2
Estudo sobre Ferramentas de Paralelização

Vários trabalhos já foram desenvolvidos propondo abordagens para amenizar os problemas relacionados ao desenvolvimento de sistemas paralelos. Estes trabalhos trazem soluções que abrangem desde o projeto e programação desses sistemas, até a abstração de programação utilizada e o suporte à comunicação coletiva.

Neste capítulo será apresentado um estudo comparativo entre algumas dessas ferramentas de programação paralela, visando oferecer uma visão geral do estado atual de sistemas e ferramentas de suporte a Computação de Alto Desempenho.

Dada a grande variedade de sistemas e ferramentas na área de computação paralela, e levando em conta que o foco principal do estudo aqui apresentado está nos mecanismos de comunicação coletiva oferecidos, reduzimos o o escopo do estudo a ferramentas para computação paralela distribuida, pois tais ferramentas tipicamente se baseiam em mecanismos de troca de mensagens. Em termos de abstração, procuramos escolher ferramentas que tratem software como um conjuto de entidades interligadas através de uma arquitetura de comunicação, por mais se assemelharem à abstração de componentes de software.

Considerando esses critérios, as ferramentas escolhidas foram: o MPI [6], o Anthill [7], o Charm++ [9], o *Common Component Architecture* - CCA [10] e o *Grid Component Model* - GCM [1].

Na próxima seção será apresentada uma breve descrição de cada ferramenta, seguida da seção 2.2 onde será feita uma comparação dos ambientes. A seção 2.3 conclui este capítulo com algumas considerações finais.

2.1
Ambientes de Paralelização Estudados

Nesta seção serão apresentadas as principais características das ferramentas de paralelização que foram estudadas neste trabalho. Procuramos abordar principalmente aspectos relacionados à granularidade de paralelização (tamanho da unidade de paralelização) que as ferramentas proporcionam e a arqui-

tetura de comunicação coletiva que é definida para a interação paralela entre essas unidades.

2.1.1 MPI

Descrito por Gropp et al. [6], o MPI (*Message Passing Interface*) é um padrão de passagem de mensagens definido por um amplo comitê de empresas, desenvolvedores e usuários, que especifica um conjunto de rotinas e regras para o suporte ao desenvolvimento de aplicações paralelas. Diversas implementações desse padrão foram feitas, destacando-se, a implementação de código aberto OpenMPI[1] e o MPICH2[2], conhecido pela sua portabilidade e alto desempenho, sendo que ambos implementam a versão 2.0 do padrão descrita por Ewing and Thakur [16].

O MPI foi uma das primeiras tentativas bem sucedidas de padronização no cenário de Computação de Alto Desempenho, e, por ser baseado no paradigma procedural, passou a ser muito utilizado para a paralelização de aplicações científicas de processamento massivo, pois em sua ampla maioria foram construídas em linguagens procedurais como Fortran e C.

A paralelização com MPI é alcançada através da troca de mensagens entre os processos participantes de um grupo de comunicação. O padrão dispõe de rotinas para comunicação ponto a ponto como, por exemplo, `MPI_Send` (envio) e `MPI_Receive` (recebimento), e comunicação em grupo (ou comunicação coletiva), como `MPI_Gather`, que faz um agrupamento dos dados de todos os processos do grupo de comunicação em um único processo.

Estas rotinas de comunicação também possuem suas respectivas versões não bloqueantes (ou assíncronas). Um envio não bloqueante pode ser realizado com a rotina `MPI_Isend`, por exemplo.

Comunicação em grupo, ou coletiva, permite a realização de algumas operações de sincronização paralela mais complexas com estratégias elaboradas para distribuição dos dados. As operações de comunicação coletiva mais usadas e que possuem suporte no MPI são:

- ***gather***: Uma operação de *gather* faz com que os dados enviados por cada membro de um grupo de processos sejam agrupados e o resultado desse agrupamento seja colocado em um processo destinatário. Por exemplo, se três processos enviam um vetor de tamanho 3, o destinatário recebe

[1] O OpenMPI pode ser encontrado em `http://www.open-mpi.org/`.

[2] O MPICH2 pode ser encontrado em `http://www.mcs.anl.gov/research/projects/mpich2/`.

um vetor de tamanho 9. No MPI essa operação é realizada através da rotina `MPI_Gather`.

- ***scatter***: Em uma operação *scatter*, um membro do grupo envia dados que serão igualmente repartidos entre todos os membros do grupo. Por exemplo, se um processo envia um vetor de tamanho 10 para um grupo de 5 processos, os 5 processos recebem um vetor de tamanho 2. No MPI essa operação é implementada pela rotina `MPI_Scatter`.
- ***reduce***: Na operação de *reduce*, os dados enviados por todos os membros de um grupo sofrem redução por uma operação previamente definida (que pode ser, por exemplo, mínimo, máximo, soma, maioria) e o resultado dessa redução é passado para o destinatário. Por exemplo, se 8 processos enviam um vetor de inteiros de 3 posições, com o *reduce* configurado para soma, o destinatário recebe um vetor de tamanho 3, cujos elementos são resultado da soma das respectivas posições dos 8 vetores enviados. A rotina MPI que realiza essa operação é a `MPI_Reduce`.
- ***broadcast***: Em uma operação de *broadcast* um processo envia o mesmo dado para todos os integrantes do grupo de comunicação. No MPI é a rotina `MPI_Broadcast` que implementa essa operação.

