GeneGrade: Um algoritmo genético para escalonamento de aplicações em grades computacionais

Leonardo de Assis
Universidade Federal de Campina Grande
leonardooo@gmail.com

Pablo Riccelly
Universidade Federal de Campina Grande
pabloriccelly@gmail.com

Resumo

O escalonamento de aplicações em uma grade computacional é uma das áreas mais desafiadoras devido a uma série de fatores, velocidade de máquinas e links, regras de segurança diferentes para domínios admistrativos diferentes e etc. Neste artigo é apresentado o GeneGrid, um escalonador de aplicações em grades computacionais que utiliza um algoritmo genético que associa tarefas de uma aplicação à recursos de uma forma inteligente visando a otimização do tempo de execução desta aplicação.

1. Introdução

A necessidade de executar aplicações que exigem cada vez mais poder computacional levou ao homem a criar soluções que quebrasse o limite de apenas uma máquina. Uma solução são as grades computacionais. Utilizando grades computacionais, aplicações que executam paralelamente têm um grande ganho de performance. Como uma grade computacional é escalável, máquinas adicionais podem ser colocadas na grande para aumentar o poder computacional. Com isso, uma grade computacional se torna uma solução muito barata.

Mas a execução de uma aplicação numa grade computacional não é tão trivial assim, um dos desafios desta área é o escalonamento de aplicações. Um escalonamento define qual parte da aplicação irá executar em tal recurso. Um mau escalonamento da aplicação pode ocasionar em vários problemas como, desbalanceamento de carga, partes grandes da aplicação sendo escalonada para um recurso com baixo poder computacional ou com um link congestionado e uma série de outros fatores.

Neste artigo é apresentado um escalonador batizado de GeneGrade. O GeneGrade utiliza um algoritmo genético que associa partes da aplicações (tarefas) à recursos (máquinas) visando a otimização do tempo de execução da aplicação. O algoritmo associa tarefas à máquinas de forma inteligente, visando um bom balanceamento de carga na grade computacional, conseqüentemente reduzindo o tempo de execução da aplicação.

O resto deste artigo está organizado da seguinte forma: uma breve descrição de algoritmos genéticos é descrita na Seção 2, na Seção 3 é apresentado o algoritmo genético do GeneGrade, na Seção 4 é mostrado os resultados obtidos através de simulações para avaliação do GeneGrade, a conclusão está na Seção 5 seguido dos anexos deste artigo que se encontram na Seção 6.

2. Algoritmos Genéticos

Os algoritmos genéticos são algoritmos que otimizam de alguma forma uma determinada solução de algum problema utilizando um modelo que tenta "imitar" a evolução humana.

Um algoritmo genético é composto por várias etapas. A primeira etapa deste algoritmo visa gerar uma população inicial. Na segunda etapa os individuos desta população são avaliados através de uma função *fitness* que indica o quão bom é este individuo. Após a avaliação, o processo de evolução começa. O primeiro passo no processo de evolução é selecionar os melhores individuos através da função *fitness* de forma que eles sobrevivam e ocorra a reprodução, destes individuos sobreviventes, gerando descendentes. Após isto, alguns individuos são escolhidos e estes são modificados através do processo de mutação.

Finalmente, os individuos novamente são avaliados e selecionados e o ciclo novamente é executado. Estas etapas (seleção, reprodução, mutação) são aplicadas iterativamente até que uma condição de parada seja obedecida.

3. GeneGrade

Nas próximas sub-sessões são descritas como são feitas todas as etapas da evolução dos individuos (cromossomos) bem como a representação de um cromossomo para o nosso problema.

3.1. Cromossomo

A modelagem de um cromossomo no GeneGrade é descrito através de uma lista de Execuções. Execução é uma abstração que associa uma tarefa à uma máquina. Conseqüentemente, o cromossomo é o mapeamento do escalonamento.

3.2. Criação

A população inicial é criada de acordo com associações aleatórias de tarefas à máquinas. São criados dez cromossomos desta forma.

3.3. *Crossover*

A reprodução dos cromossomos é feita da seguinte forma, cinco casais dentro os dez melhores cromossomos são selecionados de forma aleatória e então parte de um cromossomo é utilizado e parte do outro cromossomo restante é utilizado, gerando um descendente. Estas partes são selecionadas de forma aleatória. Após todas as reproduções, os cromossomos são avaliados e ordenados com base da função *fitness*.

3.4. Mutação

Após a etapa *crossover* ser executada, os cinco piores cromossomos são submetidos a um processo de mutação. A mutação é feita através de trocas aleatórias de máquinas e tarefas de um mesmo cromossomo. Após isso, novamente os cromossomos são ordenados e o ciclo é novamente iniciado.

