Desenvolvimento de Aplicações Distribuídas Utilizando DPC++

Eduardo Moschetta, Gerson Geraldo H. Cavalheiro

Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Ciências Exatas e Tecnológicas
Universidade do Vale do Rio dos Sinos
{eduardom,gersonc}@exatas.unisinos.br

Abstract

Nesse artigo, é apresentado DPC++, uma extensão da linguagem C++ para programação de aplicações de alto desempenho em aglomerados de computadores. Introduzindo uma quantidade mínima de primitivas à linguagem original, oferece recursos de alto nível ao programador que simplifica o desenvolvimento de aplicações distribuídas, deixando inalterada a lógica de programação seqüencial. Nesse trabalho, esses recursos da linguagem são apresentados, bem como o pré-processamento realizado em torno desses mecanismos. O modelo de objetos distribuídos empregado pela ferramenta é apresentado e validado com testes de desempenho, os quais também permitem analisar os custos gerados pelos recursos incluídos por DPC++ em uma aplicação distribuída.

1. Introdução

A crescente expansão de aglomerados de computadores e máquinas multiprocessadas como suporte de execução a aplicações concorrentes e distribuídas não tem tido o mesmo reflexo na evolução dos recursos para a programação das mesmas. O uso conjunto de várias ferramentas clássicas como MPI/*sockets* e *threads*, as quais foram desenvolvidas para a exploração de recursos específicos e distintos das arquiteturas, e a tarefa de mapear as atividades concorrentes da aplicação nos recursos de hardware da arquitetura têm dificultado o desenvolvimento de programas de alto desempenho por programadores ainda leigos ou iniciantes na área. Nesse cenário, ferramentas como ambientes de programação e extensões de linguagens têm surgido como uma alternativa às linguagens de programação tradicionais, oferecendo ao programador mecanismos de alto nível que exploram, de forma transparente, recursos de software e hardware da arquitetura. Exemplos do primeiro caso podem ser encontrados no Athapascan, Cilk e Hood. Entre as extensões de linguagem existentes atualmente, nota-se um domínio predominante daquelas que estendem as linguagens de orientação a objetos, principalmente a linguagem Java, tendo como exemplos o RMI (*Remote Method Invocation*), JavaSpaces e JavaParty.

DPC++, Processamento Distribuído em C++, é uma extensão à linguagem C++ para o processamento de alto desempenho em aglomerados de computadores. DPC++ é uma contraposição a maioria das ferramentas existentes, as quais estendem a linguagem Java: enquanto essas creditam a exploração de recursos heterogêneos através da portabilidade do Java, DPC++ assume que o hardware de suporte possua recursos homogêneos, característica comumente encontrada em aglomerados de computadores. DPC++ tem como objetivo aliar as facilidades da programação de orientação a objetos com os benefícios do processamento distribuído, oferecendo uma ferramenta que simplifique a programação de aplicações distribuídas. Como decisão de projeto, insere seu foco na obtenção de alto desempenho, melhor alcançada com uma linguagem compilada, ao invés da alta portabilidade garantida pela linguagem Java.

Descrita em [1] [2], DPC++ herda de C++ o suporte à orientação a objetos, introduzindo a esta uma quantidade mínima de primitivas, as quais oferecem recursos para o programador identificar atividades concorrentes de sua aplicação. Em DPC++, uma aplicação é um conjunto de objetos distribuídos, dotados de fluxos de execução independentes, permitindo assim que instâncias desses executem concorrentemente. Nesse trabalho, é apresentada a evolução dessa linguagem, caracterizada pela eliminação do Diretório, *daemon* no modelo original responsável por centralizar a ativação de objetos remotos, e pelo uso de *threads* ao invés de processos como suporte de execução para um objeto distribuído em uma máquina remota. Tais propriedades refletem em custos de comunicação menores

e uma exploração de recursos remotos mais eficiente, respectivamente.

O restante do artigo segue organizado como segue: a Seção 2 introduz o problema em que se insere o DPC++, descrevendo a estratégia utilizada para aliar um paradigma seqüencial a um paradigma concorrente; em seguida, alguns exemplos de ferramentas para a programação de aplicações distribuídas são apresentados e comparados com DPC++; a Seção 3 apresenta o modelo de objetos considerado pela ferramenta; a sintaxe e semântica da linguagem são apresentadas na Seção 4, seguida da análise de desempenho realizada com uma aplicação distribuída escrita em DPC++ (Seção 5); por fim, a Seção 6 relata algumas conclusões retiradas até o momento e perspectivas futuras desse trabalho.