O MPI também implementa rotinas para comunicação *all-to-all*, onde dados provenientes de todos os processos componentes do grupo de comunicação são agrupados com uma determinada estratégia e são entregues a todos os componentes do grupo. `MPI_Allgather` e `MPI_Allreduce` são exemplos desse tipo de comunicação. Na primeira é feita uma operação de *gather*, todavia, diferentemente do `MPI_Gather`, todos os processos do grupo de comunicação recebem os resultados. A segunda possui funcionamento semelhante, com a diferença de ser aplicada uma operação de redução.

Apesar desse suporte robusto a comunicação coletiva, a API do MPI é consideravelmente difícil de usar, pois para realizar uma operação como `MPI_Allgatherv` (variante da `MPI_Allgather` para envio de vetores com tamanhos diferentes) é necessário que o programador lide com estruturas de dados complexas (tipicamente vetores de *displacement* ou deslocamento e vetores de índice) para indicar como será feita a distribuição dos dados, o que pode deixar o processo de desenvolvimento propenso a erros.

Dada essa arquitetura de comunicação, a unidade de distribuição/paralelização proporcionada pelo modelo de paralelização adotado pelo MPI pode ser vista em nível de processos, pois é a entidade básica de interação paralela no MPI.

Código 2.1: Entrelaçamento de código de coordenação e código da aplicação em um programa MPI.

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <math.h>
4 int main( int argc, char *argv[] )
5 {
6     int n, myid, numprocs, i;
7     double PI25DT = 3.14159265358979323846264 3;
8     double mypi, pi, h, sum, x;
9
10    /* Código relacionado a coordenação da distribuição */
11    MPI_Init(&argc,&argv);
12    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
13    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
14
15    /* Lógica da aplicação */
16    while (1) {
17        if (myid == 0) {
18            printf("Enter the number of intervals: (0 quits) ");
19            scanf("%d",&n);
20        }
21        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
22        if (n == 0)
23            break;
24        else {
25            h   = 1.0 / (double) n;
26            sum = 0.0;
27            for (i = myid + 1; i <= n; i += numprocs) {
28                x = h * ((double)i - 0.5);
29                sum += (4.0 / (1.0 + x*x));
30            }
31            mypi = h * sum;
32
33            /* Código para distribuição de dados */
34            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
35           MPI_COMM_WORLD);
36            if (myid == 0)
37                printf("pi is approximately %.16f, Error is %.16f\n",
38                       pi, fabs(pi - PI25DT));
39        }
40    }
41
42    /* Código relacionado a coordenação de distribuição */
43    MPI_Finalize();
44    return 0;
45 }
```


2.1.2 Sistema de Runtime Anthill

O Anthill é um *framework* cujo objetivo principal é a construção de aplicações paralelas direcionadas a ambientes distribuídos heterogêneos [7], tipicamente ambientes de grade computacional. O Anthill possui uma estratégia de paralelização baseada no modelo *Filter/Stream* e implementa um esquema de roteamento de mensagens utilizando-se de rótulos.

O modelo de programação com filtros é bem semelhante ao sistema de *pipes* ou redirecionamentos de entrada/saída presente nas linguagens interpretadas do *shell* da maioria dos sistemas Unix atuais. No Anthill um programa é visto como uma série de "pedaços" de software ou filtros que são capazes de

receber dados pelas suas *streams* de entrada, processar estes dados e colocar a saída na respectivas *streams*. Neste sentido, uma aplicação completa é vista como um grafo em que os nós são os filtros e as arestas as ligações entre *streams* de entrada/saída de dois nós. A diferença básica entre o modelo usado no ANTHILL e o dos sistemas Unix, é que no ANTHILL existe a possibilidade de se ter mais de uma *stream* de entrada/saída em um mesmo filtro, como afirmam Ferreira et al. [7]. A Figura 2.1 mostra a arquitetura básica de um aplicação *Filter/Stream*.

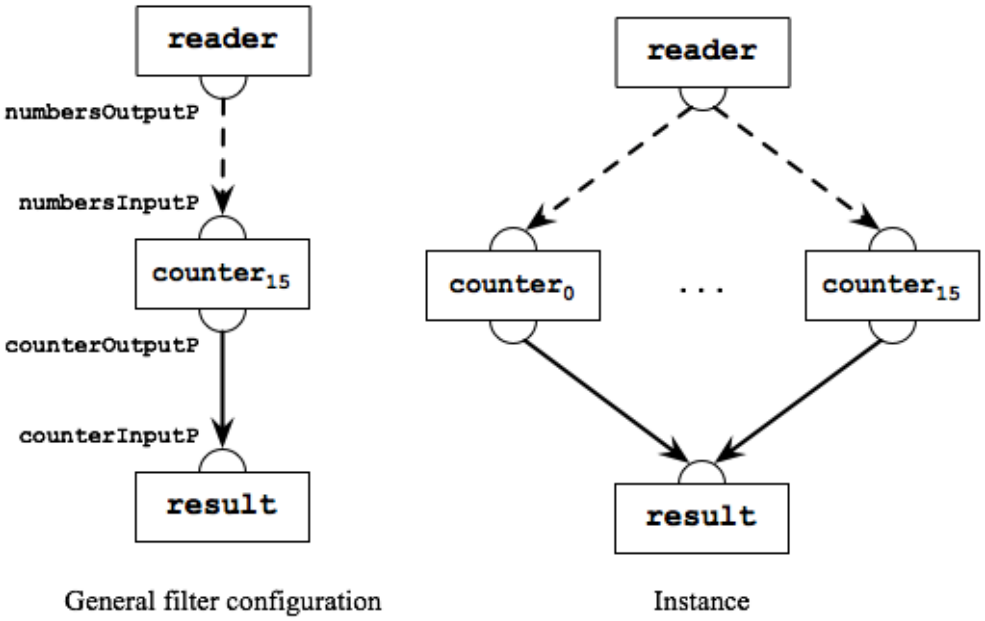

Figura 2.1: Estrutura de filtros de uma aplicação ANTHILL

A construção da aplicação pela composição de filtros traz algumas implicações, pois, além da aplicação estar dividida em unidades de software, essas unidades podem ser copiadas e levadas para diversos nós, de preferência aos nós onde reside o dado que será processado. Sendo assim, cada cópia do filtro pode processar uma determinada faixa local do dado e, assim, aumentar o ganho da paralelização. O Código 2.2 mostra a configuração de deployment e define o padrão de interação dos filtros da aplicação da Figura 2.1.

Esse esquema de levar o processamento para onde estão os dados traz algumas complicações quanto ao roteamento das mensagens entre os filtros. Para contornar esse problema, o ANTHILL faz uso de uma técnica chamada *Labeled-Stream*, onde através de um esquema de tuplas e de uma função de *hash*, é possível conseguir um roteamento customizado das mesmas e realizar a redistribuição dos dados com maior flexibilidade. É também possível selecionar vários filtros como destinatários, o que assemelha-se ao tipo de comunicação

Código 2.2: Configuração de uma aplicação Anthill.