3.5. *Fitness*

O resultado da função *fitness* é calculado de acordo com a simulação do escalonamento do cromossomo. Este resultado é o tempo de execução da aplicações. Então quanto menor tempo de execução possui um cromossomo, melhor ele é.

3.6. Parada

A condição de parada do algoritmo é atingida quando em três processos de evolução, o melhor cromossomo possui o mesmo resultado para a função *fitness*, ou seja, o algoritmo se estabilizou.

4. Simulações

Para avaliar o GeneGrade, foi implementado um simulador em Java utilizando o modelo de redes WAN de Casanova, modelo também utilizado no simulador de grades SimGrid.

Como forma de comparar o desempenho do GeneGrade com outra heurística, também foi implementada no simulador a clássica heurística de escalonamento *Workqueue*. Esta heurística associa tarefas da aplicação à máquinas de forma seqüencial, ou seja, máquina 1 para tarefa 1, máquina 2 para tarefa 2 e assim por diante.

As duas heurísticas foram comparadas através de dois tipos de cenários diferentes, estes são mostrados nas seguintes sub-sessões.

3.1. Máquinas e Tarefas Heterogêneas

Neste tipo de simulação foi executado um total de três cenários para cada escalonador. Um cenário para tarefas e máquinas com grau de heterogeneidade baixo, outro para um grau médio e outro para um grau alto. O gráfico a seguir mostra os tempos de execução de cada escalonador para cada cenário simulado.

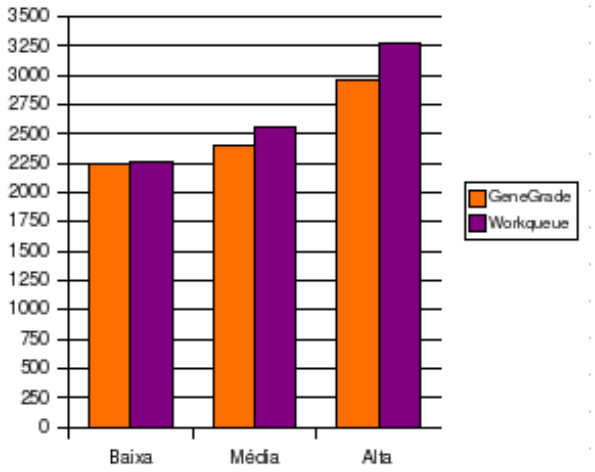

Figura 1: Resultados das simulações para máquinas e tarefas heterogêneas

Note que, de acordo com o gráfico, quanto maior o grau de heterogeneidade, melhor o desempenho entre o GeneGrade e o Workqueue. Isto se dá pelo fato do GeneGrade escolher melhor os recursos para as tarefas.

3.2. Máquinas Homogêneas e Tarefas Heterogêneas

Neste tipo de simulação também foram executados três cenários para cada escalonador. Um cenário para tarefas com grau de heterogeneidade baixo, outro para um grau médio e outro para um grau alto. Nos três cenários, as máquinas eram todas homogêneas. O gráfico a seguir mostra os tempos de execução de cada escalonador para cada cenário simulado.

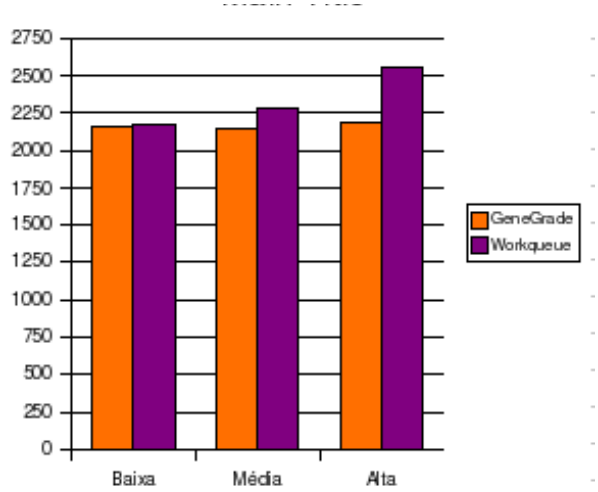

Figura 2: Resultados das simulações para máquinas homogêneas e tarefas heterogêneas

De acordo com o gráfico mostrado, mais uma vez quanto maior a heterogeneidade, melhor o desempenho. O GeneGrande neste cenário fez um balanceamento melhor das tarefas.

4. Conclusão

Este artigo apresentou o GeneGrade, um algoritmo genético para escalonamento de aplicações em grades computacionais. Foi implementado um simulador no intuito de comparar o GeneGrade com outra heurística de escalonamento clássica. De acordo com as simulações feitas, o GeneGrade se encaixa melhor em ambientes heterogêneos e/ou com aplicações com as tarefas heterogêneas do que o *Workqueue*.