2. Orientação a objetos x Processamento Distribuído

Atualmente, percebe-se que o paradigma mais semelhante ao modelo distribuído de processos é o de orientação a objetos, devido à sua estrutura fortemente modular e seu método de comunicação análogo aos sistemas distribuídos [6].

Há várias semelhanças entre uma aplicação seqüêncial composta por múltiplos objetos e um sistema distribuído de múltiplos processos. Em primeiro lugar, os objetos e processos podem ser vistos como unidades de processamento no contexto de seus respectivos paradigmas. Objetos possuem **estado interno** e uma **interface de acesso** constituída pelos métodos enquanto que processos distribuídos possuem **memória interna** e uma **interface de serviços remotos**, definida por seus pontos de acesso. Além disso, ambas abordagens permitem a cooperação entre as unidades através da troca de mensagens, implementadas sob semânticas diferentes (leitura/escrita na memória para objetos, e mensagens pela rede através de bibliotecas como MPI e *sockets* para processos distribuídos).

Nesse cenário, DPC++ permite ao programador implementar sua aplicação distribuída utilizando paradigma e linguagem de programação de orientação a objetos (C++), suportando a execução dessa aplicação em um modelo de processos distribuídos. Mais especificamente, cada objeto instanciado é representado por um processo localizado em uma máquina do aglomerado, sendo que cada mensagem de ativação de método consiste no envio de uma mensagem física à máquina que hospeda esse processo. O método ativado é executado remotamente, sendo a passagem de parâmetros e resultados entre a aplicação e o objeto realizado de forma transparente ao usuário. Esse nível de abstração é alcançado através do pré-processamento de primitivas de alto nível introduzidas pelo DPC++, a serem apresentadas na Seção 4.

2.1. Java RMI

Java RMI (*Remote Method Invocation*) é uma ferramenta para programação de sistemas distribuídos na linguagem Java, disponibilizando recursos ao programador para a criação, registro e uso de objetos remotos. Após um objeto remoto ter sido registrado no *daemon* RMIRegistry de alguma máquina por um processo servidor, um processo cliente pode obter uma referência remota a esse objeto através de uma descrição URL do mesmo (ex.: `//nomemaquina/nomeobjeto`) e invocar seus métodos utilizando a mesma sintaxe da invocação local. Entretanto, a semântica de passagem de parâmetros torna-se diferente: objetos locais são passados por cópia (devendo ser serializáveis), enquanto objetos remotos são passados como referência remota.

O uso de objetos remotos não é transparente, devendo ser explicitado através de uma classe que implemente uma interface `Remote`. Na programação de sistemas distribuídos, o processo servidor deve instanciar um objeto dessa classe e registrá-lo sob algum endereço, que deve ser conhecido pelo processo cliente, o qual então pode obter o objeto remoto e invocar seus métodos normalmente. Para a execução desses processos, deve-se ainda gerar os *stubs* e *skeletons*, a partir do comando `rmic`, responsáveis pelo empacotamento dos parâmetros, e inicializar o *daemon* RMIRegistry.

2.2. JavaSpaces

O principal objetivo de JavaSpaces [3] é prover um serviço de computação cooperativa, onde um conjunto de processos cooperam via fluxo de objetos em um ou mais espaços. Com vistas a suportar conceitos como transações, JavaSpaces permite o uso de múltiplos espaços simultaneamente. Um espaço funciona como um repositório de objetos, mas ao contrário do RMI, não é dependente da localidade. Com bancos de dados orientados a objetos, a ferramenta viabiliza a persistência dos objetos inseridos nos espaços. O modelo de objetos é manipulado através de 4 operações simples – `write` para escrever um objeto no espaço, `read` para ler um objeto do espaço, `take` para retirar um objeto do espaço e `notify` para notificar a escrita de uma entrada de *matching* – refletindo na simplicidade da ferramenta e em uma ótima escalabilidade.

Uma metodologia comum para desenvolver sistemas distribuídos com JavaSpaces é modelar os mesmos através da permuta de objetos. O cliente escreve um objeto no espaço que requisite algum serviço; o servidor retira o objeto requisitante e executa a ação associada, escrevendo no mesmo espaço um objeto contendo o resultado; por fim, o cliente retira do espaço este último. A invocação aos métodos do objeto remoto é sintaticamente igual à invocação local e semanticamente semelhante à invocaçao remota do RMI. A

única diferença está na serialização dos objetos locais: um objeto é somente serializado antes de ser escrito no espaço e desserializado após ser lido ou retirado do mesmo.