```
<config>
  <hostdec>
    <host name="oceano" mem="2048">
      <resource name="1"/>
    </host>
  </hostdec>
  <placement>
    <filter name="reader" libname="reader.so" instances="1">
    </filter>
    <filter name="counter" libname="counter.so" instances="16">
    </filter>
    <filter name="result" libname="result.so" instances="1">
    </filter>
  </placement>
  <layout>
    <stream>
      <from filter="reader" port="numbersOutputP" policy="RR" />
      <to filter="counter" port="numbersInputP"/>
    </stream>
    <stream>
      <from filter="counter" port="counterOutputP" policy="broadcast"/>
      <to filter="result" port="counterInputP"/>
    </stream>
  </layout>
</config>
```


coletiva denominada *Multicast*.

O Anthill suporta tanto o paralelismo de tarefas (facilitado pela própria abstração de unidades de software ou filtros) quanto o paralelismo de dados (cópia de filtros para processamento de partes distintas dos dados). Além disso, como a comunicação no Anthill é realizada através da escrita de dados em *streams* (forma não bloqueante de comunicação), é possível que esses filtros operem de forma assíncrona, o que eleva o grau de eficiência de paralelização de forma geral.

Outro ponto importante no Anthill é o seu suporte a reconfiguração dinâmica e tolerância a falhas, aspectos fundamentais para a robustez de ferramentas direcionados a plataformas heterogêneas e distribuídas. Através do mecanismo denominado *Global Persistent Storage*, os filtros podem salvar o seu estado atual de processamento (um esquema parecido com *commit* de transações), o que constitui um eficiente método de recuperação em caso de falha. O esquema de *Labeled-Stream*, além ser um facilitador do roteamento de mensagens, também pode ser visto como um esquema de balanceamento de carga ou mesmo reconfiguração dinâmica do sistema. Caso uma parte da rede esteja congestionada ou mesmo algum nó esteja sobrecarregado, o mecanismo *Labeled-Stream* pode ser utilizado para redirecionar as novas mensagens para *hosts* que estejam com melhor disponibilidade.

O Código 2.3 exemplifica a implementação de um filtro de processamento no Anthill, com a definição e conexão das respectivas *streams* de entrada e saída.

Código 2.3: Definição do filtro *counter* da Figura 2.1.

```
1 #include "FilterDev.h"
2 #include "messages.h"
3 /* Definição das streams de entrada e saída de dados */
4 InputPortHandler numbersInputP;
5 OutputPortHandler counterOutputP;
6
7 /* Inicialização das streams */
8 int initFilter(void * work, int size) {
9   numbersInputP=dsGetInputPortByName("numbersInputP");
10  counterOutputP=dsGetOutputPortByName("counterOutputP");
11  return 0;
12 }
13
14 /* Lógica da aplicação: Implementação do filtro */
15 int processFilter(void * work, int size) {
16  int array_size = 64, nnumbers = 0, i = 0, sum = 0;
17  int *array;
18  message new_message;
19  array = (int*)malloc(sizeof(int)*array_size);
20
21  /* Ler os dados da stream de entrada */
22  while ( dsReadBuffer(numbersInputP, (&new_message), sizeof( message))!=
      EOW ) {
23    if(nnumbers > array_size) {
24      array_size += array_size;
25      array = (int*)realloc(array,sizeof(int)*array_size);
26    }
27    array[nnumbers++] = new_message.number;
28  }
29  for(i = 0; i < nnumbers; i++){
30    sum += array[i];
31  }
32  /* Envia o resultado pela stream de saída */
33  new_message.number = sum;
34  dsWriteBuffer(counterOutputP, &new_message, sizeof(int));
35  return 0;
36 }
```

2.1.3 Charm++

O Charm++[3] nasceu com o objetivo de fornecer uma abstração de programação flexível orientada a objetos para a programação paralela. Por definição, pode-se dizer que o Charm++ é um paradigma de programação paralela orientado a objetos e baseado na passagem de mensagens assíncronas [9], cuja implementação foi baseada na linguagem C++, sobre a qual adiciona algumas funcionalidades e estruturas.

O Charm++ oferece suporte a orientação a objetos através de estruturas especiais chamadas *Chare Objects*. Essas estruturas são semelhantes aos objetos de C++, possuindo estado e métodos, e são o elemento básico de distribuição. Esses *Chare Objects* se comunicam através de passagem de mensagens assíncronas. O envio de uma mensagem é feito através da invocação a métodos especiais chamados *Entry Methods*. Utilizando-se da terminologia de

[3]Criado no Laboratório de Programação Paralela do Departamento de Ciência da Computação da Universidade de Illinois em Urbana Champaign: `http://charm.cs.uiuc.edu/`.

Componentes de Software, esses métodos funcionam como se fossem funções que foram definidas na interface de um componente e que podem ser invocados externamente. Um exemplo desses *Chare Objects* pode ser visto no Código 2.4 onde está definida a interface *Chare Object Hello*, seguido pela implementação da mesma no Còdigo 2.5.

Código 2.4: Definição da interface do *Chare Object Hello*.

