAnalyses*, desenvolvida para investigar a viabilidade de tais tipos de ferramentas. A ferramenta é composta de um módulo de coleta *hieraCollector* e de um módulo de transformação *hieraTransform*. O módulo de coleta armazena informações de desempenho em um arquivo XML com a estrutura hierárquica seguindo as estruturas das chamadas hierárquicas da execução da aplicação. O módulo de transformação gera um grafo dos dados coletados, o qual vários tipos de análises podem ser realizadas.

O artigo descreveu aplicações adicionais da ferramenta para a análise do código paralelo de dinâmica molecular escrito em duas versões: uma usando apenas operações em MPI e outra usando o *framework* OOPS de alto nível implementado usando MPI. Foi mostrado que informações importantes sobre a execução do programa e as partes do código que requerem atenção especial podem ser deduzidas do grafo que representa as informações de desempenho.

Um uso mais profundo das informações hierárquicas coletadas envolvem a visualização das informações seguindo a estrutura hierárquica apresentada nos dados. Isto pode permitir ao usuário encontrar importantes seções do código em uma abordagem *top-down*, uma sugestão para trabalhos futuros.

Referências

[1] Pablo - Scalable Performance Tools. http://www-pablo.cs.uiuc.edu/.

[2] PVM - Parallel Virtual Machine. http://www.netlib.org/pvm3.

[3] W3C: World Wide Web Consortium. http://www.w3c.org.

[4] R. A. Aydt. *An Informal Guide to Using Pablo*. Department of Computer Science - University of Illinois, May 12 1992.

[5] Barton P. Miller. *Paradyn Parallel Performance Tools - VisiLib Programmer's Guide*, release 4.2 edition, March 2005.

[6] L. S. Blackford, J. J. Dongarra, and R. C. Whaley. *ScaLAPACK Users' Guide*. Siam–Society for Industrial and Applied Mathematics, May 1997. ISBN 0-89871-397-8.

[7] S. Browne, J. Dongarra, and K. London. Review of Performance Analysis Tools for MPI Parallel Programs. *NHSE Review*, 3(1), 1998.

[8] A. Chan, W. Gropp, and E. Lusk. User's Guide for MPE Extensions for MPI Programs. Technical report, Argone National Laboratory - University of Chicago, 1998.

[9] C. Fineman, M. F. P. Hontalas, M. Hribar, and H. Jin. *The Automated Instrumentation and Monitoring System*. NASA Ames Research Center, version 3.7 edition, January 1997. http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/manual/ (visitada Maio/2008).

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.

[11] J. K. Hollingsworth, R. B. Irvin, and B. P. Miller. The Integration of Application and System Based Metrics in a Parallel Program Performance Tool. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 189–200, Williamsburg, VA, April 1991.

[12] J. Labarta, J. Gimenez, J. Caubet, and F. Escale. *PARAVER: Parallel Program Visualization and Analysis Tool*. European Center for Parallelism of Barcelona, October 2001. Version 3.1.

[13] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 12 1995.

[14] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, February 1990.

[15] S. Moore, D. Cronk, K. London, and J. Dongarra. Review of Performance Analysis Tools fo MPI Parallel Programs. In *8th European PVM/MPI Users' Group Meeting*, pages 241–248, 2001.

[16] B. P. Miller. Paradyn Parallel Performance Tools. http://www.paradyn.org.

[17] S. Plimpton. Fast Parallel Algorithms for Short-range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[18] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

[19] F. A. Rodrigues and G. Travieso. Técnicas de Orientação ao Objeto para Computação Científica Paralela. Master's thesis, Instituto de Física de São Carlos - Universidade de São Paulo, April 2004.

[20] E. Sonoda and G. Travieso. The OOPS Framework: High Level Abstractions for the Development of Parallel Scientific Applications. In *OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 659–660, New York, NY, USA, 2006. ACM Press.

[21] J. C. Yan. Performance Tuning with AIMS — An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 625–633, January 1994.