2.3. JavaParty

JavaParty [4] viabiliza o suporte a programas *multithreaded* em ambientes distribuídos como aglomerados de computadores, estendendo as capacidades do Java para a computação distribuída. Com a introdução de uma única primitiva (`remote`), JavaParty permite declarar classes remotas que, juntamente com suas instâncias, possam ser visíveis e acessíveis em qualquer lugar do ambiente distribuído de JavaParty. A localidade dos objetos remotos é decidida, de forma transparente, por meio de um distribuidor dinâmico (classe pré-definida que deve ser instanciada pela aplicação). JavaParty ainda permite a migração explícita de objetos remotos, na qual deve ser informada a nova localidade dos mesmos.

Tanto a criação como a invocação de métodos de objetos remotos é sintaticamente equivalente à linguagem original, sendo portanto transparente o uso de objetos remotos. A semântica da passagem de parâmetros é igual a do RMI. Ao contrário de RMI, porém, nenhuma exceção é implementada, visto que JavaParty é destinada à computação em aglomerados de computadores, onde a ocorrência de falhas é rara e passível de ser tratada por outro sistema.

Javaparty ainda suporta *threads* distribuídas transparentes, com seu manejo em ambientes distribuídos semanticamente igual às *threads* Java, considerando sincronização e controle de execução. Nesse contexto, a execução de um método remoto é mapeada em duas diferentes *threads* Java: a *thread* cliente que inicia a chamada e espera pelo retorno do método, e a *thread* servidora que executa o método no nó remoto e retorna os resultados.

2.4. Comparativo entre as ferramentas

A Tabela 1 mostra um comparativo entre os principais aspectos encontrados para cada ferramenta apresentada. Alguns desses aspectos são considerados a seguir, no modelo de objetos de DPC++, enquanto outros foram relatados a título de curiosidade. É importante observar que o JavaParty é a ferramenta que mais se aproxima de DPC++, principalmente por ter sido concebida para o mesmo fim – implementação de aplicações paralelas e distribuídas em aglomerados de computadores.

3. Modelo de Distribuição de Objetos

Um sistema distribuído em DPC++ consiste em um conjunto de objetos distribuídos instanciados pela aplicação do usuário. Um objeto distribuído nada mais é do que o mapeamento entre um **objeto procurador** e um **objeto remoto**. O primeiro, mais especificamente, é o *stub*, gerado pelo pré-processador de DPC++, que fica responsável pelo empacotamento e transferência de parâmetros entre a aplicação e o objeto remoto. O objeto remoto, por sua vez, é um fluxo de execução, em algum nó computacional, que disponibilize os serviços do objeto distribuído através dos recursos providos pelo DPC++. Esses dois tipos de objetos permitem a manipulação transparente de objetos distribuídos pelo usuário, que enxerga o objeto procurador como um objeto local.

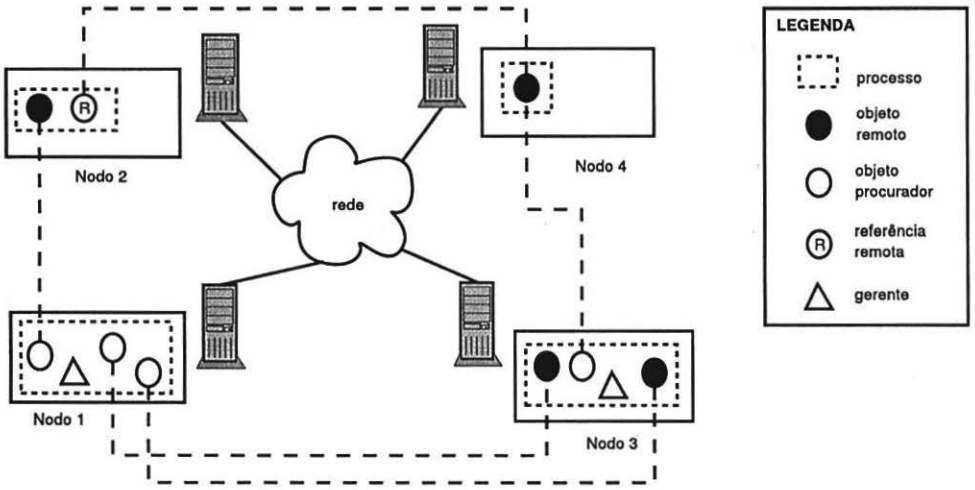
A Figura 1 apresenta o modelo de objetos distribuídos evoluído de DPC++. O antigo modelo pode ser conferido em [1] [2]. Enquanto que este fazia uso de processos independentes como forma de suporte para execução de um objeto distribuído, o novo modelo utiliza apenas um processo *multithreaded*, onde cada *thread* representa um objeto remoto. Entretanto, um processo, em um nó remoto, agrupa todos os objetos remotos instanciados de uma mesma classe que estão residentes no mesmo nó. Desse modo, caso existissem objetos remotos instanciados de classes distintas no nó 3 da Figura 1, dois processos seriam necessários. O uso de *threads* viabiliza o compartilhamento de recursos de DPC++ e da própria aplicação por objetos remotos localizados logicamente em um mesmo nó, além de gerar um menor *overhead* de escalonamento e uso de memória no sistema operacional daquele nó, que agora lida com vários processos leves ao invés de processos aplicativos.