```
module hello {

  array [1D] Hello {
    entry Hello();
    entry void sayHi(int);
  };

};
```

Código 2.5: Implementação da *Chare Object Hello*.

```
/***********
 * Header  *
 ***********/

#ifndef __HELLO_H__
#define __HELLO_H__
/* Chare Object */
class Hello : public CBase_Hello {
 public:
  Hello();
  /* Construtor necessário para a migração de objetos */
  Hello(CkMigrateMessage *msg);

  /* Entry Methods */
  void sayHi(int from);
};
#endif //__HELLO_H__

/********
 * Impl *
 ********/
#include "hello.decl.h"
#include "hello.h"
#include "main.decl.h"

extern /* readonly */ CProxy_Main mainProxy;
extern /* readonly */ int numElements;
Hello::Hello() { }
/* Construtor necessário para a migração de objetos */
Hello::Hello(CkMigrateMessage *msg) { }

/* Entry Methods Impl */
void Hello ::sayHi(int from) {

  CkPrintf("\"Hello\" from Hello chare #%d on "
           "processor %d (told by %d).\n",
           thisIndex, CkMyPe(), from);
  /* Invoca próximo Chare object se existir ou termina */
  if (thisIndex < (numElements - 1))
    thisProxy[thisIndex + 1].sayHi(thisIndex);
  else
    mainProxy.done();
}
#include "hello.def.h"
```

O Charm++ também possui uma estrutura chamada de *Chare Collections*, que são grupos de *Chare Objects* espalhados em todos os processadores. Cada membro de uma *Chare Collection* realiza operações similares em grupos de dados distintos, o que constitui uma forma de paralelismo SPMD. Existem vários tipos de *Chare Collections* entre os quais podemos destacar os *Chare Arrays*, *Chare Groups* and *Chare NodeGroups* [17]. As *Chare Arrays* podem ser vistas como um vetor indexado multidimensional de *Chare Objects* espalhados em todos os processadores mediante um determinado esquema de mapeamento. Os *Chare Groups* são *Chare Arrays* com a restrição de execução dos objetos em um único processador físico. Os *Chare NodeGroups* são *Chare Arrays* com a restrição de execução dos objetos somente em um determinado nó de processamento (máquina).

A comunicação coletiva no Charm++ é baseada nessas estruturas coletivas. A referência para um *Chare Array* pode ser usada para a realização uma invocação de *Entry Methods* de todos os *Chare Object* pertencentes a este *array*. A invocação `a1.doIt(parameters)` seria um exemplo de *broadcast*, onde `a1` é um *Chare Array*. Através do uso das estratégias descritas por Kale e Krishnan [17], também são possíveis operações de redução. Nas referências estudadas sobre essa ferramenta não foram encontrados maiores detalhes de como se procedem operações do tipo *scatter* ou *gather*.

O Charm++ também suporta algumas características importantes em distribuição/paralelização de aplicações. Essas características proporcionam uma maior robustez ao ambiente e, de certa forma, proporcionam algum grau de autonomicidade ao sistema:

- **Balanceamento de carga:** No Charm++ existe a possibilidade de migração de objetos entre processadores. Esse processo é realizado pelos objetos denominados *Migratable Objects*.
- **Checkpointing:** Realizado através da migração de todos os objetos para o disco. Essa característica é especialmente importante, pois permite que a execução de um programa paralelo interrompida por alguma falha seja reiniciada a partir do mesmo ponto em algum momento posterior ou mesmo em outro ambiente de hardware.
- **Tolerância a falhas:** Se algum *host* apresentar problemas ou mesmo se já estiver *offline*, o Charm++ pode recriar os *Chare Objects* perdidos em outros processadores e continuar suas execuções.
- **Realocação dinâmica de recursos:** Como uma característica derivada da migração de objetos, essa realocação consiste da expansão ou contração do número de processadores sendo utilizados por determinado

programa. Esse mecanismo é possibilitado pela coleta de informações de tempo de execução sobre a carga dos processadores em questão.

Como forma de facilitar a paralelização de aplicações legadas e permitir também com que aplicações já paralelizadas se beneficiem das facilidades proporcionadas pelo ambiente CHARM++, foi criada uma ferramenta chamada ADAPTIVE MPI - AMPI [18], que consiste da implementação do padrão MPI sobre o sistema de *runtime* do CHARM++.

2.1.4 CCA

A tecnologia de Componentes de Software tem sido muito bem aceita na indústria de software. Modelos de componentes como o CCM [19], que é baseado no padrão OMG CORBA, e o OPENCOM [20], que é baseado no padrão de componentes COM da Microsoft, são algum exemplos de sucesso dessa tecnologia. Apesar dessa aceitação, essa tecnologia ainda não é adequada para o desenvolvimento de sistemas paralelos, pois, segundo Armstrong et al. [10] ela não foi idealizada considerando aspectos e padrões de arquitetura que são essenciais para HPC.

O *Common Component Architecture* (CCA) [21] apareceu justamente para sanar essa limitação de modelos de componentes até então, sendo completamente projetado com foco em Computação de Alto Desempenho.

Vale ressaltar que o CCA não esta atrelado a uma linguagem de programação específica, pois há uma considerável variedade de linguagens adequadas para computação de alto desempenho como C, C++ e as diversas variantes de Fortran (77, 90, 95 até 2003), tornando um suporte multi-linguagem um aspecto fundamental. Essa característica é alcançada através do uso da *Scientific Interface Definition Language* (SIDL) e da tecnologia CORBA como base, tornando-o também compatível com outros modelos de componentes compatíveis com CORBA. Como implementações do modelo CCA pode-se citar os *frameworks* SCIRUN [22], CCAFFEINE [23] e o XCAT [24].

O CCA foi arquitetado seguindo basicamente dois padrões de interconexão de componentes:

– ***Provides/Uses Ports.*** Esse padrão é relativamente comum em modelos de componentes e consiste da disponibilização pelo componente de uma coleção de recursos para outros componentes interessados. Normalmente esses recursos são uma coleção de subrotinas que poderão ser utilizadas pelo componente que os importar. No Código 2.6 está descrito um exemplo de código que define componentes CCA, e utiliza o padrão *Provides/Uses Ports* para interconexão dos mesmos.

Código 2.6: Componentes CCA com padrão de portas *Provides/Uses*.

