INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Ginger-Video-3D

Adaptação de uma ferramenta de *rendering* gráfico/codificação vídeo para execução paralela em sistema *peer-to-peer* de partilha de ciclos

João Manuel Rebelo da Cruz Morais

Dissertação para obtenção do grau de mestre em

Engenharia Informática e de Computadores

Presidente:	Prof. José Carlos Martins Delgado
Orientador:	Prof. Luís Manuel Antunes Veiga
Co-orientador:	Prof. Paulo Jorge Pires Ferreira
Vogal:	Prof. João António Madeiras Pereira

Outubro 2009

Agradecimentos

Ao professor Luís Veiga, pela sua pronta ajuda e pelo seu espírito prático.

Aos meus pais e irmãos pelo apoio incondicional.

À Marta, Zé, Patrícia, Joana Silva, Francisco Reis, Joana Abreu e Pedro Tomé simplesmente por estarem lá.

Resumo

A Grid é um conceito que habitualmente está associado a redes de computadores construídas com propósitos claros, ou para resolver problemas de complexidade elevada ou para outros tipos de estudos nos quais a heterogeneidade e dispersão geográfica dos intervenientes são factores cruciais. Neste contexto e dado o custo associado à montagem e manutenção de uma rede desta natureza, um utilizador que queria tirar uso de processamento distribuído numa aplicação de requisitos elevados, encontra-se sem soluções. Mesmo que o acesso a estes recursos fosse aberto mantém-se a dificuldade em ter de conhecer os programas em detalhe e saber como fazer a composição das tarefas.

No Ginger este problema é abordado com uma ideia diferente. O utilizador não precisa de modificar a sua aplicação, apenas precisa de conhecer como são compostos os formatos de entrada e saída e explorar as fronteiras de independência dos mesmos para construir tarefas de menor dimensão passíveis de serem distribuídas e executadas por aplicações idênticas em máquinas com ciclos livres.

No seguimento da investigação efectuada, é proposta uma arquitectura sobre a qual a aplicação é transparentemente adaptada com base nos formatos de entrada e saída, fazendo-se uma primeira análise do conteúdo para construção de sub-tarefas e de seguida o envio para nós remotos na rede. De forma inversa as respostas são analisadas e reagregadas com base no formato de saída.

O trabalho foi testado com uma aplicação de transcodificação de vídeo e outra de processamento raytracing tendo os resultados sido especialmente positivos nesta última, cujo grande motivo foi o facto de se tratar de uma aplicação *cpu-intensive*, cujas entradas/saídas tem uma dimensão irrelevante quando contrastado com a transcodificação de vídeo.

Palavras-chave: grid, distribuição de carga, partilha de ciclos, análise de formatos

Abstract

Grid computing is a concept usually associated with institution-driven networks assembled with a clear purpose, namely to address complex calculation problems or when heterogeneity and geographical dispersion of the participants is a key factor. In this context, and given the cost of setting up such an infrastructure, a regular home user willing to take advantage of distributed processing is left without a viable option. Even if Grid access was open to the general public, a home user would not be able to express task decomposition without clearly understanding the program internals.

In Ginger, this issue is addressed in a different manner. Users are not required to modify an application they already use, needing instead to access an available format description of the application input/output, in order to decompose a job in smaller tasks that may be distributed and executed in cycle-sharing machines.

In the context of this research, an architecture is proposed on which an application can be transparently adapted based solely on input and output data formats. A first phase analyses content and builds sub-tasks that are deployed over the network. In converse fashion, results are analyzed and aggregated according to the output format.

The work was tested with a video transcoding application as well as a ray-tracing renderer, with positive results, especially in the last one, since it is a CPU-intensive application and its I/O has a smaller data load when compared with video compression.

Keywords: grid computing, load-balancing, cycle-sharing, video encoding, video format analysis

Índice

Índice de figuras

1. Introdução

A computação distribuída ou em *Grid* envolve um grande número de sistemas e recursos heterogéneos, dispersos geograficamente e sobre o domínio de diversas instituições. É o objectivo de uma infra-estrutura *Grid* abstrair este conjunto de entidades como um único recurso virtual e dinâmico, resolvendo ao mesmo tempo problemas como segurança, gestão de recursos e dados e prestação de serviços de localização. [1]

A observação que um grande número de computadores desktop na Internet estão em geral subaproveitados face às suas capacidades, gerou nos últimos anos um grande interesse no aproveitamento destes recursos livres, e tornou-se o veículo ideal para o aparecimento de várias infra-estruturas *Grid* para realização de computação distribuída cujo exemplo mais conhecido e de maior sucesso é o SETI@home [2].

1.1 Motivação

Apesar da Internet ter motivado o aparecimento de vários projectos de computação distribuída, as infra-estruturas *Grid* que os suportam são normalmente exploradas unicamente por projectos científicos ou em ambientes empresariais, impossibilitando o utilizador comum de aceder a estes recursos inutilizados, com vista a melhorar o desempenho da sua aplicação. Projectos populares de partilha de ciclos como o Folding@Home contornam a questão do porquê contribuir por terem como propósitos causas colectivas que resultam em benefício para a humanidade. Outros, como o projecto Plura, recompensam monetariamente os seus utilizadores consoante o tempo de execução dispendido. Em todo o caso o utilizador está restringido a contribuir em prol de terceiros.

1.2 Limitações dos sistemas existentes

Ainda assim, algumas soluções têm sido propostas para colmatar o problema do acesso aberto mas controlado aos recursos numa *Grid*, como o OurGrid [3] e o Integrade [4] mas ambas tiveram um sucesso muito reduzido por se tratarem de soluções incompletas e com propósitos pouco interessantes do ponto de vista do utilizador comum.

1.3 Solução proposta

O GINGER [5] tem como objectivo criar uma infra-estrutura *Grid* onde a posição de altruísta dos utilizadores na rede é substituída por uma posição mutualista, onde todos se podem ajudar a si próprios ao mesmo tempo que ajudam outros, dando acesso a tecnologias de computação *Grid* a qualquer pessoa ou comunidade interessada.

Com o crescimento do acesso residencial a banda larga surgiram novas oportunidades de partilhar informação na Internet, sendo uma das mais proeminentes a partilha de vídeos *home-made*. O YouTube é o melhor exemplo do crescimento desta área, contando com cerca de 10% (e a crescer) de todo o tráfego realizado na Internet [6]. Apesar de ser uma tarefa cada vez mais simples, as restrições actuais de largura de banda na Internet exigem que se faça compressão ou transcodificação de dados, operações demoradas que tem potencial de serem "aceleradas" através da divisão da carga computacional por vários recursos numa *Grid*.

1.3 Objectivos e Contribuição

O objectivo do Ginger-Video-3D é de implementar as ideias apresentadas no Ginger [5] e aplicá-las em concreto a aplicações de processamento multimédia, mas com um requisito primordial – as aplicações não podem ser alteradas. Deste modo o esforço principal será realizado em encontrar formas de dividir as entradas destes programas em múltiplas partições semanticamente válidas, seguido da composição de unidades de trabalho que possam ser submetidas à Grid e executadas de forma independentemente em várias máquinas. Actuando unicamente ao nível das entradas e saídas, é possível adaptar de forma relativamente simples, programas que

convencionalmente correm sequencialmente de uma forma relativamente simples, já que não há necessidade de conhecer o seu funcionamento.

Outro ponto que será alvo deste trabalho passará pela pesquisa de métodos de adaptação de aplicações já existentes a um ambiente de execução distribuída.

Cumpridos estes objectivos podemos assegurar que programas que convencionalmente correm de forma sequencial, podem estar transparentemente através do Ginger a tirar partido de computação distribuída com consequentes melhoramentos ao nível do desempenho e tempos de resposta, perceptíveis do ponto de vista do utilizador.

2. Trabalho Relacionado

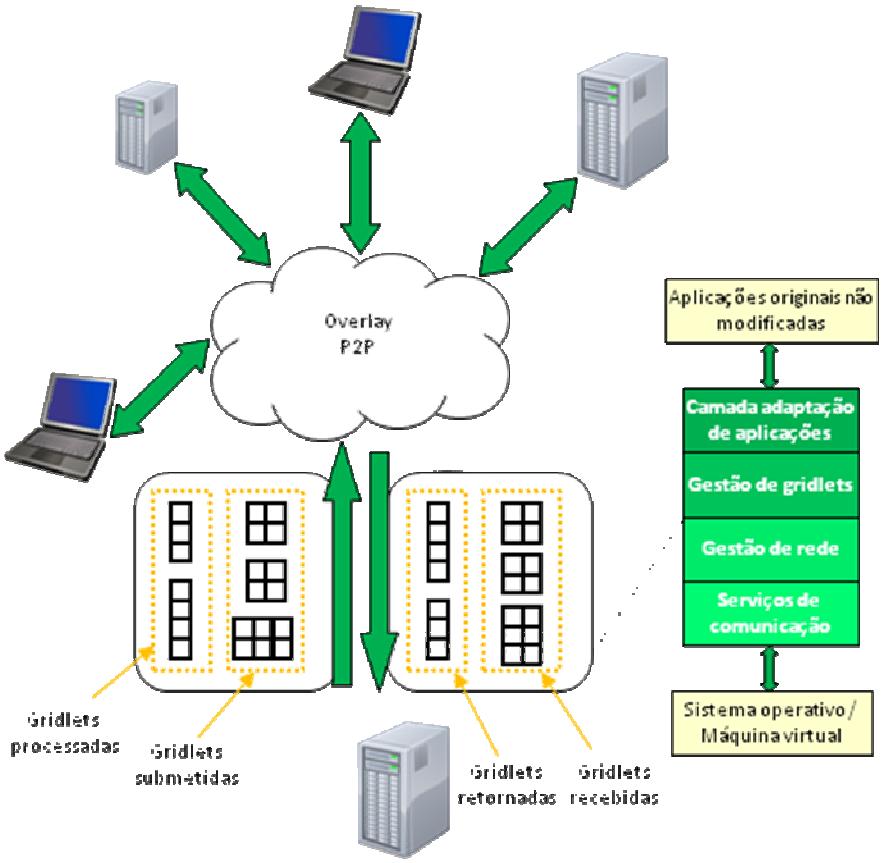


Figura 1 -Vista geral do GINGER

A figura 1 apresenta uma visão geral da arquitectura do Ginger (detalhada em [5] e extendida na secção de arquitectura proposta) e a sua composição em camadas. Apesar do Ginger apresentar uma implementação mais genérica para a adaptação de aplicações, não é um conceito novo e desta fora é necessário saber que trabalho nas áreas que este trabalho abrange. De acordo com os objectivos descritos na secção anterior, destacaram-se as seguintes áreas de interesse para o Ginger-Video-3D: desktop *grids* de partilha de ciclos, processamento multimédia distribuído e adaptação de aplicações.

2.1. Infra-estruturas e Arquitecturas Grid de Partilha de Ciclos

A computação distribuída quando aplicada à utilização de recursos em estado passivo e partilhados por diversas fontes heterogéneas, levanta uma série de questões interessantes:

- ***Incentivos***: que tipos de incentivos são necessários para motivar os utilizadores a ceder os seus recursos (nomeadamente ciclos de relógio)?
- ***Descoberta de recursos e escalonamento eficiente***: Como encontrar recursos na rede num ambiente dinâmico e de livre acesso? Como seleccionar os melhores ou mais apropriados?
- ***Reputação****:* Que clientes demonstram um comportamento confiável que irão devolver uma resposta correcta? Quais responderão primeiro? Qual é a garantia que a "pergunta" irá eventualmente obter uma "resposta"?
- ***Justiça:*** Como garantir uma partilha de recursos justa, entre os vários utilizadores da *grid*, de forma a impedir monopólios no acesso a estes?
- ***Segurança****:* Como é que um cliente e a rede como um todo se podem defender contra ataques de *DDoS*, pedidos com código malicioso ou inútil e ainda respostas manipuladas com valores incorrectos?

De seguida resumem-se algumas infra-estruturas e arquitecturas Grid, que visam resolver estes e outros desafios que a computação Grid levanta.

