6
Avaliação

A avaliação deste trabalho foi baseada em dois pontos principais:

– **Desempenho:** análise da sobrecarga e escalabilidade dos algoritmos de redistribuição de dados e da aceleração e das aplicações que exemplificam o uso dos conectores paralelos.

– **Requisitos/desafios de implementação:** identificação dos desafios encontrados na integração das ICs em um middleware orientado a componentes e na implementação C++ das mesmas.

6.1
Avaliação de desempenho

Os testes para a avaliação de desempenho foram realizados no *cluster* do laboratório de Tecnologia em Computação Gráfica (TecGraf), onde estavam disponíveis 14 nós homogêneos interligados por rede Gigabit Ethernet com processador Intel Pentium D (núcleo duplo) 3.40 gigahertz (rodando a 2.4) e 2 gigabytes de memória RAM.

6.1.1
Sobrecarga da implementação

A análise da sobrecarga gerada pelo código de redistribuição de dados foi realizada em cada política de redistribuição que foi implementada para cada conector paralelo. O principal objetivo desta análise foi verificar a escalabilidade da implementação, ou seja, o tempo de redistribuição dos dados quando o número de componentes conectados cresce.

Os testes foram realizados executando-se 100 vezes cada demo implementado e registrando-se o tempo em microssegundos dos algoritmos de redistribuição de dados. Isso foi feito para até 56 componentes conectados. Vale ressaltar que não foi possível a análise para um número maior de componentes conectados, pois o *cluster* onde os testes foram realizados disponibiliza 14 *hosts* de núcleo duplo. Para 56 componentes foram necessários 4 processos por *host*, o que implicou em mais de uma thread de execução por núcleo, o que poderá ter alguma influência nos resultados obtidos para essa configuração.

MulticastReceptacle

A análise de sobrecarga das políticas de redistribuição *broadcast* e *reduce* foi realizada através do demo `PI_Dartboard`, onde foram contabilizados os tempos (em microssegundos) de duração do processo de *broadcast* dos parâmetros para os workers e de *reduce* dos resultados retornados pelos mesmos.

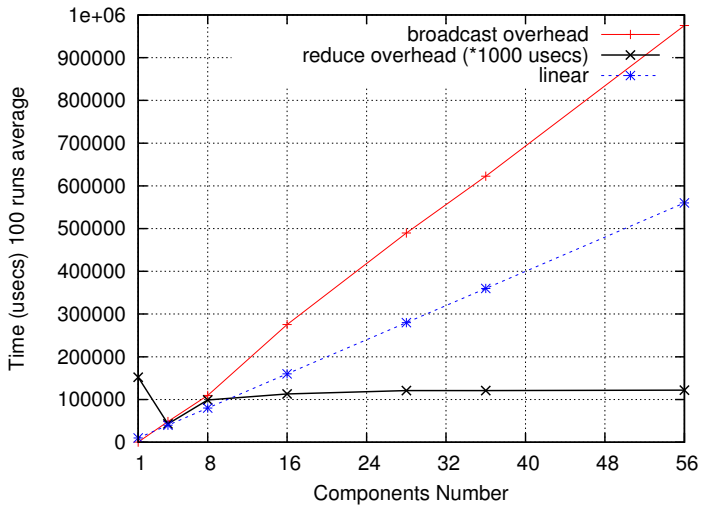

Figura 6.1: Sobrecarga imposta pelas implementações de *broadcast* e *reduce*

Como se pode observar no gráfico da Figura 6.1, o comportamento da linha referente à política *broadcast* manteve-se proporcionalmente linear com o aumento do número de componentes conectados, o que reflete uma boa escalabilidade de implementação.

Esse comportamento pode ser explicado pela necessidade de copiar o vetor de entrada para todas as x requisições para as facetas conectadas, além do tempo necessário para a criação dessas requisições, tornando o processo como um todo linear.

A rotina de *broadcast* do MPI apresenta comportamento logarítmico, pois, segundo Luecke et al. [37], o MPI utiliza uma abordagem em árvore.

A escalabilidade da política de redistribuição *reduce* se mostrou constante por que, a medida que são coletados os resultados, a operação de redução já é aplicada.

A análise de sobrecarga das políticas de redistribuição *scatter* e *gather* foi realizada através do exemplo criado no início da implementação dos conectores, o `PCGSolver`.

O gráfico da Figura 6.2 mostra a escalabilidade apresentada pelas políticas *scatter* e *gather*. Pode-se observar que a política *scatter* também apresenta um comportamento linear, dado que é necessária a divisão do vetor para

cada faceta conectada e a criação/configuração das requisições para cada conexão.