```
1  /* SimpleEcho interface */
2  interface SimpleEcho{
3    public String echoHello ( String s);
4  }
5
6  /* SimpleEcho implementation */
7  public class SimpleEchoImpl implements SimpleEcho{
8     public String echoHello(String s) throws RemoteException {
9       return "SimpleEchoImpl says: Hello " + s;
10    }
11 }
12
13 /* Component that provides SimpleEcho service */
14 public class EchoPrinterComponent implements Component{
15    public void setServices(Services cc){
16      //defines a Port
17      PortInfo portType = new PortInfoImpl( "simpleEchoProvidesPort",
18                                            "http://example.com/echo.wsdl");
19      SimpleEchoImpl simpleEchoImpl = new SimpleEchoImpl();
20      //service becomes available for use
21      cc.addProvidesPort(simpleEchoImpl,portType);
22    }
23 }
24
25 /* Component that uses SimpleEcho service */
26 public class EchoGeneratorComponent implements Component{
27   public void setServices(Services cc){
28     //defines a Port
29     PortInfo portType = new PortInfoImpl( "simpleEchoUsesPort",
30                                           "http://example.com/echo.wsdl");
31     //bind the specified service
32     cc.registerUsesPort(portType);
33   }
34 }
35
36 /* Main class */
37 public class Main{
38   public static void main(String[] args){
39     //get service
40     SimpleEcho usesSimpleEcho = (SimpleEcho) cc.getPort("simpleEchoUsesPort
         ");
41     //use service
42     usesSimpleEcho.echoHello("Extreme Lab");
43     //release service
44     cc.releasePort("simpleEchoUsesPort");
45   }
46 }
```

– ***Single Component Multiple Data* (SCMD).** Em computação paralela tradicional, um padrão de paralelização amplamente utilizado é o SPMD. Nessa padrão, cópias idênticas de um programa processam diferentes partes dos dados. Segundo Armstrong et al. [21], o padrão SCMD é uma evolução lógica do SPMD, onde cada componente consiste de uma entidade fechada que precisa ser igualmente instanciada em cada processador.

O CCA suporta também o conceito de *Collective Ports*, que é uma extensão do esquema de *Ports*. Em [10] e [21], Armstrong et al. não descrevem detalhes de como é a estrutura desse esquema de comunicação coletiva, apenas explicam que a semântica de interação é bem semelhante à semântica das

operações coletivas conhecidas como *broadcast*, *gather* e *scatter* e que as *Collective Ports* são definidas de uma forma genérica suficiente para permitir que dados sejam distribuídos arbitrariamente sobre os componentes ligados.

2.1.5 GCM/Fractal

O *Grid Component Model*[4] (GCM) proposto por Baude et al. [1, 25] é uma extensão do modelo de componentes Fractal [11] para suporte ao desenvolvimento de componentes distribuídos autonômicos. É principalmente direcionado a ambientes distribuídos heterogêneos que evoluem dinamicamente, como os sistemas de computação em grade. Por ser baseado no Fractal, o GCM traz consigo uma série de características interessantes como:

- **Modelo hierárquico de componentes.** No GCM é possível a criação de componentes chamados *Composites*, que são composições de vários outros componentes. Esse modelo aumenta a flexibilidade para programação de aplicações e para a comunicação coletiva.
- **Capacidades de introspecção.** A capacidade de um sistema poder obter dados sobre a sua própria estrutura interna (também chamada Reflexão Computacional) constitui-se uma característica muito importante nos sistemas atuais, pois possibilita a realização ajustes estruturais e o carregamento de novas implementações em tempo de execução.
- **Reconfiguração dinâmica.** Reconfiguração dinâmica consiste da mudança nas ligações entre as partes/módulos de um programa durante sua execução. No GCM isso pode ser visto como a reestruturação das ligações entre os componentes que compõe a aplicação distribuída, e também pode ocorrer como uma forma de balanceamento de carga do sistema.

Analisando a nível de abstração de programação, o GCM suporta um gerenciamento de paralelização mais alto nível que ferramentas tradicionais. Em ferramentas como o MPI, por exemplo, a orquestração de todo o esquema de distribuição é definida pelo programador no código da aplicação, o que não é necessário no GCM, pois o mesmo realiza essa orquestração de forma automática e externa a aplicação.

O suporte a comunicação coletiva no GCM é baseado em interfaces, ou seja, os métodos e estratégias de comunicação são especificados na definição das interfaces do componente. A comunicação e redistribuição de dados é definida através de anotações na interface como é ilustrado no Código 2.7.

[4]Criado pelo CoreGRID (http://www.coregrid.net/), um grupo de pesquisa em redes e infraestruturas de software para distribuição em larga escala sediado na França.

Código 2.7: Anotações em uma interface GCM *Multicast*.