Cluster Computing on the Fly [7] descreve uma arquitectura peer-to-peer de partilha de ciclos, propondo soluções a problemas de computação intensiva idênticos aos enfrentados pelo projecto SETI@home e ainda a aplicações onde os recursos são usados como Point-of-Presence, útil por exemplo para a monitorização e análise de tráfego numa rede. Aplicações de grande carga computacional, são escalonadas através de um processo designado de Wave Scheduler que organiza os recursos por zona geográfica com o objectivo de aproveitar os períodos nocturnos de menor actividade. Implementa ainda um sistema inteligente e eficaz de detecção da validade dos clientes através de um sistema de pergunta/resposta. No entanto o projecto

parece ter-se ficado pela promessa, não tendo ainda disponibilizado um middleware sobre o qual se possa construir novas aplicações Grid.

O BOINC [2] é um middleware de computação voluntária de aplicações não comerciais tornado popular por conhecidas aplicações como SETI@home e o Folding@home. Nestes sistemas existe um coordenador central que envia as unidades de trabalho para os clientes que actuam de forma altruísta não recebendo qualquer recompensa pelo seu trabalho, para além do registo da sua contribuição. Para além da motivação óbvia para o bem comum que poderá provir destes projectos, o BOINC oferece uma interface didáctica que permite visualizar alguns dos passos da execução do processo, fazendo também um esforço por gerar interesse através da competição entre grupos de utilizadores, contabilizando-a através de um elaborado sistema de créditos.

O modelo de programação está bem delineado no entanto o acesso à infra-estrutura e aos recursos obriga ao desenvolvimento de clientes e servidores BOINC o que torna esta solução inviável para o utilizador comum. Para além disso, as aplicações têm de ter um cariz científico e necessitam de passar um processo de controlo e certificação, o que limita bastante o âmbito das aplicações que podem ser executadas na rede.

Assim, no BOINC as unidades computacionais acabam por ser corridas directamente sobre o sistema operativo sem qualquer solução de sandboxing já que se assume que as entidades são confiáveis. Para prevenir problemas de man-in-the-middle o código é assinado com a chave privada da entidade responsável pela unidade de trabalho.

O envio de resultados maliciosos é atenuado através do envio de diversas cópias da mesma unidade de trabalho para diversos clientes, sendo feita uma posterior votação para aceitar a resposta com maior probabilidade de estar correcta.

O Condor [8] realiza uma parte importante das funcionalidades de uma Grid, descrevendo um mecanismo de gestão e partilha de recursos em ambientes sobre o mesmo domínio administrativo, capturando ciclos inutilizados em estações de trabalho comuns para a realização de tarefas que requerem não só um grande poder de processamento mas também uma grande quantidade de informação.

É na sua essência um sofisticado sistema de escalonamento de processos garantindo um acesso justo e equilibrado aos recursos disponíveis garantindo que o uso da máquina não é afectado, quando esta for requisitada pelo utilizador da mesma.

Globus [9] é um toolkit organizado sobre a forma de um conjunto de workflows desenvolvidos sobre uma camada SOA que por sua vez assenta e pode utilizar um conjunto de ferramentas que simplificam o escalonamento, partilha e monitorização de recursos distribuídos por uma qualquer rede, permitindo modelar e construir uma infra-estrutura Grid à medida das necessidades. A abordagem do Globus é de estruturar os componentes duma Grid em camadas numa aproximação bottom-up, começando por funcionalidades de baixo nível como comunicação, autenticação, serviços de informação e acesso a dados, desenvolvendo depois vários serviços de alto nível sobre estas. O objectivo do Globus é de revelar uma infra-estrutura que abstraia a Grid como um metacomputador virtual em permanente mudança, permitindo a fácil criação de novos serviços ou aplicações de alto desempenho adaptadas a este ambiente.

O Globus têm como característica dominante (e mais valia) ser o projecto que implementa o maior número de standards do OGF (Open Grid Forum), um consórcio de diversas entidades que têm por objectivo normalizar os vários componentes que habitualmente se encontram em infra-estruturas de computação em Grid.

Integrade [4] é um middleware que utiliza uma arquitectura inter-cluster apoiada em gestores clusters que comunicam entre si e que por sua vez controlam um conjunto de máquinas que cedem recursos ao overlay. Este tipo de arquitectura foi desenvolvida para suportar aplicações MPI ou de bag-of-tasks (com/sem comunicação inter-processo) e também para dar um maior controlo sobre as topologias de rede onde estas podem ser executadas. Neste sistema os clientes estão ligados de forma centralizada ao gestor do cluster e desconhecem da existência uns dos outros, logo a submissão é feita através deste que fará o escalonamento e reencaminhamento do binário pelos restantes clientes.

Outra aspecto interessante do Integrade é o facto da gestão de recursos ser baseada em análise de padrões de utilização, em 2 fases de aprendizagem de forma a melhor

perceber como e quando estes recursos podem ser utilizados. Com vista a recuperar de processos incompletos existe um sistema de checkpoints do estado da aplicação que permite a sua posterior continuação a partir desse ponto.

O MOSIX [10] é um sistema de gestão de clusters Linux que permite um conjunto de nós x86 actuar como um único computador paralelo. Os utilizadores podem correr código sequencial ou paralelo, ficando a cargo do MOSIX e encontrar e alocar os recursos necessários à sua execução. Para além da gestão automática dos recursos o Mosix oferece migração transparente de processos (criados por fork) entre nós e um ambiente seguro de execução que previne que processos remotos de aceder a recursos protegidos, através da captura de chamadas ao sistema.

O Ourgrid [3] é um toolkit que tal como o Integrade federa clientes em diferentes clusters e apresenta-se como uma solução completa para correr aplicações bag of tasks em grids. O sistema pode ser visualizado como uma arquitectura peer-to-peer entre peers OurGrid que por sua vez servem de servidor central para os clientes e recursos.

Este toolkit fornece ao utilizador ferramentas de abstração relativas à composição da grid, escalonamento das aplicações pelos diferentes nós, gestão de recursos através de um mecanismo de rede de favores (recompensa clientes que mais contribuem), e segurança na execução das tarefas através de virtualização (sandboxing) de processos. O objectivo do OurGrid é possibilitar a qualquer utilizador uma forma acessível de submeter a sua aplicação num sistema distribuído de partilha de ciclos, no entanto presentemente o sistema não tem completo o componente de virtualização, denominado SWAN, logo só pode ser utilizados junto de fontes fidedignas o que perde um pouco a sua utilidade prática.

Butt et al. [11] descreve um protótipo de um sistema de partilha de ciclos que aproveita tecnologias peer-to-peer já existentes como o Pastry para permitir uma gestão eficiente da entrada e saída de clientes e também a fácil descoberta de recursos, deixando de haver a necessidade de uma entidade central. Neste projecto as aplicações correm sobre a Java Virtual Machine simplificando assim questões como a portabilidade de programas e segurança na execução de código remoto.

Um dos objectivos principais é garantir que o sistema é justo para todos os participantes. Assim é proposto um elaborado sistema de créditos e ainda um mecanismo de detecção de progresso nas tarefas para compensar da forma mais correcta possível o utilizador que cedeu os ciclos de processamento, mesmo que não tenha conseguido por algum motivo terminar o processamento da unidade de trabalho que lhe foi confiada, partindo naturalmente do princípio que todas os utilizadores são confiáveis.

Legion [12] à semelhança do Globus procura construir uma camada de abstracção à volta da complexidade e diversidade que são características naturais sistema distribuído de partilha de ciclos. Alguns dos problemas aos quais o Legion dá maior relevância incluem escalabilidade, transparência no acesso a recursos, tolerância a faltas, segurança para os utilizadores e para os donos dos recursos, autonomia dos domínios, gestão de recursos, extensibilidade e facilidade de desenvolvimento de novas aplicações que explorem elevados graus de paralelismo.

Uma característica interessante é que do ponto de vista do Legion todos os componentes (utilizadores, dados, aplicações, políticas, etc.) são objectos logo têm acesso imediato as vantagens do paradigma de objectos, como reutilização, contenção de falhas e redução de complexidade. Outro aspecto é que o Legion deixa à consideração do programador que tipo de políticas de segurança, replicação e alocação de recursos serão utilizadas no sistema, permitindo decidir qual o melhor trade-off dado o processo em questão.

O Sun Grid Engine [13] é um software open-source de gestão de recursos que faz a associação de requisitos de software ou hardware a um conjunto de recursos disponíveis e possivelmente heterogéneos. É usado tipicamente em clusters e é responsável por aceitar, escalonar, despachar e gerir a execução remota de um ou mais processos distribuídos.

Baseia-se numa arquitectura cliente-servidor composta por um servidor central que supervisiona e verifica que as políticas de utilização definidas pelos administradores da grid estão a ser cumpridas e que é também responsável pela interacção com o

utilizador que submete as unidades de trabalho. Não é uma ferramenta de uso pessoal, sendo mais orientada a empresas ou a meios académicos.

Em seguida faz-se um resumo das características mais importantes de alguns dos projectos referidos acima e de maior utilização na actualidade:

	BOINC	**Globus**	**OurGrid**	**Sun Grid Engine**
Tipo de plataforma	Middleware	Toolkit	Middleware	Monitor
Arquitectura	Centralizada em cada projecto	Configurável	Federada	Centralizada
Âmbito	Projectos científicos	Empresarial ou Académico	Qualquer	Empresarial
Acesso	Limitado - necessidade de certificação Gratuito	Necessidade de montar rede Gratuito	Aberto e gratuito	Necessidade de montar rede Versão Gratuita
Modelo de programação	Bag-of-tasks	Bag-of-tasks	Bag-of-tasks ou MPI	Não definido
Funcionalidades	Distribuição de carga Sistema elaborado de créditos Validação fiável de resultados Interfaces utilizador	Segurança Gestão de recursos e de dados Protocolos de comunicação Detecção de falhas Portabilidade de processos	Descoberta e gestão de recursos Escalonamento de processos Virtualização de aplicações Justiça através de rede de favores	Distribuição de carga Escalonamento de processos Gestão de quotas de acesso Monitorização de rede e de recursos

2.2. Processamento Multimédia Distribuído e Paralelo

Esta área é de grande interesse para este trabalho uma vez que os conteúdos multimédia são de uma forma geral de grande dimensão e necessitam de elaboradas técnicas e algoritmos de compressão para se tornarem viáveis para utilização no dia-a-dia.

Esta redução de tamanho, é conseguida à custa de uma enorme intradependência da informação multimédia o que dificulta a sua segmentação em blocos básicos passíveis de serem processados em paralelo. Esta intradependência é conseguida codificando apenas diferenças entre imagens ou amostras de som relativas a um ponto independente que contenha um segmento da informação completo e independente dos restantes. No caso do vídeo, como de forma geral as imagens apresentam

reduzidas diferenças entre elas, excepto quando existem transições abruptas de cena, é possível obter uma grande redução na quantidade de informação a transmitir.

De seguida apresentam-se alguns estudos e soluções relativos a esta área.

Yang et al. [14] apresenta uma arquitectura baseada em clusters Linux interligados pelo *toolkit* Globus e monitorizado pelo Sun Grid Engine, para o processamento em paralelo de uma aplicação de *ray tracing*. São ainda feitos testes que demonstram que efectivamente a distribuição da carga por vários computadores acelera a obtenção de resultados e são ainda feitos estudos para identificar qual é a melhor relação entre o número de processadores envolvidos e o tempo final do processo.

Akramullah et al. [15] descreve uma implementação de um codificador de vídeo MPEG-2 em tempo real, usando uma aproximação de paralelismo de dados, explorada ao nível de cada frame individualmente e que corre sobre o Intel Paragon XP/S, um multi-processador com memória distribuída e arquitectura MIMD com MPI.

O funcionamento geral do algoritmo consiste em dividir cada frame em vários macroblocos que são delegados para cada processador e que comunica com os restantes para fazer uma correcta compensação de movimento nas fronteiras do seu macrobloco.

São apresentados também alguns mecanismos de controlo do débito de cada canal e de adaptação dinâmica das matrizes de quantização.

Ewerth et al. [16] propõe a aplicação de computação *Grid* e tecnologia de *WebServices* para análise de conteúdo multimédia nomeadamente detecção de cortes, texto, faces, movimentos, etc.