Segundo Luecke et al. [37], a operação `mpi_scatter` possui comportamento logarítmico, que, a partir de um determinado número de nós, fica muito próximo do constante. Luecke assume que a as mensagens são entregues em um esquema de árvore, o que explica o comportamento logarítmico da da operação `mpi_scatter`.

A escalabilidade da implementação da política de distribuição *gather* se mostrou próxima do constante. Isso pode ser explicado pelo fato de que uma vez que os resultados das invocações dos *workers* são retornados (nesse caso um valor do tipo *double*), é instanciada uma sequência a partir dos mesmos e a instanciação é uma operação considerada constante (como se pode ser confirmado com gráfico da Figura 6.2). Se cada *worker* retornasse um vetor de valores, o esforço para realizar o gathering desses vetores seria tipicamente linear, pois existiria a necessidade de cópia dos vetores retornados.

A análise do desempenho da rotina `mpi_gather` é semelhante a da rotina `mpi_scatter`, possuindo um comportamento quase constante [37].

GatherFacet

A análise da sobrecarga do conector *GatherFacet* foi realizada para 16, 32, 64 e 128 componentes clientes. O exemplo utilizado foi o `FindMAXValue`, descrito no Capítulo 5 que faz uso das políticas de redistribuição *gather* e *broadcast*.

O gráfico da Figura 6.3 mostra uma ótima escalabilidade da imple-

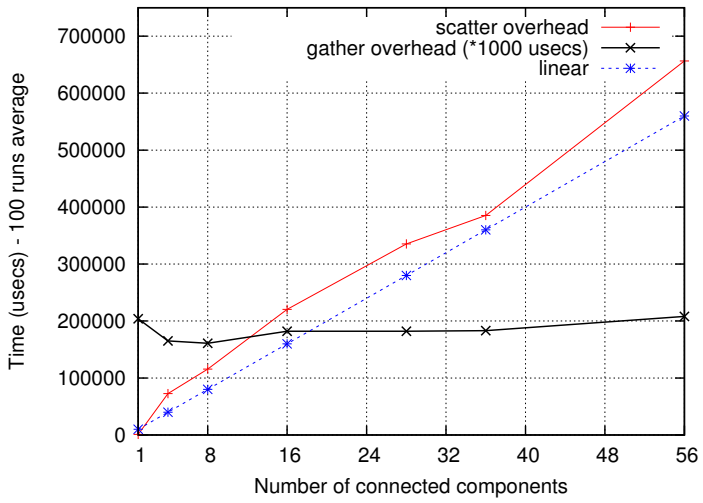

Figura 6.2: Sobrecarga imposta pela implementação das políticas *scatter* e *gather*

mentação dessas políticas quando o número de componentes cresce. O crescimento da curva para a estratégia *gather* encontra-se próxima de um comportamento logaritmo (quase constante). Isso pode ser explicado pela pelo fato da *GatherFacet* não receber as requisições no mesmo tempo. Como os clientes que invocam a *GatherFacet* estão executando em paralelo, eles invocam a *GatherFacet* em tempos ligeiramente diferentes o que diminui a sobrecarga do algoritmo que agrupa os parâmetros, pois o mesmo executa para a maioria das invocações em momentos distintos.

O crescimento da curva para a estratégia *broadcast* é praticamente constante, pois no caso da *GatherFacet* o parâmetro que foi retornado da invocação da *sub-interface* interna somente precisa ser retornado na invocação que foi feita na *sub-interface* externa pelos objetos remotos.

A oscilação atípica da curva de escalabilidade da política *broadcast* para 64 componentes pode ter sido consequência do uso compartilhado do cluster e da interferência de outros processos no momento da execução.

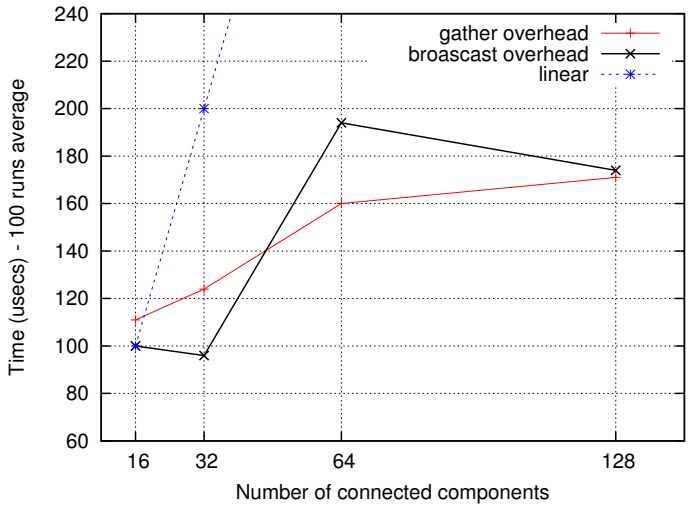

Figura 6.3: Sobrecarga imposta pelas implementações de *gather* e *broadcast*

Não foi possível a verificação de escalabilidade das políticas *reduce* (parâmetros) e de *scatter* (retorno) para o conector *GatherFacet*, pois não foi implementado um exemplo que utilizasse tais políticas.