```
1 /* Interface Multicast */
2 public interface MulticastJacobiSolver {
3   /* ************************************
4    *Configuração da política ONE_TO_ONE  *
5    * para redistribuição de dados        *
6    * ************************************ */
7   @MethodDispatchMetadata(
8     mode = @ParamDispatchMetadata(
9       mode= ParamDispatchMode.ONE_TO_ONE
10    )
11  )
12  /* Código da lógica da aplicação */
13  public void jacobiSolver(List<LineData> borders);
14 }
```

Essa estratégia evita entidades intermediárias e proporciona uma maior eficiência e escalabilidade à comunicação, características essenciais para sistemas voltados para HPC. As interfaces básicas de comunicação coletiva são:

- **Interface Multicast.** Essa interface possibilita que um única invocação seja transformada em um lista de invocações como é ilustrado na Figura 2.2. Existem algumas estratégias de distribuição de dados que precisam ser aplicadas para que o processo se complete como o *broadcast* ou *scattering* dos dados a depender do seu tipo.
- **Interface Gathercast.** Essa interface realiza o trabalho oposto da interface *Multicast*: agrupa um conjunto de chamadas em uma única chamada (Figura 2.3). Também são configuráveis as estratégias de agrupamento dos dados recebidos como por exemplo *gathering* e *reduce*.

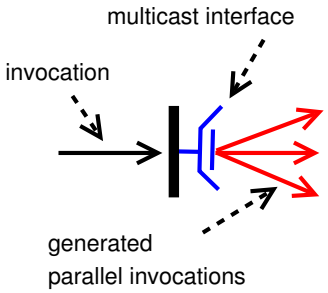

Figura 2.2: Interface GCM *Multicast*: Comunicação 1xN

Além da comunicação 1xN (*Multicast*) e Mx1 (*Gathercast*), existe o que Baude et al. [1] chamam de "Problema MxN", que consiste da comunicação e troca de dados eficiente entre os M e N componentes internos de dois *Composites* (composição de componentes). Uma solução inicial poderia ser o *binding* de uma interface *Gathercast* em um *Composite* com uma interface *Multicast* no outro *Composite*. Porém, essa solução possui um claro ponto

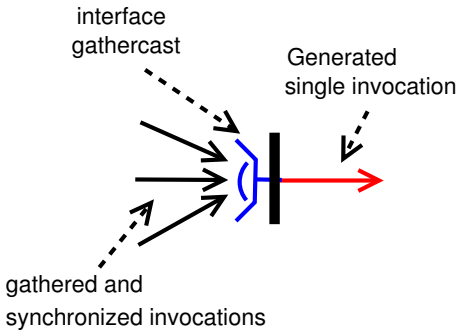

Figura 2.3: Interface GCM *Gathercast*: Comunicação Mx1

de gargalo, pois não há ligação direta (*direct binding*) entre os componentes comunicantes. Uma segunda abordagem seria a substituição das duas interfaces anteriores por M interfaces *Multicast* e N interfaces *Gathercast* o que possibilita a conexão direta entre os componentes participantes (ver Figura 2.4), e, consequentemente, melhor desempenho.

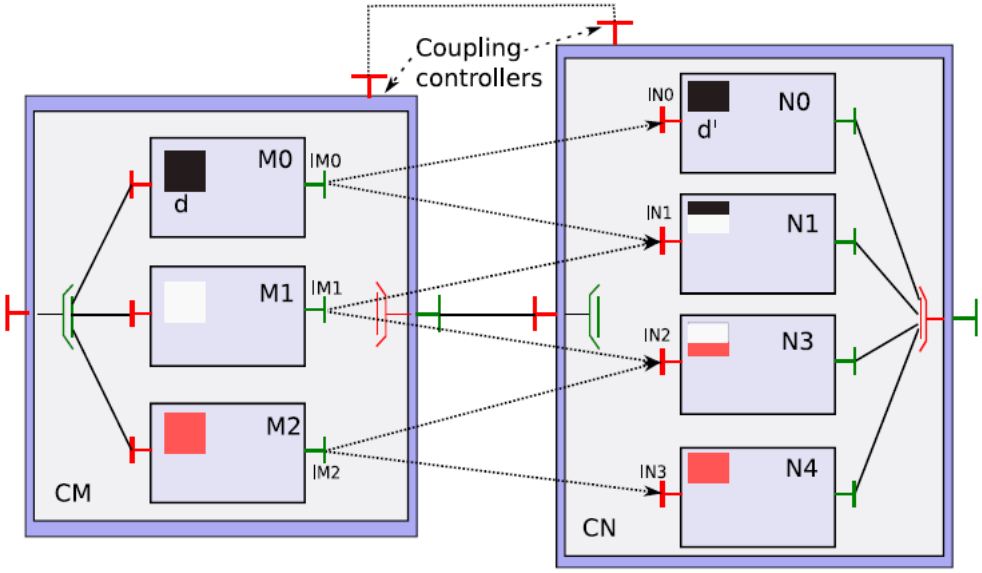

Figura 2.4: Bindig de M interfaces *Gathercast* com N *Multicast* (fonte [1])

2.2 Estudo Comparativo

Nesta seção será feita uma sumarização e comparação das características das ferramentas estudadas em relação aos seguintes critérios:

- **Conformidade dos ambientes em relação às 3 dimensões de paralelismo**: Como descrevem Ferreira et al. [7], a primeira dimensão de paralelismo refere-se à paralelização de dados/memória (*Single Process Multiple Data* - SPMD), a segunda à paralelização de controle, onde, diferentes processos possuem diferentes tarefas (*Multiple Process Multiple Data* - MPMD) e a terceira dimensão reside nas opções de comunicação assíncrona disponibilizadas.
- **Abstração de programação utilizada**: Esse critério descreverá o tipo de abstração de programação que cada ferramenta disponibiliza. Nas ferramentas estudadas estão presentes desde os mais antigos como a Programação Procedural ou mesmo Orientação a Objetos até abstrações mais atuais como *Componentes de Software* e programação *Filter/Stream* (ANTHILL).
- **Plataformas de execução preferencial**: As ferramentas estudadas também possuem determinados aspectos que as tornam mais apropriadas para uma determinada plataforma de hardware ou ambiente de execução. Como exemplo, pode-se citar o MPI, que, com suas sub-rotinas e algoritmos de comunicação extremamente otimizados, é apropriado para execução em *clusters* por se tratar de uma ambiente fechado e estável. O GCM, por exemplo, se adéqua melhor a um ambiente mais dinâmico como uma grade computacional por possuir algumas características de adaptação dinâmica e tolerância a falhas que podem facilitar o gerenciamento da aplicação em relação a modificações no ambiente de execução.
- **Suporte a Comunicação Coletiva**: O suporte a comunicação coletiva é um requisito primário em ambientes para computação paralela. Um bom suporte a comunicação coletiva proporciona flexibilidade a programação de uma aplicação paralela e evita que o programador crie suas próprias formas de comunicação, que podem não ser otimizadas ou adequadas para a topologia de rede ou o problema em questão.