Neste caso o objectivo do trabalho é a detecção de transições num vídeo, um processo moroso que consiste em comparar diferenças de histogramas numa sequência de imagens. O algoritmo desenvolvido é composto por um serviço de gestão do fluxo vídeo de entrada e do número de tarefas a serem submetidas à *Grid* (com base nas dimensões do vídeo e débito do vídeo), um outro que faz o redireccionamento dos fluxos de cada segmento de vídeo para nós com recursos disponíveis e um terceiro que procede então à execução do algoritmo de detecção de cortes.

Os resultados experimentais apontam para desempenhos notáveis face ao processamento num só nó.

Kola et al. [17] propõe soluções para o problema dos sistemas de partilha de ciclos actuais não escalarem de forma satisfatória em aplicações *data-intensive*, como acontece no processamento de vídeo. A principal diferença para as soluções existentes é que neste sistema a transferência e processamento de dados são considerados como actividades diferentes, apesar de um surgir em seguimento do outro.

Uma rede de dependências é construída com este propósito garantindo não só a ordem pela qual as tarefas se desenrolam mas também permite determinar de forma mais precisa e eficiente o aparecimento de falhas no sistema e seu posterior tratamento.

São ainda sugeridos diferentes modelos de transferência de dados até aos nós computacionais e ainda um *case study* específico do armazenamento de 500 Terabytes de vídeo miniDV.

Parallel Horus [18] é uma biblioteca de programação para clusters que permite que programadores implementem aplicações multimédia paralelizadas através da escrita de programas totalmente sequenciais. Um conjunto de tipos de dados multimédia e operações primitivas de processamento e análise de imagens (Image Algebra) suportam a paralelização recorrendo a MPI para troca de mensagens. Com vista a optimizar o desempenho do sistema é feita uma subsequente análise em tempo de execução com vista a optimizar o programa e reduzir a comunicação entre processos a um mínimo (*lazy parallelization*).

Esta iniciativa teve origem na necessidade de analisar e detectar padrões suspeitos em imagens de vídeos de câmaras de vigilância, portanto os requisitos de escalabilidade e desempenho são de crucial importância principalmente tendo em conta que o de volume de informação está na ordem de dezenas de petabytes.

Outros trabalhos referentes a processamento distribuído ou paralelo (em multiprocessadores) de informação multimédia estão em Li et al. [19] que demonstra várias estratégias de codificação de vídeo em paralelo num sistema multiprocessador,

Fellow et al. [20] que dá uma explicação extensa sobre sistemas distribuídos multimédia e Little et al. [21] que apresenta algumas considerações sobre a composição e comunicação de objectos multimédia em diversos tipos de redes.

2.3. Adaptação de Aplicações

A adaptação de aplicações é parte integrante do Ginger que tem como um dos seus objectivos a adaptação automática de programas não modificados para serem executados em sistema de partilha de ciclos.

Huang et al. [23] descreve um processo de conversão semi-automática de programas C para serem executados em serviços Triana. Apresenta duas ferramentas para o efeito: o JACAW que é uma ferramenta de encapsulamento de código baseada na Java Native Interface e que gera uma interface Java para qualquer rotina C; o MEDLI assiste o utilizador na descrição dos mapeamentos entre os tipos do Triana e os tipos C envolvidos na chamada de uma rotina em particular.

Sneed [26,27] identifica formas e técnicas de encapsulamento e descreve um processo de reengenharia de interfaces de um programa legacy escrito em Cobol, que extrai toda a informação necessária à execução de cada módulo do programa. Desta forma cada módulo (mais o wrapper) torna-se independente do programa e pode ser utilizado por outros módulos, locais ou remotos.

De Lara et al. [28] sugere extensões aos modelos tradicionais de replicação (optimistas e pessimistas), nos quais as réplicas além de um determinado número de versão possuem também um nível de fidelidade relativo à versão original. No caso de uma imagem, esta métrica pode ser por exemplo o factor de compressão ou a resolução da mesma.

O cenário alvo é a edição de documentos em dispositivos móveis ligados a um repositório central. De modo a aumentar a flexibilidade deste sistema, estes documentos são decompostos em variados componentes multimédia (texto, páginas, imagens, gráficos, etc.), sendo feita uma adaptação posterior conforme a largura de banda disponível e enviadas versões de baixa fidelidade e dimensão mais reduzida para o dispositivo.

Além da redução de tamanho dos componentes, o aumento da granularidade permite adoptar uma postura lazy e só transmitir partes do documento quando estas são realmente necessárias além de reduzir possíveis conflitos entre utilizadores a editar o documento de forma concorrente.

O artigo apresenta ainda uma implementação (CoFi – consistency and fidelity) aplicada sobre o Outlook e o Powerpoint através de extensões aos mesmos e assente sobre um proxy que gere de forma transparente a comunicação com o sistema replicado.

No seguimento do artigo anterior, De Lara et al. [29] entra um pouco mais detalhe nos mecanismos de adaptação a aplicações, nomeadamente através do uso de API's exportadas por estas.

A abordagem gira em torno de um mecanismo de adaptação iterativa a aplicações que suporta adaptação ao nível de dados através de técnicas de decomposição e transformação de componentes (como vimos anteriormente) e também ao nível de controlo do comportamento do programa através da definição de políticas – por exemplo, retornar controlo ao utilizador quando todo o texto for carregado. É importante referir ainda, que neste sistema não existe qualquer alteração às aplicações.

O artigo descreve ainda um sistema de adaptação iterativa a aplicações de produtividade em Linux e Windows, denominada Pupeteer. Este sistema comporta-se de forma semelhante ao CoFi [28], mas além da utilização de "proxies" no acesso aos repositórios de dados fornece ainda uma série de módulos (núcleo, "drivers", transcodificadores e políticas) que uniformizam as diferentes API's, tipos de documentos e plataformas onde as aplicações podem ser executadas. Com esta simplificação a implementação de políticas torna-se bastante mais cómoda já que uma política *a priori* funcionará de forma igual para qualquer uma das aplicações.

Um resultado particularmente interessante é que além das API's do Office (COM) e OpenOffice (Corba) serem bastante diferentes, nota-se que a adaptação de aplicações nunca esteve no pensamento dos engenheiros que a desenharam, levantando grandes desafios e mesmo impossibilidades para o Pupetter.

3. Análise

As soluções apresentadas no ponto 2.1. propõem middlewares e ferramentas para o desenvolvimento de aplicações em ambientes distribuídos, grande número destas dando especial ênfase à utilização ciclos livres de computadores comuns. Algumas delas apresentam elaboradas medidas de escalonamento para maximizar a utilização destes recursos, outras propõem complexos esquemas de créditos para aferir a justiça no acesso aos recursos e outras tem ainda como objectivo fornecer ferramentas configuráveis com as quais se possa construir aplicações que explorem computação Grid ou outras ferramentas mais elaboradas.

No entanto nenhuma se preocupa em suportar transparentemente aplicações já existentes, explorando técnicas de adaptação das mesmas a estas infra-estruturas, com vista a acelerar o seu tempo de resposta trazendo as potencialidades da computação Grid até ao utilizador comum. Além desta limitação a ausência de ferramentas que simplifiquem o processo de descrição e decomposição do problema de partida em sub-problemas e subsequente classificação e escalonamento de cada um destes, levantam barreiras altas de entrada para muitas aplicações de uso comum.

Acima de tudo estes projectos não foram ainda capazes de fornecer uma infra-estrutura descentralizada, suficientemente genérica e de acesso verdadeiramente livre, que permita a execução distribuída de aplicações desktop num ambiente heterogéneo como a Internet, juntando o sucesso de aplicações P2P de partilha de ficheiros como o BitTorrent e aplicações de partilha de ciclos como o SETI@home.

4. Arquitectura/soluções proposta

Na figura 1 (página 4) está ilustrado de forma bastante generalizada o modo pelo qual uma aplicação é adaptada para execução sobre o Ginger. Entrando em maior detalhe, desde a invocação do comando e a obtenção do ficheiro com o resultado a informação de entrada passa por uma sequência de blocos lógicos de processamento com o objectivo de identificar aplicações e formatos para de seguida construir tarefas de menor dimensão e enviar para clientes que fazem o processamento da aplicação. Esta sequência encontra-se ilustrada na figura 2.

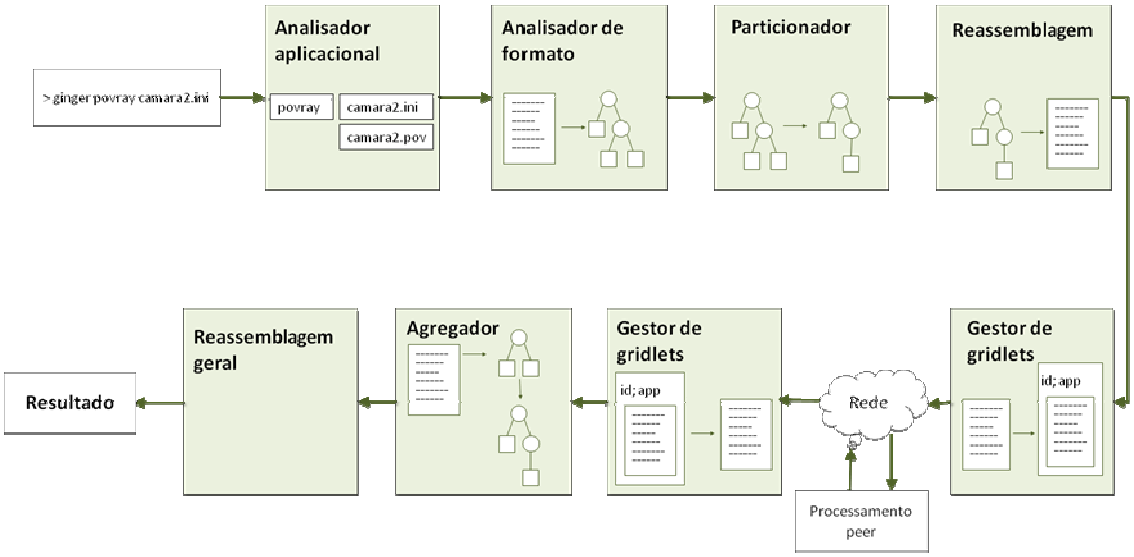


Figura 2 - Vista detalhada da operação do GINGER

Como se pode observar, o início do processo começa com a invocação da aplicação pelo utilizador. O 1º passo naturalmente consiste em identificar qual o programa a ser invocado, qual é o ficheiro de entrada e saída e quais são os argumentos adicionais a serem passados. Todas estas propriedades e outras que veremos mais à frente estão contidas num descritor de aplicação.

Na etapa seguinte já se conhecendo os formatos de entrada e saída, que podem não ser iguais, é obtido um descritor para o formato de entrada que contem uma gramática que valida o ficheiro de entrada, bem como instruções para divisão em tarefas e junção de resultados. Com base nesta gramática o analisador de formatos "varre" o ficheiro e identifica vários os blocos semânticos do ficheiro, construindo em

memória uma árvore representativa do mesmo para que mais tarde seja fácil navegar pela estrutura do ficheiro e fazer as manipulações requeridas para a correcta divisão das tarefas. Na fase seguinte é preciso saber quantas tarefas serão produzidas e qual a sua dimensão. O descritor da aplicação, sendo conhecedor das complexidades do problema, sugere um vector que atribui um peso às diferentes componentes de execução (CPU, I/O e largura de banda), vector este que é passado ao gestor de gridlets que com base no número e capacidade dos *peers* que estão disponíveis, tenta fazer a melhor atribuição possível reservando o tempo computacional necessário nos *peers* seleccionados. Conhecendo-se o número de blocos, é pedido ao descritor do formato para fazer as manipulações necessárias na árvore do ficheiro de entrada para produzir sub-árvores que representem uma porção de menor dimensão computacional do problema inicialmente proposto. Feita a divisão, existe um processo de serialização que transforma as diversas sub-árvores em ficheiros passíveis de serem executados sobre uma aplicação em tudo idêntica à original mas presente no computador remoto. Já tendo sido identificados previamente quais estes os participantes no processamento desta tarefa, cada ficheiro de trabalho é "empacotado" junto com os identificadores para este trabalho em específico e para a tarefa bem como toda a informação extraída do descritor aplicacional e que é fundamental para executar a aplicação. Este pedido é então enviado para execução no computador remoto. Para terminar este processo, os resultados são recebidos na origem, desempacotados e remetidos novamente para analisador gramatical mas que se baseia desta vez na gramática do descritor do formato de saída para construir novas sub-árvores de saída. O descritor é de novo consultado para executar as transformações de agregação das diferentes árvores com vista a construir uma árvore final de saída. Finalmente por um processo de serialização obtemos o ficheiro de resultado que é enviado ao cliente.

