1
Introdução

Computação de Alto Desempenho (do Inglês, *High Performance Computing* - HPC) é um campo da Computação de grande relevância atualmente. Esse fato decorre do constante aumento das necessidades computacionais de códigos de processamento massivo e da popularização de máquinas multi-núcleo e *clusters* de computadores.

Apesar desse crescimento, o desenvolvimento de aplicações paralelas ainda é uma tarefa complexa, como explica Akon et al. [3], e não se beneficia completamente de aspectos já bem estabelecidos na Engenharia de Software tradicional, tais como o paradigma de Componentes de Software descrito por Szyperski [4]. Esse paradigma apresenta crescente importância por oferecer abstrações mais apropriadas para a manipulação de softwares mais complexos, além de facilitar a gerência do ciclo de vida da aplicação, tópicos estes de grande importância em Computação Distribuída ou Paralela como mostra Motta [5].

Uma das primeiras tentativas de padronização do desenvolvimento de software paralelo surgiu com a definição do MPI [6] (do Inglês, *Message Passing Interface*). Baseado no *paradigma procedural*, o MPI disponibiliza para o desenvolvedor um conjunto de primitivas básicas de comunicação *one-to-one* e primitivas de comunicação coletiva, proporcionando um bom suporte aos padrões de paralelismo mais utilizados, como SPMD (do Inglês, *Single Process Multiple Data*), MPMD (do Inglês, *Multiple Process Multiple Data*) e Mestre-Escravo.

Apesar disso, de acordo com Baude et al. [1], o padrão MPI acaba deixando a cargo do desenvolvedor toda a parte de planejamento, arquitetura e orquestração da paralelização, e induz a mistura de código de comunicação e coordenação da paralelização com o código funcional da aplicação, como no Código 1.1, onde pode-se verificar a mistura de código de coordenação/distribuição (aspectos não funcionais) no código da lógica da aplicação.

Tal abordagem traz uma série de problemas como, por exemplo, o comprometimento da *manutenibilidade* do software, a diminuição da *reusabilidade* do código e a inclusão de aspectos não funcionais no processo de construção do software, o que pode degradar a *produtividade* no desenvolvimento, pois além

Código 1.1: Estrutura típica de um programa que usa MPI para paralelização.