Em termos gerais, cada objeto remoto possui:

- **Estado Interno**: armazenado na memória interna do objeto remoto, é visível apenas através dos métodos desse. Corresponde ao estado do objeto sob o ponto de vista de seus dados internos.
- **Métodos**: conjunto de procedimentos que executam, de forma seqüencial, as funções do objeto.
- **Canal de entrada**: canal de comunicação com o objeto procurador, recebe mensagens de invocação de métodos e envia respostas.
- **Delegação**: gerencia o recebimento, pelo canal de entrada, das mensagens, ativando os métodos solicitados e repassando os respectivos parâmetros. Responsável também pela entrega de respostas de confirmação de recepção da mensagem, retorno de funções e parâmetros de resultados ao objeto procurador.

Com a eliminação de um *middleware*, responsável pela ativação dos objetos distribuídos, o mecanismo de controle passou a ser exercido pela própria aplicação, a qual qual é vinculada, em tempo de pré-processamento, a um código aplicativo destinado a tal função. Este código aplicativo cria um **gerente** na máquina cliente (máquina em que

Aspectos/Ferramentas	Java RMI	JavaSpaces	JavaParty	DPC++
Uso de objetos remotos	Explícito	Explícito	Transparente	Transparente
Programação de	Cliente e Servidor	Cliente e Servidor	Cliente	Cliente
Localidade dos objetos	RMIRegistry do nodo	Espaço de objetos	Ambiente JavaParty	Aplicação cliente
Transparência de localidade	Não	Apenas dentro do espaço	Sim	Sim
Utiliza *middleware*	Sim	Sim	Sim	Não
Passagem de parâmetros	Obj. locais e remotos	Obj. locais e remotos	Obj. locais e remotos	Dados serial. e obj. remotos
Threads distribuídas transparentes	Não	Não	Sim	Sim
Persistência dos objetos	Não	Sim	Não	Não
Principal aplicação	Cliente/Servidor	Computação cooperativa	Aplicações paralelas em aglomerados	Aplicações paralelas e distribuídas em aglomerados



está sendo executado), encarregado pelo gerenciamento de todos os objetos distribuídos instanciados pela aplicação. Além disso, objetos remotos que instanciem outros objetos distribuídos também têm seus próprios gerentes, formando uma rede hierárquica de gerência, cuja raiz é o gerente da aplicação. É importante observar que o gerente não é um *daemon*, mas um objeto local, instanciado localmente sempre que o primeiro objeto distribuído é criado. O gerente irá atuar sobre o conjunto de nodos envolvidos na execução da aplicação, que formam a **arquitetura virtual** de DPC++ durante esse interím.

O modelo de distribuição de objetos de DPC++ também oferece mecanismos de suporte à referência de objetos distribuídos – denotada como **referência remota**. Em DPC++, uma referência remota nada mais é do que a criação de um novo objeto procurador que se comunique, por um canal alternativo, com o mesmo objeto remoto. A localização do objeto remoto é encontrada na rede hierárquica de gerentes, que intermedia a criação do novo canal de comunicação.

3.1. Comunicação e sincronização entre objetos

Em DPC++, o único meio de comunicação e sincronização entre os objetos distribuídos, e entre esses e a aplicação, é através de mensagens de invocação de métodos e respostas. Essa interação pode ser realizada de forma síncrona ou assíncrona, através dos três tipos de mensagens permitidos:

- **Mensagem síncrona**: objeto solicitante espera até que o método seja executado e os resultados da função sejam recuperados pelo mesmo.
- **Mensagem assíncrona**: objeto solicitante invoca método remoto e prossegue com a execução.
- **Mensagem assíncrona com confirmação**: objeto requisitante invoca método e espera uma resposta de recebimento da mensagem, voltando em seguida à execução.

A sincronização entre mensagens de objetos instanciados e de suas referências remotas, ou entre duas chamadas assíncronas adjacentes, é realizada automaticamente pela ferramenta. Mais especificamente, mecanismos de DPC++ garantem que nenhuma mensagem de requisição ative um método do objeto remoto enquanto outro método do mesmo objeto estiver em execução, independente da forma em que o último foi ativado (assíncrona ou não), permanecendo assim inalterada a lógica de programação seqüencial.