Vamos entrar agora em detalhe nas áreas de maior desafio que são:

1. Adaptação de aplicações
2. Análise e divisão/reagregação de ficheiros
3. Distribuição e escalonamento de tarefas

4.1. *Adaptação de aplicações*

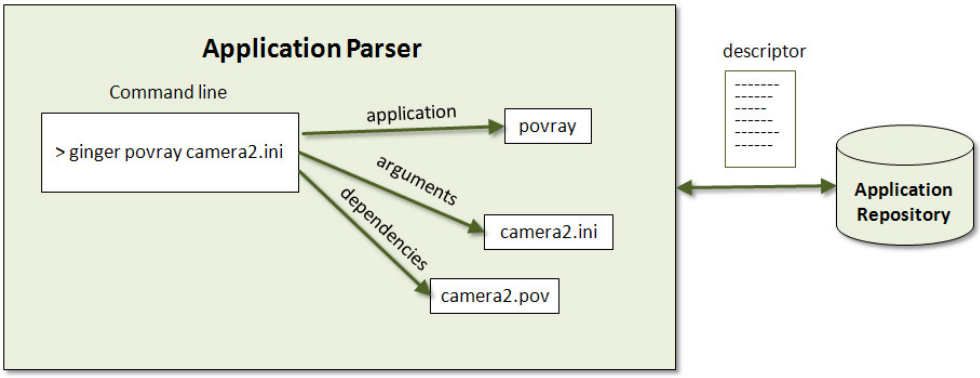


Figura 3 - Formato de descritor aplicacional

Esta área é o ponto de entrada da aplicação no Ginger. A invocação é feita através da linha de comandos:

- `ginger nome_aplicação argumentos`

O objectivo nesta fase é descobrir qual a aplicação em questão, quais os ficheiros de entrada e saída, quais os argumentos passados, bem como outras dependências que sejam necessários replicara para os outros nós que irão executar esta aplicação. Para tal o primeiro passo é consultar o repositório de descritores aplicacionais para encontrar o que corresponde a esta aplicação. O descritor é um ficheiro xml com o seguinte formato exemplificativo para a aplicação de compressão vídeo:

```
<application name="ffmpeg">
    <arguments passthrough="true">
    <argument name="inputfile" regex="-i (\w+)"/>
    <argument name="outputfile" regex="\s(\w+)$"/>
    <argument name="vcodec" regex="-vcodec (\w+)"/>
  </arguments>
  <input argument-name="inputfile">
    <extension equals="avi" format="avi"/>
  </input>
  <output argument-name="outputfile">
    <extension equals="avi" format="avi"/>
  </output>
    <costs>
    <cost select="${vcodec} = "h263p"" value="0.7"/>
    <cost select="${vcodec} = "h264"" value="0.8"/>
  </costs>
</application>
```

Figura 4 - Formato de descritor aplicacional

O formato explica para aplicação em questão, quais os argumentos de entrada e saída, como podem ser descobertos e quais os formatos respectivos. Podem conter ainda restrições à execução, como por exemplo no caso do PovRay, ter de haver um ficheiro de inicialização com o mesmo nome do ficheiro de entrada.

A título de exemplo são também incluídas as sugestões do peso aplicacional para determinada tarefa, que dê uma noção ao gestor de *gridlets* de quais os participantes mais acertados para realizar a execução da aplicação. Na determinação do vector característico para determinada entrada, o descritor pode aceder à estrutura em árvore do ficheiro de entrada sendo possível ao descritor do formato *povray*, por exemplo, descobrir quantas frames é que vão ser geradas na animação duma cena.

4.2. Análise e transformação de ficheiros

Um dos formatos que foi alvo principal deste trabalho foi o AVI um formato binário que serve de contentor de informação vídeo e áudio. Neste formato bem como em diferentes outros formatos multimédia, existe uma forte dependência entre blocos num ficheiro e portanto fazer uma divisão cega em partições de tamanho fixo não é possível. Assim, é preciso identificar os pontos de independência nos quais se pode efectuar uma divisão coerente – no AVI são chamadas *keyframes*, frames que não dependem de nenhuma frame nem para a frente nem para trás. Desta forma é possível remover todas as frames a partir da próxima *keyframe*, bem como outra informação associada tendo a garantia que temos um vídeo na mesma sintacticamente correcto mas com uma duração menor. É uma operação algo delicada, porque qualquer bit errado e o ficheiro pode ficar inválido.

Indo por passos, a primeira etapa passar por analisar o ficheiro de entrada e construir uma representação em memória do mesmo através da qual seja fácil identificar os blocos mais importantes de um ficheiro e também que seja facilmente manipulável para ser possível construir novas representações, de menor dimensão computacional.

Análise de ficheiros

Por consulta do repositório de formatos é seleccionado o descritor de formato para este ficheiro. Com base na gramática contida no descritor do formato é feita uma análise sintática do documento até que a gramática seja aceite ou falhe, indiciando

neste caso um ficheiro que não corresponde ao formato. À medida que o ficheiro vai sendo processado os *tokens* lidos são acrescentados a uma representação em memória na forma de uma árvore. Como pode ser visto na figura 5 os *tokens* são meros apontadores para blocos do ficheiro, cuja representação em memória iremos denominar daqui para a frente como documento.

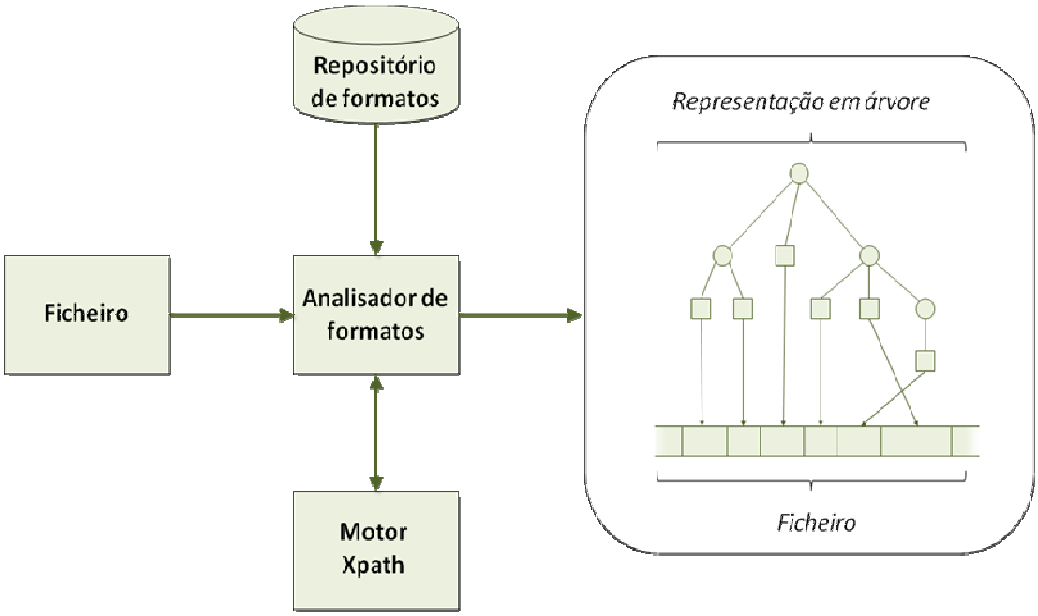


Figura 5 - Processo de análise de ficheiro e construção de representação em árvore

A composição de um *token* extraído de um ficheiro é a seguinte:

Ficheiro de Origem	Deslocamento no ficheiro	Deslocamento actual	Dimensão do bloco	Tipo de dados

A razão por haver um deslocamento no ficheiro e um deslocamento actual, é que sempre que há manipulações na árvore o deslocamento de um bloco no documento poderá ter mudado, enquanto o deslocamento no ficheiro naturalmente mantêm-se. É possível também criar novos *tokens*, como iremos ver na fase de transformação de documentos mais à frente. Estes *tokens* têm a seguinte composição:

Deslocamento actual	Valor	Tipo de dados

Nestes *tokens*, o valor é guardado como uma cadeira de caracteres e a sua dimensão é determinada por consulta directa com o validador de tipo de dados. Além desta

informação, todos os nós da nossa árvore o que inclui tanto os *tokens* como os elementos é caracterizado por um nome, elemento pai e documento associado.

Como de forma geral os *tokens* apontam para blocos de dimensão bastante superior, a representação com base nesta estrutura é bastante leve em memória. Mesmo que se coloque o caso dos blocos do ficheiro serem de dimensão reduzida maximizando os impactos em memória, serão raros os casos em que todos os campos de um formato serão necessários para consulta e como tal necessitem de estar representados em memória. Nestes casos é possível agrupar múltiplos blocos num só *token* a apontar para o primeiro bloco e com dimensão igual ao somatório dos diferentes blocos, ou simplesmente saltar sobre todos estes blocos e continuar a análise. De notar ainda que é guardado o ficheiro de origem nos *tokens* apenas nos casos em que o nó foi importado doutro documento, como iremos ver adiante. Em todos os restantes o ficheiro de origem está presente no nó de documento, que é um antecessor de todos os nós na nossa representação.

Na figura 5, aparece ainda referenciado um motor XPath. Esta ferramenta, chamada Saxon e que foi desenvolvida por terceiros [31], é indispensável na análise do ficheiro uma vez que permite facilmente fazer consultas sobre o nosso documento, desde saber qual o valor de determinado *token* como fazer elaboradas relações entre diferentes *tokens* na árvore. Esta característica é fundamental na análise de formatos onde existem campos cuja leitura depende do valor de campos que lhe antecedem, entre outros. Um exemplo prático é o formato RIFF (que inclui o AVI) onde o tamanho de um bloco é determinado pelo primeiro campo *size* que o antecede, mas tal acontece de uma forma ou de outra em todos os formatos onde possam existir blocos de dimensão variável. Assim foi desenvolvida uma interface com o Saxon, que adapta o nosso modelo de objectos para suportar pesquisas sobre a linguagem XPath que habitualmente serve como método de pesquisa para árvores XML e que sendo altamente reconhecida e aceite pela comunidade surgiu como o método ideal fazer pesquisas e seleccionar um mais *tokens* e que é usada tanto nesta fase de análise como também na fase de transformação como iremos ver adiante. De referir ainda que todos os nós da árvore possuem atributos de nome, deslocamento, dimensão e valor que são indispensáveis para utilizar algumas funções XPath.

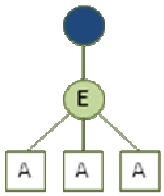
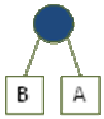
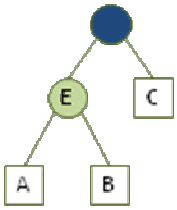
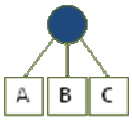
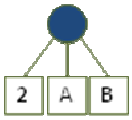
A nossa gramática é inteiramente descrita em XML e adopta em grande parte as regras descritas na linguagem de validação de documentos XML, Relax NG. É composta por 7 regras de base e outras 2 que permitem a reutilização de regras ao longo da gramática. As regras são as seguintes:

Data	Lê um *token* com base num tipo de dados e adiciona o mesmo ao elemento actual da árvore que não sendo especificado é a raiz da árvore.
Element	Elemento da árvore, permite separar *tokens* em diferentes nós
Group	Agrupa *tokens*. *Tokens* tem de aparecer na ordem com que aparecem no grupo.
Interleave	Permite que *tokens* apareçam em qualquer ordem
Block	*Tokens* dentro de um bloco tem de estar contíguos. Um bloco só é aceite quando todos os *tokens* forem encontrados.
Choice	Escolhe entre 2 regras distintas. Caso a primeira escolha for aceite, a segunda não é testada. No entanto este comportamento pode ser alterado.
Sequence	Esta regra é o quantificador na nossa gramática e permite especificar quantas vezes a regra tem de ser encontrada ou que tamanho de entrada tem de ser lido. Para simplificação são fornecidos os dois quantificadores de uso mais habitual "*" e "+" que correspondem às regras **zeroOrMore** e **oneOrMore**.
Define/Ref	A construção *define* é colocada fora da gramática "principal" e engloba um conjunto de regras que pode ser referenciado em qualquer ponto da gramática através da regra *ref*. De notar que uma definição se pode referir a si própria, podendo-se desta forma criar definições recursivas.
Include	Este construtor é também definido fora da gramática e permite importar definições contidas noutros ficheiros.

