5
Aplicações

Neste capítulo relatamos algumas experiências que realizamos com o LuaCharm. Em primeiro lugar, descrevemos implementações distintas de um problema clássico, para medir o impacto de desempenho de uso do modelo híbrido que propomos. Descrevemos também experiências de integração com o balanceamento de carga do Charm++ e com a carga dinâmica de código em um chare.

Para a realização dos testes foi usado um cluster de computadores Pentium 400MHz, ligados por uma rede Ethernet 100Mbps e utilizando o sistema operacional Linux.

5.1
Testes de desempenho

Para o teste de desempenho foi escolhido uma aplicação *embaraçosamente paralela* (25). Uma aplicação embaraçosamente paralela é constituída de tarefas independentes e sua paralelização é trivial uma vez que as tarefas tenham sido definidas, pois não é necessária comunicação alguma entre elas. Diversas classes de aplicações como soma vetorial, ray-tracing, computação branch-and-bound entre outras, caem nessa categoria.

O problema utilizado nos testes de desempenho foi o da quadratura adaptativa (22), calculando a área da função e^x no intervalo de 0 a 15 com uma tolerância de 10^{-16}. Para a implementação do problema, foram desenvolvidas duas abordagens diferentes. A primeira foi a implementação utilizando bolsa de tarefas (23), implementada por um chare, e consumidores de tarefas, implementados por um grupo, que mantém uma instância do tipo em cada processador. A segunda abordagem foi a criação de um array, onde cada elemento fica responsável por um pequeno pedaço do intervalo.

A primeira abordagem sugerida exige um papel maior do controlador, responsável pela bolsa de tarefas, que deve enviar novas tarefas a chares que não estão em execução e deve receber tarefas dos chares quando elas forem subdivididas. Essa implementação dispensa o uso de um balanceador de carga, já que o controlador é o responsável por garantir que os chares estão sempre

	1. Impl	2. Impl	3. Impl	4. Impl
Tempo (s)	29,75s	29,85s	168,77s	30,59s
Acréscimo em relação a 1.		0,35%	467,36%	2,82%

Tabela 5.1: Primeira abordagem

ocupados com o cálculo de um intervalo da curva.

Uma das vantagens da implementação do controlador em Lua, é a possibilidade de redefinir o funcionamento do controlador através da carga dinâmica de código. Apesar de não ter sido explorado nesse teste, a redefinição de qualquer método seria trivial pela interface disponível no binding. Essa característica é especialmente útil em aplicações com longo tempo de execução.

A segunda implementação, por sua vez, demanda uma participação menor do controlador, que criará o array já com suas tarefas pré determinadas. Seu papel então será coletar os resultados dos elementos do array e terminar a execução quando todos enviarem os seus resultados. Para esta segunda solução é recomendável o uso de um balanceador de carga, pois o esforço do cálculo de cada intervalo não é uniforme e de acordo com a divisão dos chares entre processadores pode ocorrer uma sobrecarga de alguns deles.

Para cada abordagem, foram feitas quatro implementações diferentes do problema:

- Implementação padrão em Charm++;
- Implementação do controlador em Lua e dos consumidores (primeira abordagem) e array (segunda abordagem) em C++;
- Implementação de ambos os chares em Lua;
- Implementação de ambos os chares em Lua, mas o código Lua utiliza uma biblioteca implementada em C para o processamento matemático.

No apêndice A é mostrado o código dos testes feitos.

5.1.1 Primeira abordagem

A primeira abordagem, utilizando um chare para bolsa de tarefas e um grupo para consumidores, foi testada no cluster utilizando oito processadores. O número inicial de tarefas (partições do intervalo) foi de 16, sendo que após o início do cálculo, mais tarefas são criadas dinamicamente pelos chares, através da subdivisão de um intervalo grande. Os resultados são apresentados na Tabela 5.1.