3.2. Algoritmo de distribuição de objetos

DPC++ ainda está em uma fase de validação, possuindo um algoritmo de escalonamento simples, mas capaz de viabilizar o modelo de distribuição e comunicação entre objetos remotos. Apesar de existir uma rede hierárquica de gerentes durante a execução distribuída da aplicação, o gerenciamento continua sendo efetivado de forma centralizada, a

exemplo da versão original de DPC++. Nesse algoritmo de gerenciamento, o gerente localizado na raiz da árvore retém informações referentes à carga global da aplicação, tais como o número total de objetos distribuídos instanciados, números de referências remotas, entre outros dados. Tais informações permitem um balanceamento de carga equilibrado em um nível de granularidade mais grossa. Todavia, esse balanceamento é realizado apenas na criação de objetos distribuídos, ficando obsoleto nos demais casos, visto a dificuldade de realizar a migração de código e dados em linguagens como C++, nas quais a serialização não pode ser garantida por um custo viável.

No esquema de controle citado, os demais gerentes assumem uma função meramente informativa, auxiliando na constituição da carga global da aplicação no gerente principal. Além disso, são responsáveis pela criação das referências remotas a objetos dos quais são responsáveis.

A heurística para se decidir qual a localidade do novo objeto remoto leva em conta fatores como número de objetos remotos, número de métodos ativos em cada máquina, a quantidade de referências remotas, o tempo médio de vida e custo médio de comunicação de objetos instanciados da mesma classe, ambos calculados em instâncias anteriores da classe distribuída, entre outros.

3.3. *Threads* distribuídas transparentes

A exemplo de JavaParty, DPC++ disponibilizará suporte a *threads* distribuídas, que operem de forma transparente ao usuário. Esse recurso expande o uso de *multithreading* para dentro da parte cliente da aplicação, permitindo obter-se as mesmas vantagens visívies nas máquinas remotas, cujos processos utilizam *threads* para cada instância de objeto distribuído. Além dos custos de comunicação, a espera por métodos síncronos pode ser sobreposta por processamento realizado em outros objetos procuradores. Um certo grau de paralelismo pode ser alcançado, caso a máquina cliente seja multiprocessada. Esses fatores visam melhorar o desempenho da aplicação também na parte cliente, aumentando sua disponibilidade para com as respostas dos objetos remotos e sobrepondo custos de comunicação e sincronismo por cálculo efetivo.

Na implementação a ser realizada, as *threads* distribuídas serão mapeadas em duas *threads* lógicas: uma *thread* cliente, que execute o objeto procurador, e uma *thread* servidora, que resida no processo servidor. A comunicação entre essas duas *threads* obedece aos mesmos mecanismos adotados para objetos procuradores e objetos remotos. A ativação de um método síncrono bloqueia apenas a *thread* responsável pelo objeto procurador, sendo que o fluxo de execução principal continua a execução, até que todas *threads* estejam bloqueadas em seus métodos síncronos.

4. Linguagem

O nível de abstração da linguagem descrito até o momento é alcançado através de recursos de alto nível providos pelo DPC++. O usuário declara uma classe remota com a primitiva `dclass`, em substituição à palavra reservada `class` do C++. A interface de métodos da classe é definida conforme às seguintes regras:

- métodos declarados como `private` são de uso local ao objeto remoto (apenas acessíveis no processo servidor);
- métodos públicos (`public`) são considerados remotos, sendo acessados no lado cliente e executados pelo processo servidor;
- métodos estáticos são de uso privado ou público, porém não conseguem acessar o estado interno do objeto remoto;
- métodos com dados de retorno (tanto retorno de método como parâmetro de resultado) são considerados síncronos;
- métodos com retorno `void` são considerados assíncronos;
- métodos precedidos pela primitiva `confirmation` também são considerados assíncronos, porém sua execução procede somente após receber confirmação de envio dos parâmetros.

A passagem de parâmetros permite argumentos do tipo primitivo (`char`, `int`, `float` etc.), tipos estruturados, vetores de uma ou duas dimensões destes tipos suportados e referências a objetos remotos. As estruturas, ao serem declaradas, necessitam ser precedidas da primitiva `dstruct`, com vistas a permitir o pré-processador verificar se a estrutura pode ser serializada. Uma estrutura é serializável se possuir somente membros do tipo primitivo e/ou outras estruturas já validadas, excluindo portanto ponteiros e referências. Os vetores podem ser estáticos, caso o tamanho dos mesmos seja conhecido em tempo de compilação, ou dinâmicos, caso contrário. Três classes pré-definidas, `dpCVector`, `dpCMatrix` e `dpcString` são utilizadas para implementar vetores dinâmicos, que possam ser serializados e passados como parâmetros em métodos remotos, possuindo os mesmos recursos de seus equivalentes na linguagem C++ (uso de [ ] para indexação, por exemplo).