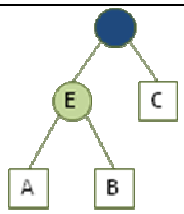
Para melhor compreender a forma de funcionamento das diferentes regras, faz-se de seguida uma breve exemplificação do seu uso na construção de gramáticas suportadas pelo analisador sintáctico:

Gramática	Entrada	Resultado	Observação
`<data type="char" equals="A"/>`	A		Leitura de *token* do tipo char => string dimensão 1.

`<element name="E">`
`<data type="char" equals="A"/>`
`</element>`
	A		*Token* colocado dentro de um elemento

<element name="E"> <oneOrMore> <data type="char" equals="A"/> </oneOrMore> </element>	AAA		Se <oneOrMore> e <element> estivessem trocados, haveria vários elementos E com apenas um *token* A
<zeroOrMore> <data type="char" equals="A"/> </zeroOrMore>	-		O *token* é opcional e logo a árvore é vazia
<oneOrMore> <choice> <data type="char" equals="A"/> <data type="char" equals="B"/> </choice> </oneOrMore>	BA		Note-se que primeiro é escolhido 2º *token*. Outra entrada válida podia ser BB
<optional> <data type="char" equals="A"/> </optional> <data type="char" equals="B"/>	B		Todos os *tokens* dentro de <optional> não tem de ser encontrados.
<interleave> <element name="E"> <data type="char" equals="A"/> <data type="char" equals="B"/> </element> <data type="char" equals="C"/> </interleave>	**ACB**		A regra <interleave> permite que os *tokens* estejam desordenados. A ordem no <element> tem de se manter! BAC não é válido.
<interleave> <block> <data type="char" equals="A"/> <data type="char" equals="B"/> </block> <data type="char" equals="C"/> </interleave>	**ABC**		A regra <block> obriga a que os *tokens* tenham de estar contíguos. Só existe mais uma entrada válida > CAB
<data name="size" type="char"/> <sequence size="/size"> <data type="char/> </sequence>	**2AB**		<sequence> continua a consumir *tokens* até atingir tamanho "size". Note-se o uso de Xpath para determinar o size.

Uma nota para a seguinte gramática:

<interleave> <element name="E"> <data type="char" equals="A"/> <data type="char" equals="B"/> </element> <data type="char" equals="C"/> </interleave>	**ACB**	

Numa pesquisa em profundidade da nossa árvore de resultado o *token* C é encontrado antes de B, apesar de aparecer depois no ficheiro, uma vez que o elemento E é construído quando A é encontrado. Para a entrada ABC ou CAB tal não acontece. Este

comportamento da nossa gramática é problemático quando pretendemos passar esta árvore de novo para um ficheiro ou quando pretendemos copiá-la para um documento diferente. Para o contornar, o nó de raiz contem uma lista de todos os *tokens* ordenados pela sua posição real no ficheiro. Isto facilita a subsequente serialização do documento para ficheiro porque basta percorrer a lista e fazer a escrita de acordo com o tipo de dados de cada *token*. No entanto dificulta a cópia da árvore ou de segmentos da mesma na fase de transformação, porque após a cópia é necessário garantir que cada nó está na posição correcta no documento de destino, de acordo com a ordem pela qual aparecia no documento de origem.

Abre-se também uma ressalva relativamente ao determinismo da gramática. No caso de uma escolha entre duas regras, caso a primeira seja encontrada segue-se por esta regra e ignora-se a outra opção. No entanto o analisador pode ser configurado para testar todos os *lookaheads* possíveis em cada iteração. Neste modo, havendo dois *tokens* encontrados numa única iteração o analisador pára e reporta o erro. Neste modo estamos na prática a obrigar que todas as gramáticas sejam determinísticas, ou seja, em cada iteração só pode ser um único *token* encontrado. Há no entanto casos em que é útil que uma gramática possa seguir por um conjunto de regras, mas no entanto recuar caso verifique que não é o caminho correcto. Para tal foi criada a regra `<block>`, que só é aceite quando todas as regras que contem sejam aceites. Caso tal não se verifique os *tokens* já encontrados são removidos do documento e o *file-pointer* é acertado, sendo a análise retomada como se tal bloco nunca tivesse sido processado. Este comportamento é muito útil por exemplo no formato AVI onde todas as listas são precedidas do *token LIST* mas como nem todas são obrigatórias o número de ocorrências deste *token* teria de ser separado das regras para processar cada lista e teria de ser também variável. Se bem que era possível ter uma gramática determinística para analisar o ficheiro, seria bastante mais complicada de construir e de entender posteriormente.

Resumindo, apesar de ser um processo propício a erros, é possível com base nestas poucas regras descrever inteiramente diferentes formatos tanto textuais como binários. Dando o exemplo do formato AVI é possível passar para árvore qualquer

ficheiro deste formato com base numa gramática com menos de 200 linhas. Esta gramática pode ser visualizada integralmente nos anexos.

Divisão de tarefas

Construído o documento é necessário agora particioná-lo em documentos de menor dimensão computacional. No caso do AVI terão de ser removidas frames e indices, bem como corrigidos tamanhos e outras propriedades, enquanto para o formato de propriedades do PovRay teremos de introduzir novos nós que permitam restringir o processamento a um intervalo de frames de menor dimensão.

A figura 6 ilustra o processo de forma resumida. Antes de se fazer qualquer manipulação é preciso saber quantas tarefas deverão ser criadas e qual o seu peso. Para tal consulta-se o descritor da aplicação, passando o documento gerado para esta entrada como argumento caso seja necessário. O descritor devolverá um vector com o custo que determinou para esta invocação específica da aplicação e que é usado pelo Gestor de Gridlets com vista a escolher os participantes cujas características melhor servem a execução desta aplicação e também qual o número de tarefas que devem ser criadas para maximizar os recursos cedidos por estes.

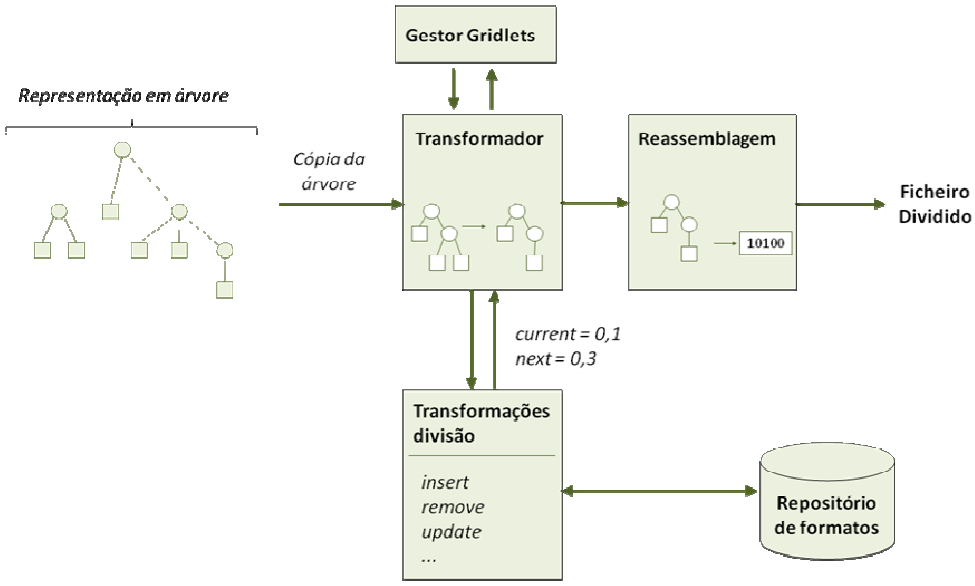


Figura 6 - Processo de criação de tarefas

O que interessa para o Transformador é o número de tarefas em que o documento irá ser particionado bem como o seu peso relativo, que será um número entre 0 e 1. O

somatório dos pesos de todas as tarefas, naturalmente tem de dar 1. As partições são feitas de forma sequencial e em cada uma o documento de partida é uma árvore composta apenas pela raiz. É sobre esta árvore/documento que todas as manipulações são feitas por omissão, isto é, o documento original pode ser não só consultado como também alterado mas é importante apontar que as partições que se seguem irão igualmente observar estas modificações o que pode ser útil para efectuar o conjunto de operações de forma mais eficiente.

As transformações são totalmente descritas num formato XML que coexiste lado a lado com a gramática do formato. As transformações, independentemente de serem de partição ou agregação de tarefas, são realizadas através de um conjunto de operações CRUD que estão contidas no descritor de cada formato. As operações que estão disponíveis para o transformador do formato são as seguintes:

- **Insert** – insere um ou mais nós (elementos ou *tokens*) antes/depois/dentro do elemento seleccionado. Os nós podem ser criados pelo transformador ou podem ser importados do documento de origem.
- **Update** – actualiza um mais nós folha da árvore com um novo valor. O novo valor é uma expressão XPath no contexto do nó que está a ser actualizado o que pode é útil em casos em por exemplo é necessário adicionar um novo valor ao já existente.
- **Delete** – apaga um mais nós da árvore, o que inclui nós folha ou nós elemento, o que no último apaga todos os nós folha que são seus filhos.

As operações são executadas de forma sequencial e podem ser aplicadas a todos os documentos transformados ou apenas a alguns deles. Para tal existem agrupamentos que restringem a que documentos as operações devem ser aplicadas:

- **First** – aplica operações só à primeira tarefa.
- **Last** – aplica operações só à última tarefa.
- **All** – aplica operações a todas as tarefas
- **Apply** – aplica a um conjunto específico de tarefas que passem um teste determinado por uma expressão XPath

Para além destes agrupamentos, é possível também aplicar um conjunto de regras a todos os documentos que não tenham sido transformados pelos agrupamentos anteriores, denominados de agrupamentos contextuais:

- **Follow** – aplica operações às tarefas seguintes (usado com first e apply)

- **Precede** – aplica operações às tarefas anteriores (usado com last e apply)
- **Otherwise** – aplica operações a todas as outras tarefas (usado com first, last e apply)

Existe também algumas funções XPath acessíveis ao transformador que são necessárias para efectuar uma partição correcta:

- **seqn**, **seql***:* número de tarefas e número total de tarefas
- **srcdoc***:* se necessário consultar o documento de origem
- **curr**, **next**, **size** – onde começa/termina a tarefa e qual o seu tamanho (só para partição)

Abaixo apresenta-se um excerto do particionador do formato AVI. Neste excerto, onde já foram descobertas as frames e índices que são necessários para esta tarefa, necessitamos agora de remover todas as frames/indíces que são redundantes e actualizar os índices que permancem para apontarem para as frames que provavelmente mudaram de posição no documento. Finalmente é necessário actualizar os cabeçalhos para reflectir todas as alterações anteriores. O transformador pode ser encontrado integralmente nos anexos.

```
    <all>
      <!-- delete unnecessary frames and indexes -->
      <variable name="rem_size" select="sum($start_frame/preceding-
sibling::frame/@size)"/>
      <delete context="$start_frame" select="preceding-sibling::frame"/>
      <delete context="$end_frame" select="following-sibling::frame"/>
      <delete context="$start_index" select="preceding-sibling::index"/>
      <delete context="$end_index" select="following-sibling::index"/>

      <!-- update indexes -->
      <update select="/idx1/index/chunk-offset" value="@value -
$rem_size"/>

      <!-- update list sizes -->
      <update select="/size" value="sum(/*/@size) - 8"/>
      <update select="/movi/size" value="sum(/movi/*/@size) - 8"/>
      <update select="/idx1/size" value="sum(/idx1/*/@size) - 8"/>

      <!-- update frame counts -->
      <variable name="vfcount" select="count(//frame[substring(chunk-
id,3,1)='d'])"/>
      <variable name="afcount" select="count(//frame[substring(chunk-
id,3,1)='w'])"/>
      <update select="//total-frames" value="$vfcount"/>
      <update select="//strh[fccType='vids']/length" value="$vfcount"/>
```

```
        <update select="//strh[fccType='auds']/length" value="$afcount"/>

    </all>
```

Figura 7 - Partição de formato AVI

Apesar de não ser usado na descrição do formato anterior é possível actualizar automaticamente o valor de determinados *tokens* em casos simples, onde o valor pode ser determinado em qualquer instante por uma pesquisa XPath ao documento. Para evitar a escrita de regras para fazer a correcção destes valores que dependem unicamente de outros nís, é possível aos *tokens* "escutarem" por mudanças num ou mais elementos:

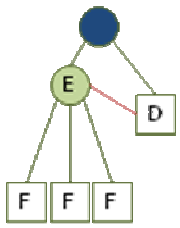


Figura 8 - Nó D escuta por mudanças no elemento E

Neste exemplo, sempre que haja mudanças no nó E ou nos seus filhos, o nó D é avisado para actualizar o seu valor que é determinado com base numa expressão XPath. Actualmente estas associações são feitas na gramática, mas seria interessante no futuro ter uma verdadeira rede de dependências entre nós onde fosse possível apenas remover as frames indesejadas, ou adicionar apenas as que interessam e tudo o resto seria removido/adicionado/actualizado por mera análise desta rede.