	1. Impl	2. Impl	3. Impl	4. Impl
Tempo (s)	10,01s	10,01s	67,83s	10,01s
Acréscimo em relação a 1.		0%	577,66%	0,03%

Tabela 5.2: Segunda abordagem

5.1.2 Segunda abordagem

A segunda abordagem, utilizando um array de chares, onde cada um é responsável pelo cálculo de uma parte da aplicação, foi testada no cluster utilizando oito processadores. O número de elementos do array foi de 500. Os resultados são apresentados na Tabela 5.2.

Como dito anteriormente, o balanceador de carga só atua nos chares que não estão em execução. Na nossa implementação, a iteração sobre o intervalo acontece recursivamente e a execução é iniciada através da chamada de um método do array global. Assim, quando o balanceador estivesse ativo, todos os chares já estariam em execução ou já teriam finalizado. Duas soluções foram propostas: a primeira seria a implementação de uma iteração sobre o intervalo na qual o método responsável pelo cálculo, no lugar de fazer uma recursão, salvasse o seu estado em uma variável global e chamasse o próximo passo da iteração através de uma chamada assíncrona para o próprio chare. A segunda possibilidade é que a execução dos intervalos seja feita em etapas, e, quando uma etapa for finalizada, a seguinte seja iniciada pelo coordenador. Para este teste escolhemos a segunda opção.

5.1.3 Discussão dos resultados

Os resultados dos testes acima demonstram, como esperado, que o gargalo da aplicação está na parte de processamento matemático. Quando os chares foram totalmente implementados em Lua, tivemos resultados muito mais lentos que uma implementação totalmente em C, confirmando que as linguagens de script perdem muito em eficiência para linguagens compiladas tradicionais. Mas quando tivemos a parte de processamento matemático implementada em C, tanto através do uso de uma biblioteca carregada de Lua, quanto através da implementação do chare em C++, mas com o controlador em Lua, o resultado observado foi de uma perda de desempenho muito pequena.

Podemos observar que os testes apresentam uma grande diferença no tempo de execução entre a primeira e a segunda abordagem. Essa diferença se dá pela maneira que o algoritmo foi implementado. Na primeira abordagem,

cada vez que um chare trabalhador busca uma tarefa, ele vai avaliar o tamanho do intervalo e particioná-lo caso ele seja muito grande. Uma das partições será calculada no próprio chare e a outra será enviada de volta para a bolsa de tarefas. Como o número iniciais de partições é pequeno, o número de mensagens entre os chares é muito grande, o que provoca um grande aumento no tempo de execução. Isso poderia ser melhorado com um particionamento melhor no início da execução. Como nestes testes a implementação ótima do algoritmo não é o nosso objetivo, mas sim a comparação entre as diferentes implementações, não abordamos este problema.

5.2 Integração com o balanceador de carga

Conforme descrito anteriormente, arrays de chares têm uma forte integração com o balanceador de carga, podendo ser migrados de processador em qualquer momento que não estejam executando um método. Na seção anterior, implementamos uma aplicação que poderia sofrer ação do balanceador de carga, mas como não era necessário guardar dados entre os métodos, não foi preciso utilizar o mecanismo de migração de dados de Lua.