O método de passagem de parâmetros pode ser realizado por valor (cópia da entrada), por resultado (cópia da saída) ou por valor-resultado (cópia da entrada e saída) [5], sendo declarado através das palavras-chaves `in`, `out` e `inout`, respectivamente. O método *default* é `in`, sendo portanto opcional a explicitação do método de passagem. Caso seja

necessária, pode ser feita através do uso da palavra-chave antes do tipo do argumento (ex.: `out int demo;`).

Para a programação de *threads* distribuídas, a classe remota deve estender a classe abstrata `dpCThread` e implementar os métodos `create_thread` e `join_thread`, utilizados para criar e esperar pelo término da *thread*, respectivamente. Esse nível de sincronização proporcionado pelo método `join_thread` é apenas utilizado para conciliar as datas de execução das *threads* em relação à produção e consumo de dados. Assim como em outras ferramentas, o usuário deve explicitamente utilizar outros tipos de primitivas de sincronização para proteger as sessões críticas de seu código, como *mutex* e variáveis de condição, por exemplo.

4.1. Pré-processamento e geração de código

O hábito de os programadores desenvolverem suas classes em dois arquivos separados, um com a declaração da classe e outro com a implementação da mesma, influenciou fortemente a construção do pré-processador do DPC++. Nesse cenário, uma classe remota é declarada em um arquivo (estensão `dpC`), enquanto sua implementação é escrita em outro. Entretanto, somente o primeiro arquivo é pré-processado pelo DPC++; o segundo arquivo é escrito com a linguagem pura de C++. Nesse último, a definição dos métodos é idêntica à declarada na interface da classe, retirando-se apenas as palavras-chaves que definem o método de passagem dos argumentos e a primitiva `confirmation`.

O pré-processador recebe como parâmetros na linha de comando o nome dos dois arquivos citados e mais o nome do arquivo que contém a composição da arquitetura virtual de DPC++, onde cada linha do arquivo é o nome/IP de um *host*. A vinculação dessa arquitetura com o programa distribuído se dá, em tempo de pré-processamento, apenas pelo nome do arquivo e não pelo seu conteúdo. Mais especificamente, toda vez que aplicação inicia, o arquivo é lido para então o DPC++ formar a arquitetura virtual. Por fim, o código DPC++ é pré-processado gerando, dois arquivos, ambos na linguagem C++:

- *stub* cliente, contendo a definição da classe do objeto procurador e outras funções que vinculam a aplicação ao código aplicativo de DPC++, responsável pelo gerenciamento dos objetos remotos.
- programa servidor, contendo a definição da classe do objeto remoto e outros mecanismos necessários ao funcionamento da arquitetura virtual.

Ao escrever uma aplicação distribuída, o programador necessita apenas incluir o *stub* cliente na sua aplicação (`#include`), compilando-a com *linkage* para `-ldpc++`. Ao contrário da maioria das ferramentas, em DPC++ não é necessário desenvolver o programa servidor, pois este é automaticamente gerado pelo pré-processador, de uma forma otimizada para seu propósito. O gerenciamento é embarcado no programa cliente em tempo de pré-processamento, retirando o *overhead* que era gerado por uma camada de *middleware* e facilitando a execução dessas aplicações.

É importante observar que DPC++ garante um bom grau de portabilidade no código gerado para arquiteturas paralelas, como aglomerados de computadores e máquinas multiprocessadas, utilizando para sua confecção primitivas de ferramentas padrões como *threads* POSIX e *sockets* do Unix, e utilizando a ferramenta `ssh` como mecanismo para a execução de processos em máquinas remotas.

4.2. Exemplo de Aplicação em DPC++

No Algoritmo 1 encontra-se a declaração da classe remota `ImageOp`. Além de nos permitir validar a simplicidade da linguagem, essa classe também nos viabilizou uma detalhada avaliação de desempenho, documentada na Seção 5. O objetivo dessa classe é prover como principal funcionalidade a convolução de uma imagem através de uma matriz de convolução 3×3. Como métodos auxiliares, a classe possui o `openImage` e `saveImage`, para abrir uma imagem em `img` e salvar uma imagem de `img_res`, respectivamente. Os objetos locais `img` e `img_res` representam a imagem antes e após a convolução, respectivamente.