6.1.2 Speedups obtidos

O *speedup* é um medida muito utilizada para verificar a qualidade da paralelização de uma aplicação. Se um programa precisa de um tempo X para executar com algoritmo serial, então ele precisará de, por exemplo, um tempo dez vezes menor para executar em 10 máquinas com um algoritmo

paralelo. Supondo T_1 como sendo o tempo de execução serial de um programa e T_p como sendo o tempo tempo de execução de um programa paralelo em p processadores, tem-se a seguinte proporção para o *speedup* S_p:

$$S_p = \frac{T_1}{T_p}$$

Como complementação dessa análise dos conectores paralelos, foi realizada também uma análise do *speedup* alcançado pelas aplicações que exemplificam o uso dos mesmos. Essa análise foi feita com até 28 nós de execução, número máximo de núcleos de processamento do *cluster* utilizado para os testes, sendo realizada para o demo `PI_Dartboard`.

Demo PI_Dartboard

Dada a arquitetura de paralelização de foi aplicada (*Master-Worker*) e a natureza *embarrassingly parallel*[1] (embaraçosamente paralelo) do problema, era esperado uma boa escalabilidade para a paralelização, que foi confirmada na curva de *speedup* da Figura 6.4.

Foram realizados testes inicialmente com geração de 1 bilhão de pontos randômicos e também com 2 bilhões de pontos, todavia a escalabilidade de ambos mostrou-se bem semelhante, o que pode ser explicado pela quantidade reduzida de comunicação (somente um ponto de sincronização) e de dados a serem enviados (somente é enviado o número de pontos que cada *worker* deverá gerar, e cada *worker* retorna a sua aproximação para o número π).

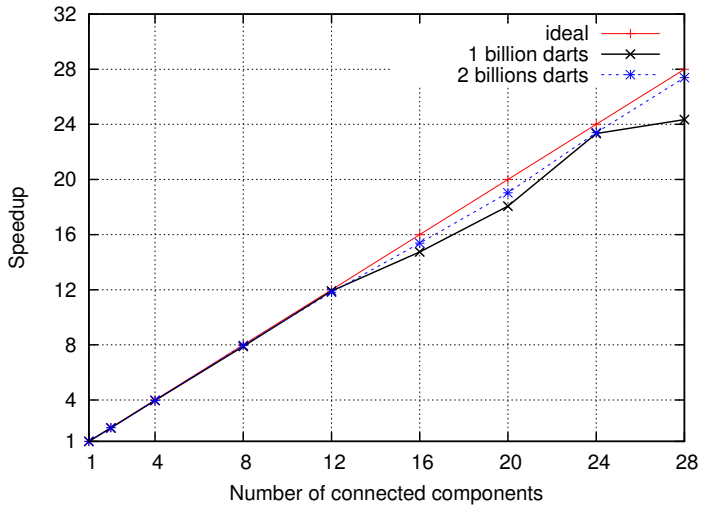

Figura 6.4: *Speedup* da aplicação `PI_Dartborad`

[1]Em computação paralela, uma carga de trabalho *embarrassingly parallel* (ou *embarrassingly parallel problem*) é aquela para a qual pouco ou nenhum esforço é necessário para separar o problema em um número de tarefas paralelas.

6.2 Avaliação qualitativa

Nas próximas seções serão apresentados os principais desafios e decisões que permearam a adaptação do conceito de *Interfaces Coletivas*, para a implementação C++ de um middleware orientado a componentes que utiliza tecnologia CORBA para comunicação distribuída.

6.2.1 Desafios de uma implementação C++ das ICs

Como descrito no Capítulo 3, a implementação padrão das *Interfaces Coletivas* feita pelo GCM faz uso da API reflexiva da linguagem Java para a leitura dos metadados de configuração paralela presentes nas interfaces. A implementação SCS das ICs foi realizada em C++, uma linguagem compilada com verificação estática de tipos e que, em seu padrão, não possui suporte oficial para reflexão computacional [38].

Este trabalho propôs uma abordagem semelhante a solução proposta por Devadithya et al. [39], que é compatível com o padrão da linguagem C++ e utiliza geração de código através de metadados.