2.2.1 Dimensões de Paralelismo

As ferramentas estudadas mostraram um bom suporte já em sua especificação[5] a SPMD, MPMD e assincronismo. O MPI proporciona paralelização SPMD e MPMD, e possui versões não bloqueantes das rotinas de comunicação, oferecendo dessa forma suporte a assincronismo.

O Anthill possui uma arquitetura baseada em filtros que podem tanto serem cópias de um mesmo código quanto podem representar partes complementares de uma aplicação, além do seu esquema de envio de mensagens assíncronas, o que confirma a sua adequação às três dimensões.

O GCM pode suportar tanto o assincronismo quanto o sincronismo, pois ele não especifica em sua definição uma estratégia fixa. O GCM suporta o paralelismo de dados (ex. um conjunto de componentes idênticos se comunicando via interfaces coletivas) e de tarefas (componentes realizando tarefas complementares interligados pelas interfaces coletivas) de forma implícita.

O Charm++ tem como unidade básica de distribuição os *Chare Objects* e, assim como no GCM, SPMD e MPMD podem ser alcançados através da forma como é implementada a aplicação com esses objetos. A comunicação no Charm++ é assíncrona e por passagem de mensagem.

O CCA foi construído seguindo, dentre outros, o padrão SCMD que é evolução do SPMD para componentes. Seu suporte a MPMD não fica claro nas referencias estudadas [10, 21], todavia, dado que é um modelo de componentes, a criação uma aplicação MPMD não deve demandar muitos esforços. Em sua definição o CCA é síncrono, pois a semântica de comunicação que utiliza é baseada em invocação remota de método.

Veja um resumo na Tabela 2.1.

	SPMD	**MPMD**	**Assincronismo**
MPI	Suporte nativo	Suporte nativo	Suporte nativo
Anthill	Filtros	Filtros	Mensagens assíncronas
Charm++	Chare Objects	Chare Objects	Mensagens assíncronas
GCM	ICs	ICs	Pelo ProActive
CCA	SCMD	implícito (RMI)	Síncrono

Tabela 2.1: Suporte às três dimensões de paralelismo

[5]Na maioria das vezes, apesar do suporte não ser definido na especificação do modelo, ele é provido por alguma implementação dessa especificação (*frameworks*).

2.2.2 Abstração de Programação

Uma abstração de programação define a forma de modelagem para resolução de determinado problema computacionalmente. O nível de uma determinada abstração pode influenciar diretamente no esforço em programação que é necessária para a construção de uma aplicação.

Nessa seção foi feita uma identificação e comparação dos paradigmas de programação que são proporcionados pela ferramentas estudadas. O resumo dessa comparação pode ser visto na Tabela 2.2.

	Programação Procedural	**Orientação a Objetos**	**Filter Stream**	**Componentes de Software**
MPI	Nativo	–	–	–
ANTHILL	–	–	Nativo	–
CHARM++	–	Nativo	–	–
GCM	–	Implícito	–	Nativo
CCA	–	Implícito	–	Nativo

Tabela 2.2: Paradigmas de Programação suportados por cada ferramenta

2.2.3 Plataformas de Execução

Os ambientes estudados possuem alguns comportamentos e características específicas que podem ser mais adequadas a determinadas plataformas de hardware paralelas, ou mesmo, já são desenhados para uma determinada plataforma, como, por exemplo, o GCM, que foi concebido para ambientes de grade.

Assim como com o GCM, o MPI também foi construído para uma arquitetura específica: aglomerados de computadores homogêneos interligados, também conhecidos como *clusters*. O CHARM++ foi desenhado para suportar uma maior variedade de ambientes, assim como é indicado para execuções compostas de milhares de nós [9].

O CCA veio proporcionar, inicialmente, a utilização de HPC em ambientes de memória compartilhada, pois outros modelos de componentes como o OpenCOM [20] e o CCM [19] não possuíam o suporte adequado para HPC. Com o seu desenvolvimento e utilização, e com o surgimento de várias ferramentas e *frameworks* baseados em sua especificação como o DCA [26] e o Xcat-c++ [27], o suporte a paralelização em ambiente de memória distribuída foi melhorado.

O ANTHILL tem como objetivo principal a construção de algoritmos e aplicações paralelas direcionadas tipicamente para ambientes de grade compu-