Algoritmo 1 Definição da classe remota – ImageOp.dpC

```
dstruct mask_t { int map[3][3]; };
dclass ImageOp {
private:
  QImage *img;
  QImage *img_res;
  int status;
public:
  ImageOp();
  ~ImageOp();
  void openImage_assync(dpCString filename);
  int openImage(dpCString filename);
  void apply_convolution(mask_t mask);
  int saveImage(dpcString filename);
  void saveImage_assync(dpCString filename);
}
```

Com vistas a avaliar os efeitos da utilização de sincronismo e não-sincronismo em uma aplicação distribuída, tanto na forma de programar como no desempenho, foram criadas duas versões da aplicação, uma realizando chamadas a métodos síncronos e outra a métodos assíncronos. Isso explica a existência de dois métodos para abrir e salvar uma imagem: um método síncrono, que re-

torna o sucesso da operação, e outro assíncrono. No segundo caso, a forma de programação é um pouco diferente: cada método pode opcionalmente verificar se o método anteriormente chamado executou com sucesso (verificando o valor de `status`). Os mecanismos de DPC++ garantem que nenhum método remoto seja ativado enquanto outro estiver em execução, seja o primeiro síncrono ou assíncrono.

5. Avaliação de Desempenho

A avaliação de desempenho foi dividida em duas etapas. A primeira tem como principal objetivo validar o modelo de distribuição de objetos de DPC++. Já a segunda fase concentra-se na validação dos mecanismos de comunicação e gerenciamento providos pelo mesmo.

A classe remota `ImageOp`, apresentada na Seção 4.2, foi utilizada para o desenvolvimento de uma aplicação distribuída, cuja principal função seria proporcionar esta avaliação de desempenho, em ambas etapas. Nesse contexto, a aplicação comportava-se da seguinte maneira: uma instância de `ImageOp` era criada, seguida de uma chamada para seus métodos de abrir, aplicar a convolução e salvar uma imagem, finalizando com a destruição da instância; para várias imagens, essa seqüência de operações era feita em lote por operação, ou seja, primeiro eram criadas todas as instâncias, depois cada instância tinha seu método `openImage` ativado etc.

A arquitetura física utilizada nos experimentos consiste de máquinas PIV 1.8GHz (256 Mb RAM, Fast Ethernet), ligadas por um *switch* também Fast Ethernet.

5.1. Ganho de desempenho com DPC++

A Tabela 2 retrata os tempos de execução obtidos com as duas versões da aplicação, tendo cada tempo acompanhado do respectivo ganho de desempenho obtido com a execução distribuída. A linha *Seqüencial*, que mostra o tempo de execução da versão seqüencial da mesma aplicação, está presente apenas para fins de comparação. As demais linhas são utilizadas para diferenciar as execuções sobre diferentes configurações da arquitetura virtual de DPC++. A carga de trabalho em todas as execuções consistiu de 30 imagens, armazenadas em um servidor de arquivos, sendo portanto equivalentes os custos para acessá-las e salvá-las em qualquer nó da arquitetura, assim como no nó utilizado para a execução seqüencial.

Os tempos desta tabela nos permitem, antes de tudo, verificar os bons ganhos de desempenho obtidos com as execuções distribuídas. Porém, estes ficaram em uma escala menor na versão síncrona, devido ao fato dessa se aproximar à execução seqüencial da aplicação. O pequeno ganho obtido nessa versão deve-se à existência de um

	Versão Síncrona		Versão assíncrona	
	Tempo	Ganho	Tempo	Ganho
Seqüencial	21,21 s			
Dist. – 2 nodos	16,43 s	1,29	10,60 s	1,99
Dist. – 3 nodos	15,16 s	1,40	7,54 s	2,81
Dist. – 4 nodos	14,84 s	1,43	5,92 s	3,58
Dist. – 5 nodos	14,55 s	1,46	4,83 s	4,47

Tabela 2. Tempos de execução e ganhos de desempenho da aplicação

única operação assíncrona, a qual é a convolução da imagem. Já na versão assíncrona, os ganhos obtidos, em geral, foram próximos ao número de nodos utilizados para a execução, o que nos leva a crer que a exploração dos recursos disponíveis é realizada de forma eficiente por DPC++, mesmo estando essa eficiência vinculada à forma de como é implementada a aplicação. A diferença de desempenho entre diferentes implementações tende a ser reduzida com a introdução de *threads* distribuídas, que permite a sobreposição da espera de sincronização de métodos por cálculo efetivo.