Agregação de resultados:

A agregação de resultados é um processo semelhante ao da divisão. A grande diferença é que as transformações são feitas no sentido de adicionar os nós necessários à árvore final. O primeiro passo é saber em que consiste a árvore final. Ao descritor do formato de saída são dadas três hipóteses. Partir de uma árvore vazia, partir da árvore do ficheiro com nº de sequência 1, ou o primeiro ficheiro a ser recebido. No caso do AVI, como a ordem é respeitada escolhe-se partir da árvore do ficheiro com nº de sequência 1.

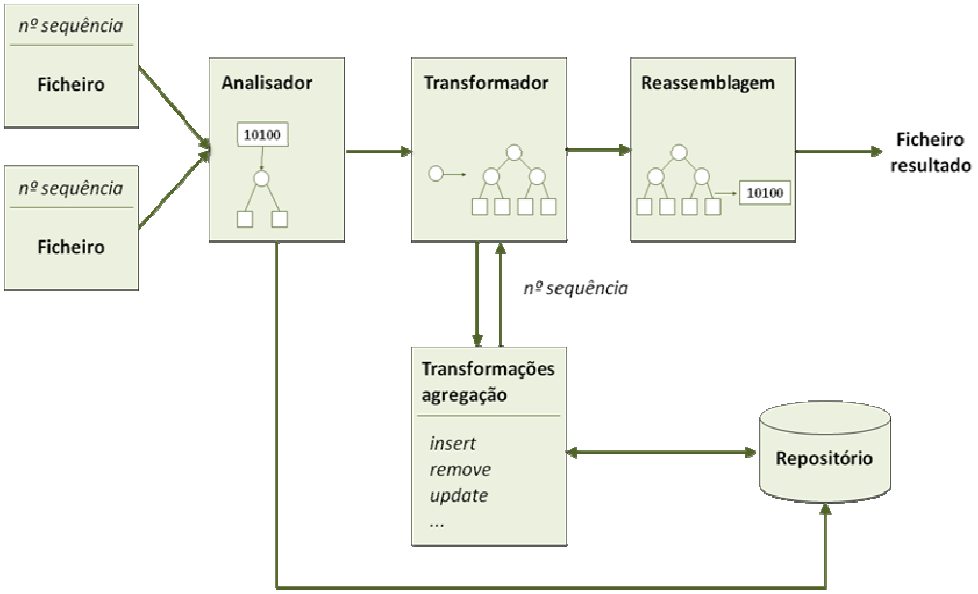


Figura 9 - Processo de agregação de resultados

Para o AVI surge agora o caso inverso, é necessário adicionar todas as frames e índices dos ficheiros resultado. E de resto o processo é semelhante, é necessário novamente corrigir indices para apontarem para as frames certas e de seguida corrigir todos os cabeçalhos, dando-se o processo por completo.

```
<first>
        <insert into="/">
          <copy select="*"/>
        </insert>
      </first>
      <otherwise>
        <variable name="movi_end" select="/movi/*[last()]/@offset +
/movi/*[last()]/@size"/>
        <insert into="/movi">
          <copy select="/movi/list-name/following-sibling::*"/>
        </insert>
        <insert into="/idx1">
          <copy select="/idx1/size/following-sibling::*" var="indexes"/>
        </insert>
        <!-- update the indexes (the context is crucial) -->
        <update select="$indexes/chunk-offset" value="@value + $movi_end -
gxf:srcdoc()/movi/@offset - 12"/>
      </otherwise>

      <!-- when the last job reply arrives, correct the headers -->
      <last>
        <update select="/movi/size" value="sum(/*/@size) - 8"/>
        <update select="/movi/size" value="sum(/movi/*/@size) - 8"/>
        <update select="/idx1/size" value="sum(/idx1/*/@size) - 8"/>
        <variable name="vfcount" select="count(//frame[substring(chunk-
```

```
id,3,4)='d'])"/>
        <variable name="afcount" select="count(//frame[substring(chunk-
id,3,4)='w'])"/>
        <update select="//total-frames" value="$vfcount"/>
        <update select="//strh[fccType='vids']/length" value="$vfcount"/>
        <update select="//strh[fccType='auds']/length" value="$afcount"/>
                                    </last>
```

Figura 10 - Agregação de AVI

Para este caso a transformação da primeira resposta consiste nas transformações dentro da tag “first”. Para as restantes, aplicam-se as transformações da tag “follow”.

4.3. Distribuição e escalonamento de tarefas

A esta fase deu-se uma menor relevância dada a enorme quantidade de literatura a este respeito, muita dela referida no trabalho relacionado.

A arquitectura de distribuição e escalonamento de tarefas está assente numa aplicação cliente servidor. Quando um novo cliente entra na rede, este envia um valor para a percentagem de ciclos que está disposto a ceder, bem como as aplicações que suporta. De seguida o servidor gera um desafio que consiste num problema, que pode ser um benchmark comum, com alguma carga computacional e que faça um grande número de leituras/escritas no sistema de ficheiros, gerando um ficheiro de saída de dimensão suficiente para ser transferido de volta e se poder aferir a velocidade de transmissão do novo participante. Tanto o cliente como o participante resolvem o problema e na recepção da resposta observa-se primeiramente se a saída gerada no cliente é correcta e campos adicionais na resposta indicam o tempo que demorou a executar o desafio dividido em tempo CPU e em tempo de I/O. Fica-se assim a conhecer a afinidade deste nó para execução de aplicações *CPU-intensive* e *I/O-intensive*. Conhecendo-se o tempo de envio é possível ainda extrapolar um valor para a largura de banda de e para este novo participante.

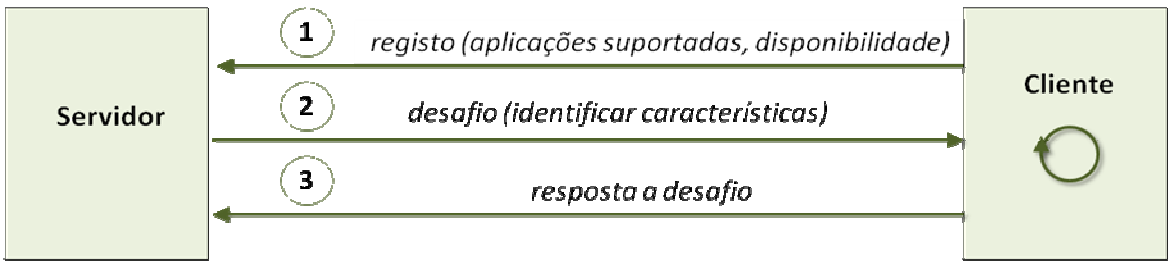


Figura 11 - Entrada de cliente

Como se falou anteriormente, na selecção de participantes para a execução da tarefa, o primeiro passo consiste em inquirir o descritor aplicacional de forma a obter um vector de custo para a mesma. Este vector tem a seguinte forma:

W = <CPU, I/O, Bandwidth>

0 =< I/O =< 1; CPU > 0; Bandwidth > 0

O elemento *CPU* neste vector é um inteiro positivo que indica o nº de ciclos de CPU que será aproximadamente necessário para executar esta tarefa relativo ao nº de ciclos de base que demora a processar o desafio que é enviado aos participantes e executado também pelo gestor de gridlets. Esta variável dá uma ideia do tempo real que demora a executar a aplicação de forma local e serve de métrica para determinar quantas gridlets devem ser criadas para executar esta tarefa de forma distribuída.

O elemento *I/O* que se segue dá um valor percentual à característica de processamento desta tarefa específica, isto é, se os seus requisitos estão mais orientados para realização de operações matemáticas ou para operações de leitura/escrita. Habitualmente esta variável estaria apenas relacionada com a aplicação em questão, mas pode haver casos em que a entrada influencia drasticamente o comportamento do programa.

Finalmente, o elemento *Bandwidth* dimensiona o tamanho das entradas e saídas e é uma variável importante na escolha de participantes quando estes se encontram em redes cuja interligação tem um débito baixo, sendo nestes casos mais interessante naturalmente enviar tarefas cujas entradas ou saídas tenham uma dimensão reduzida.

Sabendo-se o vector de custo da tarefa, é necessário agora saber em quantas *gridlets* a iremos dividir e a que participantes serão atribuídas. Naturalmente se dividirmos a tarefa em múltiplas fracções e a dividirmos para nós diferentes é previsível que o tempo total, que será determinado pelo nó mais lento a processar a sua *gridlet*, seja menor que a execução local, no entanto podemos não estar a fazer uma atribuição eficiente dos recursos até porque o desempenho pode não ser crucial do ponto de vista do utilizador, privilegiando-se um uso controlado dos recursos cedidos.

O termo gridlet tem vindo a ser utilizado ao longo do documento, simplesmente como uma porção da tarefa inicial (programa, argumentos, ficheiros, etc) que é executada de forma remota num participante. Adicionalmente uma gridlet contêm também uma noção de crédito que permite quantificar todas as tarefas que circulam na rede. Este crédito é dado igualmente por um vector:

G$ = <CPU, Bandwidth>

CPU > 0; Bandwidth > 0

A gridlet padrão é <1,1> onde as unidades podem ser MFLOP ou GFLOP para o CPU e KB ou MB para a largura de banda. Todas as gridlets são definidas com base nesta gridlet padrão que pode ser definida com base num qualquer *benchmark* típico. Como vimos anteriormente o descritor de formatos devolve uma métrica para o número de ciclos de CPU necessário para executar uma tarefa. Esta métrica pode ser baseada numa simples análise dos atributos e ficheiros de entrada da aplicação, ou pode haver uma instrumentação prévia da mesma para poder aproximar de forma mais eficiente o verdadeiro custo da tarefa face ao custo de processar a gridlet padrão. No entanto este custo varia de nó para nó, portanto são necessários ajustes para obter o verdadeiro peso computacional da gridlet.

Conhecendo-se na fase de registo a característica de cada potencial participante, é corrido um algoritmo que minimiza o tempo previsto para a execução da tarefa e que maximiza também os recursos libertos com vista a usar nós que podem ceder mais recursos, que não estão a ser utilizados momentaneamente ou aos quais podemos requisitar mais tempo computacional. Este algoritmo naturalmente poderá ser configurado para dar primazia ao desempenho gastando-se neste caso mais créditos, ou inversamente tendo uma postura mais conservadora e optando por usar menos créditos e obviamente sujeitar-se a que a aplicação demore mais a ser executada.

O protocolo de submissão de tarefas encontra-se ilustrado na figura 12.

O servidor envia primeiro um pedido de nova tarefa com um identificador da Gridlet e o custo da mesma. O cliente de acordo com a sua disponibilidade no momento pode responder OK ou Deny. Se for Deny, o servidor ignora este nó e tenta outro. Se não

houver nós disponíveis a aplicação é executada localmente. Caso seja OK é enviado a Gridlet com a tarefa que é composta pelo identificador assim como o descritor aplicacional e os ficheiros que são necessários para executar a aplicação. Finalmente recebida a Gridlet de resposta, é extraído o ficheiro de saída e passado para o reagregador de ficheiros.