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4
5 int main (argc, argv)
6      int argc;
7      char *argv[];
8 {
9   int rank, size;
10  /* Código relacionado a infraestrutura de paralelização */
11  MPI_Init (&argc, &argv);  /* starts MPI */
12  MPI_Comm_rank (MPI_COMM_WORLD, &rank);  /* get current process id */
13  MPI_Comm_size (MPI_COMM_WORLD, &size);  /* get number of processes */
14  /* Código funcional (lógica de aplicação) */
15  printf( "Hello world from process %d of %d\n", rank, size );
16  /* Código relacionado a infraestrutura de paralelização */
17  MPI_Finalize();
18  return 0;
19 }
```

de gerenciar a lógica de negócio, aspectos da distribuição também precisarão de atenção.

Um outro aspecto importante relacionado com a simplificação do desenvolvimento e manutenção de sistemas paralelos é a possibilidade de programação através de *interfaces* que expressem uma semântica própria da aplicação e que manipulem diretamente tipos de dados estruturados ou abstratos do domínio da aplicação. Arquiteturas como as utilizadas por ANTHILL [7] e GCM [1] são exemplos dessas interfaces. Apesar do conjunto de primitivas de comunicação do MPI possibilitarem a utilização dos padrões de paralelização mais comuns, a sua utilização não é trivial, além de ser muito susceptível a erros: operações de *gathering* (agrupamento) e *scattering* (distribuição) de dados exigem que o desenvolvedor construa estruturas de dados elaboradas para indicar ao MPI como realizar a redistribuição de dados.

Devido às limitações que podem ser observadas em MPI, diversos trabalhos têm sido propostos com o intuito de oferecer um melhor suporte ao desenvolvimento de aplicações paralelas, através da *separação de código de comunicação e de computação*, da *orquestração automática* das atividades paralelas, da maior facilidade de *expressão de comunicação coletiva*, do suporte aos *padrões de paralelização* mais comuns, e de *interfaces de programação* de mais alto nível. Alguns exemplos de trabalhos nessa linha são:

– ANTHILL [7] é um sistema para desenvolvimento de aplicações paralelas que segue o paradigma de programação *Filter/Stream* para proporcionar paralelização tipicamente MPMD. Essa abordagem traz consigo a separação de comunicação (*streams*) e computação (*filtros*), e oferece um esquema simples de comunicação coletiva através de *Labeled Streams*.

– Charm++ [8] é um ambiente de computação paralela e distribuída orientado a objetos, cuja comunicação é baseada em passagem de mensagens assíncronas. Por ter a unidade de distribuição definida em termos de objeto, o Charm++ possui uma granularidade fina e é apropriado para execuções com milhares de nós, como é mostrado por Kale et al. [9]. Proporciona comunicação coletiva através de invocação a coleções de objetos.

– O modelo de componentes *Common Component Architecture* (CCA), definido por Armstrong et al. [10], tem como objetivo principal o suporte à computação científica. Ele define os padrões de comunicação *Provides/Uses* e SCMD (do Inglês, *Single Component Multiple Data*), padrão análogo ao SPMD.

– O *Grid Component Model* - GCM [1] é uma extensão do modelo hierárquico de componentes Fractal [11]. O GCM tem como principal meta gerenciar todos os aspectos do ciclo de vida de uma aplicação distribuída, permitindo que toda a orquestração da comunicação da aplicação seja realizada à parte do código funcional. Para realizar comunicação em grupo, o *GCM* usa o conceito de *Interfaces Coletivas* - ICs [12], que permite de uma maneira intuitiva definir o comportamento paralelo de um componente diretamente na sua interface.

Abordagens baseadas em componentes de software, tais como CCA e GCM, já induzem, por *design*, a separação entre código de comunicação/coordenação e código de computação, ao tratarem as unidades de computação (componentes) como *caixas pretas* que se comunicam com outras unidades de computação através de interfaces bem definidas. As conexões entre os componentes são tipicamente definidas externamente a estes. No caso de GCM, as ICs também são responsáveis por definir a coordenação da comunicação entre os componentes, e permitem que o comportamento paralelo do componente seja anotado em suas interfaces. Essas características das ICs tornam a definição da comunicação em grupo uma tarefa simples e facilita o reuso dos componentes da aplicação. Essas anotações indicam, por exemplo, quais estratégias de distribuição de dados serão utilizadas.

A implementação de referência do modelo GCM, baseada no *framework ProActive* [13, 14], foi desenvolvida na linguagem Java e utiliza intensamente mecanismos de reflexão computacional oferecidos por essa linguagem, sendo esse um dos principais desafios para o uso de ICs em implementações puramente C++ desse mecanismo. Mathias et al. [15] propuseram melhorias para suporte a C++ no GCM, todavia essas melhorias constituíram-se somente de

bindings (ligações) de acessoo ao núcleo da implementação que permaneceu em linguagem Java.

1.1 Objetivos e Contribuições

Este trabalho tem como objetivo realizar um estudo do mecanismo de sincronização paralela entre componentes denominado *Interfaces Coletivas* [12]. Esse estudo foi baseado em uma implementação desse mecanismo no middleware de componentes SCS, onde foram projetados e implementados dois conectores para sincronização e comunicação paralela.

Essa implementação viabilizou uma análise dos requisitos para a integração das *Interfaces Coletivas* em um middleware orientado a componentes e possibilitou a identificação dos desafios de implementar esse mecanismo em uma linguagem como C++, amplamente usada em aplicações científicas.

Outro estudo possibilitado por essa implementação consistiu da avaliação da extensibilidade dos conectores presentes no modelo de componentes SCS que não foram projetados com foco em computação de alto desempenho ou paralelismo.

Foram definidos e implementados os conectores *GatherFacet* e *MulticastReceptacle*. O primeiro é um tipo de faceta especializada em comunicação em grupo que realiza agrupamento de invocações, enquanto o segundo possui a forma de um receptáculo que possui a capacidade de gerar invocações paralelas. O SCS-Collective (nome do middleware SCS estendido com esses conectores para comunicação coletiva) será descrito em mais detalhes no Capítulo 4.

As contribuições deste trabalho podem ser resumidas primordialmente nos seguintes pontos:

- A identificação dos desafios de uma implementação em uma linguagem como C++, muito usada em aplicações científicas;
- A identificação dos requisitos básicos para que o mecanismo de ICs seja integrado a um middleware orientado a componentes.

1.2 Estrutura do Documento

Este documento está organizado da seguinte forma: o Capítulo 2 descreve o estudo e a comparação dos trabalhos relacionados; o Capítulo 3 apresenta o middleware de componentes SCS e descreve do conceito das *Interfaces Coletivas*; o Capítulo 4 apresenta o SCS-Collective; o Capítulo 5 descreve

o uso dos conectores *MulticastReceptacle* e *GatherFacet*, com a criação passo a passo de aplicações exemplo; no Capítulo 6 é apresentada uma análise de desempenho das ICs no SCS-Collective e uma análise dos desafios de sua implementação; finalizando com o Capítulo 7, que apresenta as conclusões e os possíveis trabalhos futuros.