5.2. Custos gerados por DPC++

Os custos de comunicação e gerenciamento produzidos pelos mecanismos transparentes de DPC++ são visualizados na Tabela 3. Estes valores são relativos à execução da versão distribuída assíncrona da aplicação desenvolvida, utilizando-se 5 nodos. A primeira coluna descreve o custo qualitativamente, seguida do seu tempo (em microsegundos) em uma operação simples e do tempo total (em segundos) decorrido das 30 operações ativadas (uma para cada imagem).

Chamada de método	Custo individual	Custo total
Criação do objeto	28.161 ms	0,845 s
openImage_assync()	156 ms	0,005 s
apply_convolution()	36.702 ms	1,101 s
saveImage_assync()	56.394 ms	1,692 s
Destruição do objeto	34.896 ms	1,047 s
Custos gerados pelo DPC++		4,69 s

Tabela 3. Custos gerados pelo DPC++

Os tempos visualizados nessa tabela são referentes ao tempo decorrido desde o momento em que o método foi ativado localmente, até o instante em que o mesmo retorna à execução na aplicação cliente. Portanto, espera-se que métodos síncronos produzam tempos de resposta

maiores, proporcionais ao custo computacional da função do método. As operações de criação e destruição de objetos remotos são síncronas por natureza, o que reflete nos altos tempos observados na tabela, em relação a um método assíncrono como o `openImage_assync`. Nota-se que as chamadas aos métodos assíncronos (`apply_convolution` e `saveImage_assync`) geraram um custo maior do que a operação de criação. A explicação plausível para esse fato é que estes apresentaram uma latência maior de ativação por já existirem outros métodos em execução no momento.

A Tabela 3 também nos retrata o custo total gerado pelos mecanismos inseridos pelo DPC++ na aplicação. Dividindo esse tempo, de $4,69\ s$, pelo número de nodos utilizados na execução (5 nodos), obtemos uma margem de *overhead* teórica na faixa de $0,94\ s$, representando $19,46\%$ do total da execução distribuída. Essa margem tende a diminuir à medida em que se aumente a carga computacional das operações realizadas. Nessa aplicação, nota-se que os custos computacionais não são elevados, visto os baixos tempos de execução encontrados. Como nenhuma comparação foi feita com outra ferramenta do gênero, não sabe-se o quanto representativa é essa faixa de *overhead* gerada. Porém, a expectativa é que seja menor do que em outras ferramentas, pois não utiliza nenhum *middleware* para comunicação e gerenciamento, mecanismos estes compilados juntamente com a aplicação.

6. Conclusões

A implementação de sistemas distribuídos utilizando o ambiente DPC++ torna-se atrativa pelo fato de ser baseada em uma linguagem bastante popular, a linguagem C++. O usuário adaptado a linguagem C++ facilmente o será com DPC++, o qual introduz poucas modificações a serem conhecidas. O uso de objetos remotos é transparente ao usuário, viabilizando ao mesmo implementar aplicações distribuídas pensando na lógica da programação seqüencial. DPC++ provém recursos que explorem a arquitetura almejada pelo usuário, de forma eficiente e transparente, através de códigos aplicativos vinculados à aplicação e *stubs* gerados pelo pré-processador da ferramenta.

A avaliação de desempenho obtida foi considerada satisfatória, por permitir validar o uso da ferramenta com propósito de obter bom desempenho sob uma margem de *overhead* teoricamente pequena. Entretanto, em alguns aspectos essa avaliação precisa ser melhorada, tais como a comparação com outras ferramentas para programação paralela e distribuída e a análise do desempenho de execuções utilizando *threads* distribuídas. Isso nos leva a trabalhos futuros, em que também podem ser abordados algoritmos de gerenciamento distribuído, cujas implementações seriam viáveis por meio da rede hierárquica de gerentes provida pelo DPC++.

Referências

[1] G. G. H. Cavalheiro and P. O. A. Navaux. Dpc++: Uma linguagem para processamento distribuído. In *V Simpósio Brasileiro de Arquitetura de Computadores – Processamento de Alto Desempenho*, volume 2, pages 732–744, Florianópolis, SC, 1993. SBC.

[2] G. G. H. Cavalheiro and P. O. A. Navaux. Um modelo distribuído para linguagens orientadas a objetos. In *XX Seminário Integrado de Software e Hardware*, volume 2, pages 518–532, Florianópolis, SC, 1993. SBC.

[3] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns and Practice*. Addison-Wesley, Reading, 1999.

[4] B. Haumacher, T. Moschny, and M. Philippsen. Javaparty, 2003. http://www.ipd.uka.de/JavaParty/ (visitado 08/06/2004).

[5] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, Reading, 1999.

[6] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, New Jersey, 2000.