Para que essa migração possa ser feita, o gerador de código implementa o método *pup* do array, que é chamado no chare antes e após a sua migração. O código gerado de pup é responsável por chamar as funções equivalentes em Lua para fazer a serialização e desserialização dos dados, que são `pack` e `unpack`. Para um array chamado Calc, o método `pup` seria o seguinte:

```
void Calc::pup(PUP::er &p){
  CBase_Calc::pup(p); // chama o pup do proxy
  int len;
  char* data;
  // Se está recuperando dados serializados
  if (p.isUnpacking()){
    // Busca a função Calc.unpack
    lua_getglobal(L, "Calc");
    lua_pushstring(L, "unpack");
    lua_gettable(L, -2);
    // Se Calc.unpack não existir, retorna
    if (lua_isnil(L, -1)){
      lua_pop(L, 2);
      return;
    }
    // Pega valor da variável len
```

```
    p|len;
    data = new char[len];
    // Pega string data de tamanho len
    p(data, len);
    lua_pushstring(L, data); // Coloca a string data na pilha
    // Chama Calc.unpack passando data como parametro
    lua_pcall(L, 1, 0, 0);
    lua_pop(L, 1);
  } else if (!p.isUnpacking()){ // Está serializando os dados
    // Busca a função Calc.pack
    lua_getglobal(L, "Calc");
    lua_pushstring(L, "pack");
    lua_gettable(L, -2);
    if (lua_isnil(L, -1)){
      lua_pop(L, 2);
      return;
    }
    // Chama Calc.pack esperando 1 resultado
    lua_pcall(L, 0, 1, 0);
    data = (char*) lua_tostring(L, -1);
    lua_pop(L, 1);
    len = strlen(data);
    // Exporta os dados das variávels len e data
    p|len;
    p(data, len);
  }
}
```

O código acima é gerado no chare para realizar a serialização e desserialização dos dados do chare. Se a função pack estiver definida no chare, ela será chamada para a exportação dos dados de Lua. Os dados exportados são o tamanho da string de dados vinda de Lua e a própria string. Quando o chare é migrado e o método pup recebe os dados migrados, o chare recupera o tamanho da string e a string e se o método `unpack` estiver definido, ele é chamado recebendo a string como parâmetro.

Suponhamos um chare que receba uma string em um método e a processe através de uma função local, que pode ter sido definida dinamicamente, e guarda o resultado do seu processamento até que um outro chare consulte o resultado. A sua interface seria:

```
module Prog {
```

```
...
 chare [lua "map.lua"] Map {
    entry [lua] Map();
    entry [lua] void processString(int size, char str[size]);
    entry [lua] void sendResult();
  };
...
};
```

Após a migração, é importante que o chare mantenha o resultado do processamento das strings, e possivelmente que ele mantenha a implementação da função utilizada para o processamento, caso ela tenha sido definida dinamicamente, e não através do arquivo inicial map.lua. Para isso é necessário que os métodos pack e unpack sejam implementados. Abaixo apresentamos um exemplo de uma aplicação que exporta a implementação de uma função através da API de serialização de dados em Lua:

```
...
Map = {
  -- A função Map.processString chama a função processFunction
  processString = function(str)
    processFunction(str);
  end,
  sendResult = function()
    ...
  end,
  -- Função para serializar os dados
  pack = function()
    -- serializa a função
    local tfunc = debug.content(processFunction);
    -- Serializa os up values
    for key, value in pairs(tfunc.upvals) do
      tfunc.upvals[key]=debug.content(value)
    end
    -- Serializa dados de protótipo da função
    tfunc.p = debug.content(tfunc.p)
    -- função que tranforma a tabela tfunc em uma string
    return save("ffunc", tfunc)
  end,
  unpack = function(data)
    -- recupera os dados através da execução do conteúdo de data
```

```
    loadstring(data)()
    -- recupera os dados serializados
    for key,value in pairs(ffunc.upvals) do
      ffunc.upvals[key] = debug.apply(value,'upval')
    end
    ffunc.p = debug.apply(ffunc.p,'proto')
    processFunction = debug.apply(ffunc,'function')
  end
}
...
```

O código acima é um exemplo simplificado do uso da biblioteca de serialização de objetos em Lua para uma tabela. Essa biblioteca contém dois métodos principais: *content*, que transforma objetos Lua em tabelas (com protótipo e upvalues, no caso de existirem), e *apply*, que faz o contrário.

Após serem transformados em tabelas, os objetos são convertidos para strings pela função *save*, que é descrita no livro *Programming in Lua*. `save` é uma função que transforma objetos em strings, tomando cuidado de lidar com referências circulares de tabelas. Após a serialização da função no método `pack`, a função é migrada e, no método `unpack`, é recuperada.