Anexos

COMO USAR O SIMULADOR SimGrade4J

Requisitos:

- Java 1.5.x ou superior

Para Executar:

- No prompt da linha de comando no diretório raiz do simulador:

leonardo@computer:~ java -cp bin/ ui.Simulador <Heurística> <Cenário> > log.txt

onde:

- <Heurística> - Workqueue ou GeneGrade
- <Cenário> - Arquivo que descreve um cenário (podem ser encontrados alguns exemplos no diretório cenários)
- > log.txt - Se você quizer guardar o log da simulação, redirecione a saída padrão p/ um arquivo
-

Como gerar um cenário:

- No prompt da linha de comando no diretório raiz do simulador:

leonardo@computer:~ java -cp bin/ util.GeradorCenario <NomeDoArquivo> <MAQ> <VL> <VLV> <VM> <VMV> <TAR> <T> <TV> <VT> <VTV>

onde:

- <NomeDoArquivo> - Nome do arquivo do cenário a ser gerado
- <MAQ> - Número de máquinas na grade
- <VL> - Vazão dos links das máquinas
- <VLV> - Variância das vazões dos links das máquinas
- <VM> - Velocidade das máquinas
- <VMV> - Variância das velocidades das máquinas
- <TAR> - Número de tarefas da aplicação
- <T> - Tamanho das tarefas
- <TV> - Variância do tamanho das tarefas
- <VT> - Vazão da tarefa
- <VTV> - Variância das vazões das tarefas

- Se você desejar, pode montar um cenário "na mão" mesmo. Existe alguns exemplos na pasta cenários no diretório raiz do
simulador. Você deve seguir rigorosamente a sintaxe.

Código Fonte

```
private void criacao() {
  for(int i = 0; i < 10; i++) {
    cromossomos[i] = cria_cromossomo_random();
  }
  fitness();
  ordena();
}

private List<Execucao> cria_cromossomo_random() {
  List<Execucao> cromossomo = new LinkedList<Execucao>();
  List maquinas_list = new LinkedList( maquinas );
  List tarefas_list = new LinkedList( tarefas );

  int cont_execucao_id = 1;
  int cont_maquina_id = 0;
  while(tarefas_list.size() > 0) {
     int tarefa_index = (int)(Math.random() * tarefas_list.size());
     Maquina m = (Maquina) maquinas_list.get(cont_maquina_id++);
     Tarefa t = (Tarefa) tarefas_list.remove( tarefa_index );
     Execucao e = new Execucao(cont_execucao_id++, m, t);
     cromossomo.add( e );
    if(cont_maquina_id >= maquinas_list.size())
        cont_maquina_id = 0;
     }
     return cromossomo;
}


private void cruzamento() {
  int cont = 10;
  for(int i = 0; i < 5; i++) {
  int c1_index = (int)(Math.random() * 10);
  int c2_index = (int)(Math.random() * 10);
   List<Execucao> cromossomo_resultante = gera_descendente(
cromossomos[c1_index], cromossomos[c2_index] );
  cromossomos[cont++] = cromossomo_resultante;
  }
  fitness();
  ordena();
}

private List<Execucao> gera_descendente( List<Execucao> crom1,
List<Execucao> crom2 ) {
  List<Execucao> resultante = new LinkedList<Execucao>();
 List<Execucao> clone_crom1 = new LinkedList<Execucao>(crom1);
 List<Execucao> clone_crom2 = new LinkedList<Execucao>(crom2);
   for(int i = 0; i < clone_crom1.size(); i++) {
     Execucao crom1_exe = clone_crom1.get(i);
   Execucao crom2_exe = clone_crom2.remove( ((int)(Math.random()
* clone_crom2.size())) );
     resultante.add( new Execucao(i+1, crom1_exe.getMaquina(),
crom2_exe.getTarefa()) );
   }
  return resultante;
}
```

```
private void mutacao() {
   for(int i = 14; i >= 10; i--) {
   List<Execucao> cromossomo = cromossomos[i];
   for(int j = 0; j < cromossomo.size() / 3; j++) {
      int index1 = (int)(Math.random() * cromossomo.size());
      int index2 = (int)(Math.random() * cromossomo.size());

     while(index2 == index1)
        index2 = (int)(Math.random() * cromossomo.size());

      if(index1 > index2) {
         int temp = index2;
         index2 = index1;
         index1 = temp;
      }

      Execucao e1 = cromossomo.remove(index1);
      Execucao e2 = cromossomo.remove(index2 - 1);
      Execucao novo_e1 = new Execucao( e1.getId(), e1.getMaquina(),
e2.getTarefa() );
      Execucao novo_e2 = new Execucao( e2.getId(), e2.getMaquina(),
e1.getTarefa() );

      cromossomo.add(novo_e1);
      cromossomo.add(novo_e2);
    }

   fitness();
   ordena();
}
```