6
Considerações Finais

O objetivo deste trabalho foi discutir a viabilidade do uso de linguagens de script para programação paralela através do desenvolvimento de uma integração entre duas linguagens, de modo a oferecer uma solução que permitisse a implementação de partes de uma aplicação paralela em uma linguagem de script, mas mantendo sua execução baseada em um sistema robusto de computação paralela. Através dos resultados de análises da integração construída, esperávamos demonstrar casos de uso de linguagens de script para programação paralela que fossem ágeis e flexíveis, com uma perda de eficiência aceitável.

Após a escolha da tecnologia que usamos para o desenvolvimento desta dissertação, estudamos as soluções já propostas para esta tecnologia e avaliamos seus pontos positivos e negativos, para depois desenvolvermos uma nova integração tentando solucionar os problemas encontrados anteriormente, agregando maior flexibilidade, mas mantendo os pontos positivos encontrados.

Através de testes e exemplos, podemos sugerir que o uso de linguagens de script está se tornando viável para programação paralela, e, apesar de estar em estágios iniciais, a combinação de linguagens de programação pode se tornar uma poderosa ferramenta para o desenvolvimento de aplicações paralelas em um futuro próximo.

6.1
Trabalhos correlatos

Antes de iniciarmos este trabalho, procuramos estudar as tecnologias já existentes, buscando uma tecnologia madura, robusta e extensível na qual fosse possível a integração com uma linguagem de script. Apesar de havermos encontrado poucos trabalhos correlatos, podemos citar alguns abaixo.

6.1.1
pyMPI

pyMPI (30) é um binding entre a linguagem Python e a interface de comunicação MPI. pyMPI dá suporte à maior parte da API de MPI. Sua grande vantagem é a intimidade que grande parte dos programadores paralelos têm

com o modelo e API de programação de MPI. Sua desvantagem é justamente o uso de MPI, que apesar de ser um modelo de comunicação poderoso, é pouco flexível e adaptável. MPI oferece pouca abstração em relação ao meio de execução e não possui ferramentas úteis para dinamismo de execução, como balanceamento de carga e um modelo de divisão de tarefas similar ao oferecido por arrays de Charm++.

6.1.2 Charisma

Charisma (33) é uma linguagem de alto-nível, baseada em Charm++, que visa facilitar a expressão do fluxo de controle de uma aplicação. Charisma separa fragmentos de código sequencial e construções paralelas, e possibilita o programador definir o fluxo global da aplicação através de uma linguagem de script. O script escrito é responsável por coordenar uma coleção de objetos migráveis.

Através de um parser do script, Charisma gera código Charm++, ou seja, um arquivo de interface e a implementação de objetos que são os chares. Esse código gerado, em conjunto com o código seqüencial fornecido, será compilado e gera a aplicação Charm++.

6.2 Melhorias para o binding

Durante a implementação do segundo binding, foram feitas algumas restrições de uso de funcionalidades do Charm++ em aplicações implementadas em Lua. Essas funcionalidades podem ser oferecidas para o usuário através de novas extensões da linguagem de interface e implementações no parser. Entre essas restrições, as principais são as seguintes:

- Reduções em arrays

  Uma redução executa uma operação (add, min, max, etc) em dados espalhados em diversos processadores e coleta o resultado em um único lugar. Charm++ suporta reduções nos elementos de um array. Para isso todos os elementos de um array devem contribuir com dados e executar o mesmo tipo de redução e o resultado é passado para uma callback informada pelo programador. Como a implementação de callback depende de código C++, uma nova extensão é necessária no binding para possibilitar a exportação e criação de callbacks através de Lua.

- Inicialização de variáveis readonly

Charm++ disponibiliza um tipo de variável readonly, que deve ser inicializada durante a execução do construtor de um mainchare e após a inicialização, seu valor é transmitido via broadcast para todos os chares presentes na aplicação. Como as variáveis readonly precisam ser inicializadas através de C++, novamente uma mudança no binding tem que ser feita para possibilitar que Lua atribua valores a essas variáveis.

– Passagem de mensagens como parâmetro de um entry method

Uma grande limitação que existe no binding atual é a impossibilidade de que mensagens sejam passados como parâmetros para entry methods. A solução para este problema não é complexa, sendo necessária a geração de código para exportação dos dados das mensagens.

– Extensão do modelo de programação para incluir chamadas síncronas

Uma extensão que poderia fazer uso do nosso binding, e ao mesmo tempo estender o modelo de programação, é o uso de co-rotinas em Lua para a implementação de uma API para chamadas remotas síncronas como no modelo RPC(32). Apesar de haver muitas críticas em relação ao modelo de RPC, em muitos casos é mais vantajoso durante o desenvolvimento de uma aplicação a chamada de um método remoto síncrono (31). Apesar de Charm++ já disponibilizar uma maneira de se fazer chamadas de métodos remotos assincronamente, ele impõe uma restrição de que as threads paradas na execução não podem ser migradas de processador, devido a dificuldades de lidar com a pilha de C e endereçamentos. Em Lua não temos essa restrição; através da biblioteca de serialização de dados descrita anteriormente, co-rotinas podem ser serializadas e recuperadas durante o processo de migração do chare, desde que sua pilha de chamadas não contenha funções escritas em C.

– Redefinição de funcionamento do balanceador de carga

Durante a execução de uma aplicação de grande duração, muitas vezes acontecem mudanças no ambiente de execução, em carga de processamento pouco previsíveis, entre outros. Esses problemas podem fazer com que o algoritmo de balanceamento de carga não seja mais adequado à nova realidade da aplicação.

O balanceamento de carga é uma área onde o uso de uma linguagem dinâmica pode ser bastante útil devido à sua necessidade de dinamismo e adaptabilidade. Através do uso de um balanceador de carga baseado em uma linguagem de script, será possível a redefinição do algoritmo em tempo de execução.

– Desenvolvimento de uma geração de código em duas etapas

A abordagem utilizada nesse trabalho foi de modificar o compilador de Charm++ para que este gerasse o código de integração com Lua. Uma outra opção seria implementar um gerador desacoplado do compilador para que fosse mais simples a troca de uma implementação em Lua e em C++ e para que a implementação ficasse desassociada da interface.