tacional [7], e pode, sem maiores problemas, ser utilizado em *clusters*, visto que um *cluster* pode ser considerado um subsistema de uma grade computacional. Seguindo esse mesmo raciocínio, o GCM também pode ser utilizado em *clusters*. Um resumo dessa comparação pode ser visto na Tabela 2.3.

	Multi núcleo	**Cluster**	**Grid**
MPI	Não otimizado	Suporte padrão	–
ANTHILL	–	Implícito	Suporte padrão
CHARM++	–	Suporte padrão	–
GCM	–	Implícito	Suporte padrão
CCA	Suporte padrão	Por *frameworks*	–

Tabela 2.3: Plataformas de hardware suportadas

2.2.4 Suporte a Comunicação Coletiva

O suporte a comunicação coletiva define as possibilidades de interação entre as unidades de distribuição do ambiente.

O GCM suporta esse tipo de comunicação através de *Interfaces Coletivas*, o ANTHILL suporta através do mecanismo de *Labeled-Streams*. O CCA tem, na sua definição, o conceito de *Collective Ports* para o suporte a comunicação coletiva, mas um suporte mais completo é somente encontrado nos *frameworks* nele baseados [27, 26]. O CHARM++ proporciona comunicação coletiva através dos *Chare Collections*, como já explicado na subseção 2.1.3.

Um resumo dessa comparação pode ser visto na Tabela 2.4.

	Mecanismo
MPI	Sub-rotinas especializadas
ANTHILL	Labeled-Stream
CHARM++	Chare Collections (invocações em grupo)
GCM	Interfaces Coletivas
CCA	Collective Ports

Tabela 2.4: Comunicação coletiva nos ambientes

2.3 Considerações Finais

Um aspecto importante e que motiva o desenvolvimento de ferramentas com melhores abstrações de programação é o esforço de programação para a paralelização de uma aplicação. Em ferramentas procedurais como MPI, todos os aspectos da paralelização precisam ser codificados na implementação

do programa, o que obriga o programador a lidar com aspectos não-funcionais do desenvolvimento e, consequentemente, degrada sua produtividade [5]. O GCM traz como principal diferencial o seu suporte tanto a definição explícita da paralelização quanto ao que chama de orquestração da paralelização externa as entidades paralelas utilizando-se de linguagens de *workflow* [1].

O suporte a características como balanceamento de carga, adaptação/reconfiguração dinâmica e tolerância a falhas são essenciais, dado que os ambientes paralelos de larga escala são dinâmicos, e evoluem ao longo do tempo (adição e remoção de recursos ou mesmo falhas destes). O GCM, o ANTHILL e o CHARM++ foram as ferramentas estudadas que possuem suporte a estas características.

A tecnologia de Componentes de Software, por sua flexibilidade e capacidades de reconfiguração dinâmica, está presente nas ferramentas estudadas. O GCM, que é uma extensão do modelo de componentes Fractal, e o CCA são exemplos da utilização dessa tecnologia em paralelismo.

Apesar dessas características, na questão de desempenho, ainda existem pontos a serem ajustados nas tecnologias de componentes existentes, como Armstrong et al. [10] mostram. Os conectores utilizados nos modelos de componentes tradicionais não são adequados e não têm um desenho apropriado para sincronização paralela eficiente.

Existem vários trabalhos que tentam encontrar uma solução para este problema como o CCA e, mais recentemente, o *framework* HPE proposto por Carvalho-Junior et al. [28], que é um ambiente baseado no modelo de componentes # (*# Component Model*) [29] e traz uma abordagem que faz uso da teoria de sistemas de tipos e é baseado no conceito de componentes abstratos, propondo a definição de conectores abstratos para o suporte a operações coletivas. Todavia ainda não foram encontrados resultados práticos interessantes dessa abordagem como se pode verificar em [28] e [29].

Conforme apresentado neste capítulo, o GCM foi a ferramenta que apresentou as características mais apropriadas para o estudo proposto nesse trabalho, pois se utiliza de interfaces para proporcionar a separação entre o código funcional da aplicação e a orquestração da paralelização, permitindo, de forma simples, a definição da interação das unidades paralelas da aplicação. Ou seja, o GCM possibilita que a programação de sistemas paralelos possa ser realizada em um nível de abstração mais alto, tornando interessante o estudo dessas características no *middleware* de componentes SCS, que também usa um nível de abstração de programação mais alto para o desenvolvimento de sistemas distribuídos, todavia, sem as capacidades de paralelismo disponibilizadas pelas *Interfaces Coletivas* definidas no GCM.