O SCS utiliza a linguagem CORBA IDL para a definição de suas interfaces, de onde são gerados os skeletons e stubs responsáveis pela comunicação entre os objetos remotos. Como não existe nenhuma definição de metadados no padrão da linguagem IDL, a abordagem utilizada nesse trabalho definiu um grupo de anotações a serem utilizadas diretamente na definição das interfaces IDL como descritos no Capítulo 4, e propôs a criação de um parser de IDL para a geração do código C++.

6.2.2 Requisitos para a integração de ICs em middlewares de Componentes

Os requisitos que serão elicitados nessa seção foram provenientes das necessidades que apareceram no decorrer do desenvolvimento do SCS-Collective. A medida que o conceito das ICs era adaptado para o SCS, eram verificados os mecanismos que o SCS já disponibilizava (ou não) visando uma solução apropriada na implementação.

A seguir estão descritos os principais pontos que são necessários para uma implementação das ICs em um middleware de componentes e as respectivas soluções adotadas para o middleware SCS.

Suporte para definição de conectores duais

Na maioria dos modelos de componentes como CCM, FRACTAL, SCS e CCA, os serviços disponibilizados por um componente são definidos através de uma única interface (ou *faceta*), onde ficam definidos os possíveis comportamentos daquele serviço. Essa definição de comportamento entre o provedor do serviço e o cliente, segundo Liu e Cunningham em [40], pode ser visto como um *contrato de software*.

Conforme descrito neste documento, as *Interfaces Coletivas* são formadas necessariamente pela definição de duas *sub-interfaces*, a interna e a externa. Por exemplo, no caso da *GatherFacet* as requisições dos consumidores do serviço são realizadas através de sua *sub-interface* externa (interface exposta). Entretanto, a implementação do serviço disponibilizado é definida pela *sub-interface* interna. Esse mecanismo dual implica de certa forma na quebra do conceito de *contrato de software*, pois a implementação do serviço disponibilizado não é mais definido pela respectiva interface oferecida para seus clientes.

Outro problema é que essas *sub-interfaces* precisam estar devidamente associadas. Se a definição de um método na *sub-interface* externa possui uma determinada assinatura e está anotado com uma dada política de distribuição de dados, ele requer que na *sub-interface* interna esse método possua uma assinatura específica que seja compatível com a combinação de assinatura e políticas definidas.

Uma abordagem interessante para essa questão poderia ser a geração da definição da *sub-interface* interna a partir da *sub-interface* externa anotada com as especificações das estratégias de redistribuição de dados a serem utilizadas. Isso poderia implicar na necessidade da adição de mais metadados à definição da *sub-interface* externa, todavia o processo seria menos susceptível a erros.

Neste trabalho, foi definida uma estratégia de geração dos esqueletos de implementação da *sub-interface* interna a partir de informações da externa. Como essa estratégia não teve implementação completada, ela pode ser melhorada para realizar também a geração da própria definição da *sub-interface* interna, conforme descrito no parágrafo anterior.

Meios eficientes de cópia de memória

Um gargalo de desempenho que era claro desde o início da implementação foi o número consideravelmente grande de cópias de memória que precisariam ser feitas na implementação das políticas de redistribuição de dados. Além de ser uma operação custosa, o excesso de diretivas de alocação e desalocação de

memória pode causar *memory leaks* e tornar a aplicação instável, dificultando a correção de erros.

O SCS utiliza o padrão CORBA que modela uma lista de dados com um estrutura chama `sequence` que fornece apenas capacidades básicas para gerenciamento de memória e não se mostrou interessante, apesar de ter sido a opção utilizada.

Mecanismo de comunicação eficiente

Em computação paralela é muito importante que mecanismos de sincronização paralela possuam desempenho razoável. Nas implementações do padrão MPI as rotinas de comunicação coletiva possuem algoritmos extremamente otimizados como Luecke et al. [37] mostram.

O conceito de ICs não define uma arquitetura de comunicação a ser usada, ficando a cargo da infraestrutura de comunicação do middleware onde as ICs são implementadas a responsabilidade de prover um desempenho aceitável para se que obtenha uma boa escalabilidade na sincronização paralela.

O SCS realiza a comunicação entre os objetos distribuídos através do padrão CORBA. Conforme Denis et al. demonstram em [41], implementações desse padrão como Orbix [42], OmniORB [43], TAO [44] e MICO [45] foram desenvolvidos levando em consideração requisitos de desempenho na comunicação e podem alcançar resultados semelhantes ao MPI. O SCS-Collective foi implementado na versão C++ do SCS e utilizou o ORB MICO para a comunicação, estando assim em condições de competir com o MPI em desempenho. Um primeiro passo para isso seria a otimização dos algoritmos de redistribuição já implementados. Outra otimização importante que poderia ser aplicada seria a utilização da rotina **`send_multiple_requests_deferred`** que o padrão CORBA define para realização das invocações paralelas para o *MulticastReceptacle*, eliminando a ineficiencia da criação de uma *thread* para cada invocação.

Outro ponto importante seria a implementação de esquema de envio baseado em árvore para as operações coletivas como, segundo Luecke [37], é feito no MPI. Nessa estratégia o envio se inicia pela comunicação a somente dois processos, estes dois reenviam para mais dois até que todos tenham recebido os dados. Para suportar essa estratégia, tendo em vista o *MulticastReceptacle*, talvez fosse preciso definir um componente controlador que fizesse o gerenciamento dessa distribuição.

Todavia, melhores estudos seriam necessários para a validação dessa estratégia, pois existem algumas diferenças na forma como a comunicação é realizada nas ICs e no MPI. Por exemplo, no MPI não existe o retorno de dados nas operações coletivas como nas ICs: os dados são enviados e é

retornado somente um valor do *status* de finalização da operação. Para que ocorra o retorno dos resultados para o processo *Master* é necessário que os *Workers* iniciem outra operação de comunicação.

Controladores para coordenação de conexões

Uma das questões mais básicas descobertas durante o processo de desenho do conector *GatherFacet* era como tornar uma *GatherFacet* ciente do número de receptáculos aos quais ela está conetada. A *GatherFacet* precisa dessa informação para fazer a sincronização das chamadas dos receptáculos, pois sem essa informação não há como saber se o número de invocações feitas já é igual ao número de receptáculos conectados para que a invocação na *sub-interface* interna seja finalmente realizada. Tipicamente, um receptáculo sabe quantas facetas estão conectadas a ele, mas uma faceta não sabe a quantos receptáculos está conectada.

Além dessa questão existem vários outros ajustes que precisam ser aplicados aos conectores para que a sincronização paralela possa ser efetivada, como o incremento de contadores diversos e o gerenciamento de semáforos.

A solução adotada para esse problema foi a adição de uma faceta de configuração `ICollective` no modelo do SCS que ficaria responsável por agregar todas as operações de configuração do conectores *GatherFacet* e *MulticastReceptacle*, conforme descrito no Capítulo 4.

O GCM define explicitamente controladores de componentes para realizarem essa coordenação [1].

API de requisições robusta e flexível

Para que uma implementação de ICs seja simplificada é interessante que os mecanismos de construção de requisições disponibilizados pelo middleware possuam flexibilidade para a criação dinâmica de requisições e viabilizem meios otimizados para a realização de grupos de requisições paralelas.

O CORBA, padrão utilizado pelo SCS para comunicação, disponibiliza uma API para gerenciamento de requisições dinâmicas chamada *Dinamic Invocation Interface* - DII, que foi a solução adotada nesse trabalho. Como descrito no Capítulo 4, essa API permite a criação de requisições através da montagem de parâmetros e disponibiliza formas assíncronas e síncronas de invocação, além de oferecer meios para invocação paralela de grupos de requisições.

Reflexão computacional versus desempenho

O mecanismo de reflexão computacional permite que a configuração e coordenação do paralelismo seja feita de uma forma simples e intuitiva e possibilita a separação do código de coordenação do código funcional da aplicação. Para isso, entretanto, a API de configuração da ferramenta de paralelização geralmente introduz sobrecarga computacional na aplicação causando perda de desempenho.

Existe também a necessidade de mais passos para se completar o processo de criação da aplicação como demonstram os exemplos de código apresentados no Capítulo 5. Apesar desses passos serem relativamente simples, eles podem gerar um código genérico que não se beneficia de particularidades da infraestrutura alvo da aplicação. Apesar de facilitar o processo de configuração, recursos de *hardware* podem estar sendo subutilizados e, a depender do comportamento distribuído da aplicação, isso pode significar perda de desempenho.

No MPI, por exemplo, toda coordenação e sincronização é definida no código da aplicação. É possível configurar cada aspecto da comunicação, ajustando-se parâmetros, momento extato de invocação, forma de invocação a depender da infraestrutura de hardware disponível e outros detalhes. Comparando com uma ferramenta de abstração de programação de nível mais alto como o SCS-Collective, o tempo necessário de desenvolvimento e depuração de um programa paralelo com MPI é maior, além de produzir como resultado software de difícil reutilização e baixa manutenibilidade, conforme já discutido na seção 2.1.1.