No Ginger-Video-3D toda a comunicação é feita através de simples comunicação socket. O protocolo consiste no envio de mensagens na forma de objectos serializados. Todas as mensagens podem ter ficheiros anexados que são transmitidos em seguida, sendo todo o tempo de transmissão e recepção registado para efeitos estatísticos.

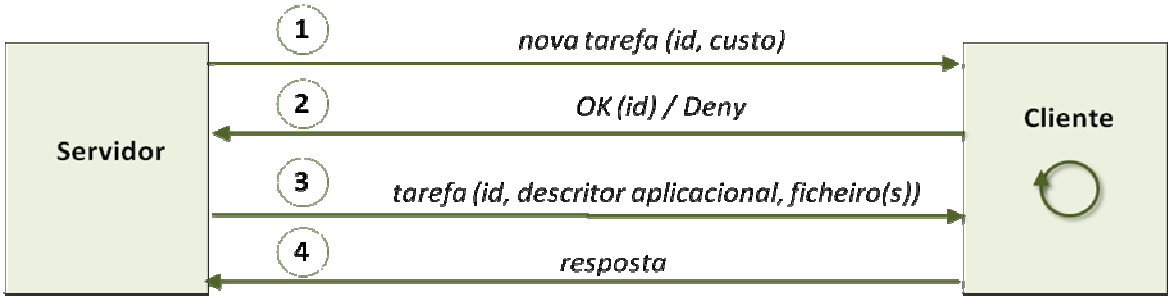


Figura 12 –Protocolo de envio de tarefas

5. Resultados Práticos

Para testar este trabalho escolheu-se duas aplicações de elevada carga computacional mas com características bastantes diferentes:

- Transcodificação de vídeo mpeg4 para h263+ (44.4MB) - Partição em keyframes.
- Transcodificação de vídeo h264 para h263+ (7.1 MB) – Partição em keyframes
- Animação de cena raytracing (250 frames) seguido de compressão em vídeo mpeg4 de alta qualidade – uso de elemento Subset_Start_Frame, Subset_End_Frame

As características das máquinas onde as aplicações foram executadas foram as seguintes:

- Sigma: Dual Opteron 2.4GHz, sistema de ficheiros partilhado AFS
- Desktop: AMD 2.2 GHz single-processor, 1GB RAM
- Portátil: Intel Core 2 Duo 2.0GHz, 2GB RAM

Estas aplicações foram testadas sobre 5 configurações possíveis, de acordo com a seguinte tabela:

Distribuição de carga	Sigma	Desktop	Portátil
Cenário 1	33%	33%	33%
Cenário 2	66%	33%	0%
Cenário 3	50%	50%	0%
Cenário 4	33%	66%	0%
Cenário 5	100%	0%	0%
Local	0%	0%	100%

Note-se que para nos resultados das configurações I-IV, o Pedido, Processamento e Resposta são retirados da *gridlet* que demora mais tempo a ser executada pelo nó seleccionado, uma vez que os outros resultados já estão todos presentes quando a

última resposta chega. Note-se ainda que na configuração V e Local os tempos de partição e agregação são nulos, uma vez que apenas uma gridlet é construída.

Transcodificação de vídeo, ficheiro de 45MB

Neste teste pode-se observar que a execução local é muito mais rápida que a execução remota. O principal problema aqui prende-se com 2 factores: a divisão do ficheiro e o tempo de envio do pedido. Por um lado a divisão obriga a fazer cópias da representação em memória do ficheiro, o que é uma operação demorada dada a sua dimensão. O algoritmo pode ser optimizado, mas o próximo problema é incontornável, porque a transmissão do pedido para o Sigma é extremamente lento o que atrasa todo o processo. É interessante observar que o Sigma apesar de ter o CPU mais rápido de todas as máquinas é o mais lento a fazer a compressão. O motivo para tal acontecer prende-se provavelmente por o sistema de ficheiros ser distribuído e estar localizado numa máquina diferente o que acumula algum overhead. Nota-se também que o tempo de divisão e reagregação é nulo no cenário 5 dado que só existe um nó de destino.

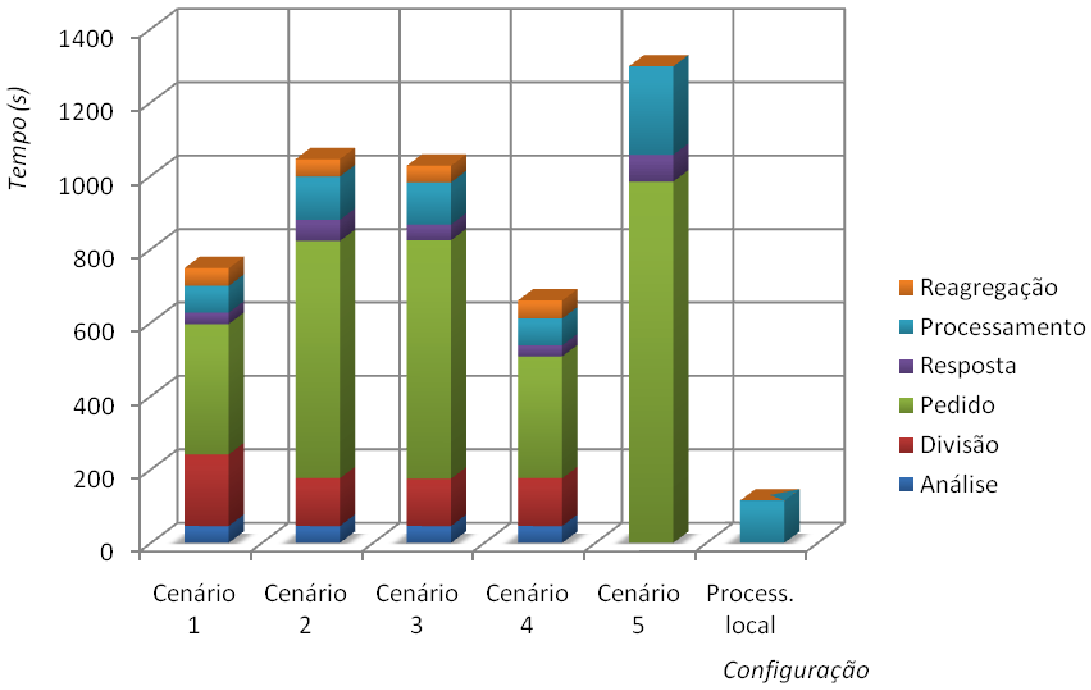


Figura 13 - Transcodificação de video, ficheiro de 45 MB

Supondo uma rede perfeita onde na prática os tempos de transmissão fossem nulos, obteríamos o seguinte resultado:

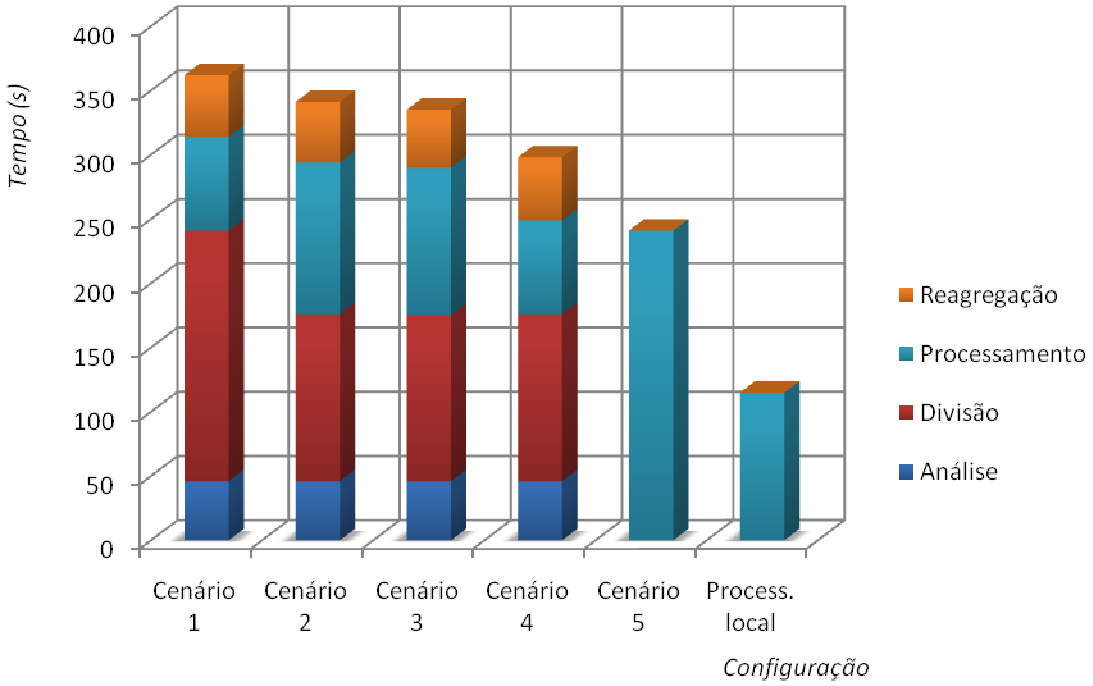


Figura 14 - Transcodificação de video, ficheiro de 45 MB, s/ tempos de transmissão

Os resultados são obviamente mais favoráveis e tirando os casos onde o Sigma tem uma fatia significativa do processamento (cenário 1, 2 e 5), o tempo de processamento necessário para executar a tarefa é menor que fazendo a execução local. É necessário agora optimizar os algoritmos de análise, partição e agregação que no cenário onde são construidas mais *gridlets* (cenário 1) compõem mais de 75% do tempo necessário para executar a aplicação de forma remota, logo resultados inspiradores uma vez que grande peso recai sobre as funções desempenhadas pelo Ginger e portanto está ao nosso alcance fazer as optimizações necessárias.

Transcodificação de vídeo, ficheiro de 7MB

Neste teste nota-se uma aproximação dos tempos da codificação nos nós remotos face à execução local, mas por observação do cenário 5 comprova-se mais uma vez que o entrave está relacionado com o envio do vídeo para o Sigma. No entanto considerando o cenário 4, onde o nó na rede local é dada a maior gridlet demonstra que executando-se a aplicação em mais que um nó exclusivamente na rede local provavelmente igualará ou melhorará o tempo obtido no processamento local.

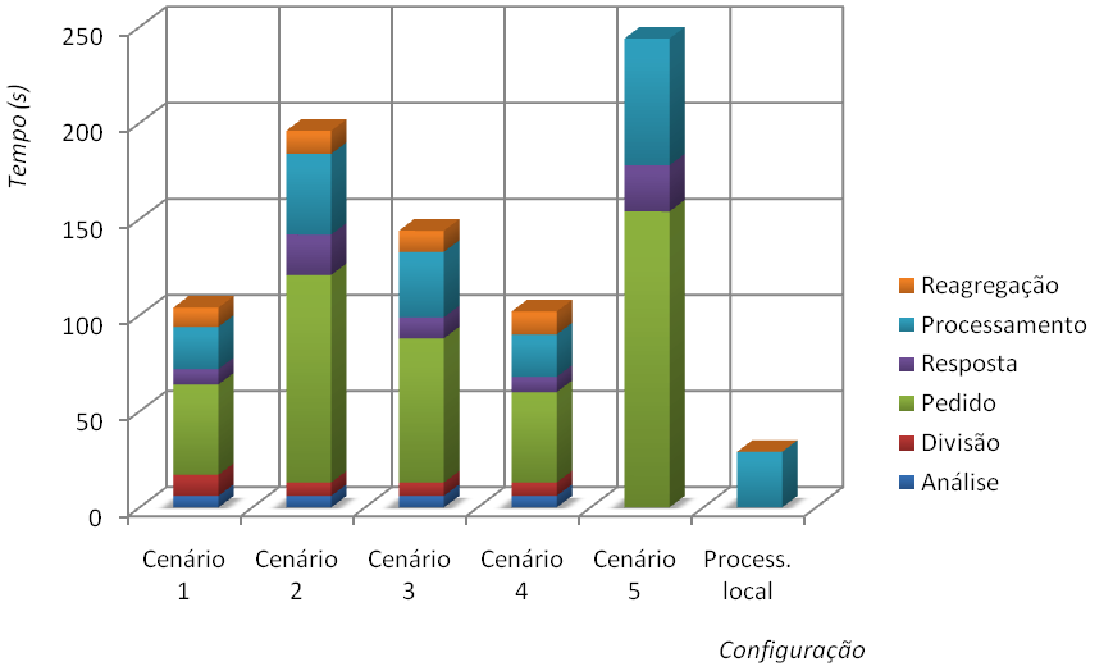


Figura 15 - Transcodificação de video, ficheiro de 7 MB

Vejamos novamente como seria o resultado se retirássemos os tempos de transmissão das gridlets para os participantes e de resposta para o cliente:

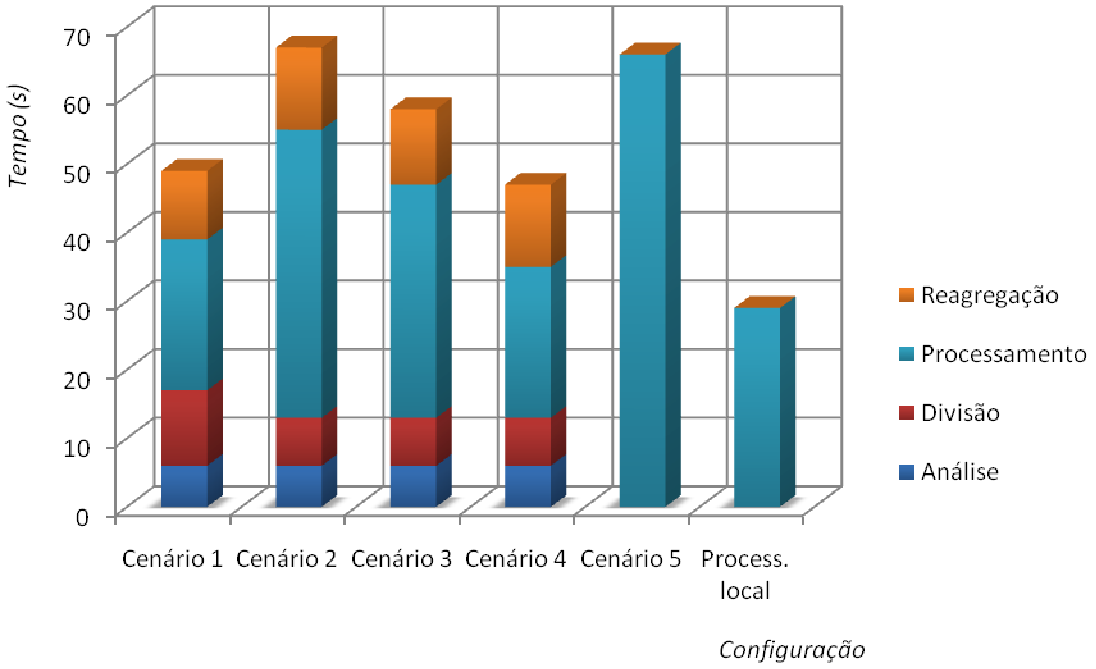


Figura 16 - Transcodificação de video, ficheiro de 45 MB, s/ tempos de transmissão

Neste teste novamente os cenários I e IV apresentam tempo de processamento abaixo do tempo da execução local. Neste teste fica é menos evidente a influência das operações de adaptação aplicadas pelo Ginger, tornando-se também claro que estas operações são bastante sensíveis à dimensão dos ficheiros de entrada, no caso do AVI à quantidade de frames e índices existentes. É também menos evidente porque o ficheiro de entrada está codificado com um codec h264 que é um codec de alta qualidade mas que tem também maior requisitos computacionais do que o codec utilizado no vídeo do teste passado.

Animação de cena povray (250 frames) + compressão video mpeg4 (alta qualidade)

A aplicação de ray-tracing é maioritariamente *cpu-intensive*, bem mais que a aplicação de transcodificação de vídeo que trabalha com entradas e saídas de grande dimensão sendo o esforço partilhado entre processamento CPU e I/O. Aliado a esta característica está o facto da descrição da cena e partição/reagregação da tarefa serem bastante mais simples e de dimensão muito reduzida, o que torna o problema do envio do pedido para o Sigma praticamente insignificante. Sendo o processador bastante mais veloz na renderização da cena as melhorias tornam-se bastantes significativas neste cenário em particular, mas naturalmente as melhorias sentem-se em todos eles em particular no cenário 2, onde o nó mais rápido (Sigma) processa a maior gridlet e o 2º mais rápido (Desktop) fica encarregue do restante.

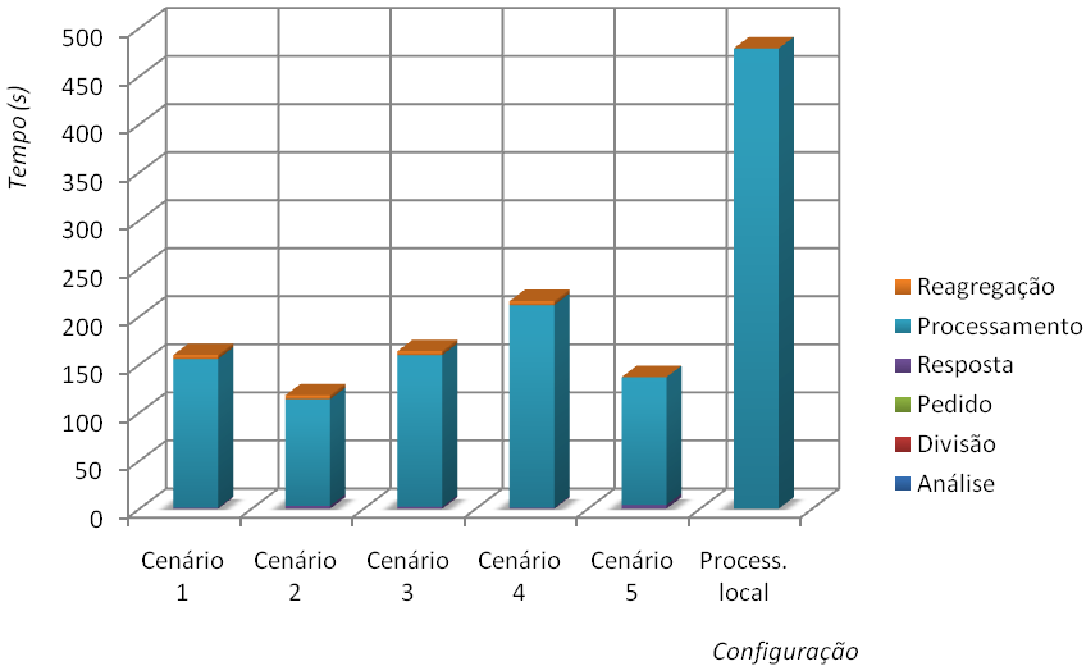


Figura 17 - Tempos de processamento - Compressão de video, ficheiro de 45 MB

6. Conclusões

Nos últimos 10 anos a computação *Grid* passou da teoria à prática e variadissimos estudos e propostas têm aparecido para preencher este espaço, de enorme potencial mas de grande dificuldade para o utilizar em pleno.

Projectos científicos ou com propósitos empresariais/académicos perfazem ainda a grande maioria das infra-estruturas *Grid* actuais e apesar de algumas destas serem acessíveis a qualquer pessoa, habitualmente são pouco versáteis e inviáveis por razões monetárias ou burocráticas.

O crescimento do interesse público na partilha de vídeos é observável no aumento claro de tráfego para alguns sites que disponibilizam este tipo de conteúdo mas levanta algumas questões técnicas. Uma delas é relativa à compressão vídeo, uma vez que os vídeo que as máquinas de filmar convencionais produzem, têm um *bitstream* bastante superior às ligações de banda larga actuais. Como a compressão ou transcodificação são processos longos, a execução paralela usando ciclos livres de máquinas de outros utilizadores na Internet surge então como uma boa alternativa no sentido de acelerar a produção de vídeos de menor dimensão e com perdas de qualidade aceitáveis.

Com este desafio em mente o Ginger apresenta-se com uma arquitectura flexível, centrada num conceito simples de *Gridlet*, capaz de se adaptar não só a ambientes altamente diversificados como é o caso da Internet, no contexto do utilizador comum, como também a diferentes aplicações (concretamente um codificador no caso do Ginger-Video-3D), abstraindo quanto possível as dificuldades presentes na execução de código em paralelo.

Foi desenhada uma arquitectura para facilmente adaptar formatos e aplicações de uso comum, com base unicamente em descritores de formatos e aplicações declarativos e facilmente reutilizáveis. Para tal criou-se uma interface que permite adaptar de forma totalmente transparente uma aplicação que normalmente se executaria num ambiente local, para passar a ser executada num ambiente distribuído em aplicações

idênticas, em que de facto se *virtualiza* a aplicação uma vez que o cliente que está a fazer o pedido de execução pode nem sequer ter a mesma instalada no seu computador. Estabeleceu-se ainda uma unidade monetária, a gridlet, que regula a economia da nossa Grid e que permite fazer uma atribuição justa de tarefas e de créditos a cada participante.

Os resultados obtidos foram também bastante satisfatórios. Ficou claro que em aplicações de vídeo o *bottleneck* é a dimensão das tarefas e não tanto o seu processamento e como tal a escolha de participantes mais indicados é fundamental, ficando também claro que é necessário fazer optimização aos algoritmos de análise, partição e agregação do Ginger. Na animação ray-tracing as entradas e saídas representam uma fracção pequena da dimensão do problema e como tal as melhorias de processamento são bastante mais evidentes.

Em qualquer dos casos é importante evidenciar que a aplicação pode estar apenas parcialmente a ser executada no computador do cliente, ou não a ser executada de todo, e como tal esta é livre de desempenhar outras tarefas.

7. Referências

[1] J. Joseph and C. Fellenstein, “Introduction to Grid Computing,” Apr. 2004; http://www.informit.com/articles/article.aspx?p=169508&seqNum=5.

[2] David P. Anderson, “Designing a Runtime System for Volunteer Computing,” Nov. 2006; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath= /dl/proceedings /&toc=comp/proceedings/ sc/2006/2700/00/2700toc.xml&DOI=10.1109/SC.2006.24.

[3] N. Andrade et al., “OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing,” *9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003; http://citeseer.ist.psu.edu/andrade03ourgrid.html.

[4] A. Goldchleger et al., “InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines,” *Concurrency and Computation: Practice and Experience*, vol. 16, 2004, pp. 449-459.

[5] L. Veiga, “GiGi: An Ocean of Gridlets on a "Grid-for-the-Masses",” May. 2007; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/ccgrid/2007/2833/00/2833toc.xml&DOI=10.1109/CCGRID.2007.54.

[6] J. Miller, “YouTube Comprises 10% Of All Internet Traffic,” Jun. 2007; http://www.webpronews.com/topnews/2007/06/19/youtube-comprises-10-of-all-internet-traffic.

[7] V. Lo et al., Cluster Computing on the ly: P2P Scheduling of Idle Cycles in the *Internet*, 2004; http://citeseer.ist.psu.edu/lo04cluster.html.

[8] M. Litzkow et al., “Condor-a hunter of idle workstations,” *Distributed Computing Systems, 1988., 8th International Conference on*, 1988, pp. 104-111.

[9] “Globus: a Metacomputing Infrastructure Toolkit,” Jun. 1997; http://hpc.sagepub.com/cgi/content/abstract/11/2/115.

[10] A. Barak et al., “An organizational grid of federated MOSIX clusters,” *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 2005, pp. 350-357 Vol. 1.

[11] A.R. Butt et al., “Java, peer-to-peer, and accountability: building blocks for distributed cycle sharing,” *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, San Jose, California: USENIX Association, 2004, pp. 13-13.

[12] A. S. Grimshaw, “Legion-a view from 50,000 feet,” Aug. 1996; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/hpdc/1996/7582/00/7582toc.xml&DOI=10.1109/HPDC.1996.546177.

[13] W. Gentzsch, “Sun Grid Engine: Towards Creating a Compute Power Grid,” May. 2001; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/ccgrid/2001/1010/00/1010toc.xml&DOI=10.1109/CCGRID.2001.923173.

[14] Chao-Tung Yang, Chao-Tung Yang, and Chuan-Lin Lai, “Apply cluster and grid computing on parallel 3D rendering,” *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, 2004, pp. 859-862 Vol.2.

[15] S.M. Akramullah, I. Ahmad, and M.L. Liou, “A data-parallel approach for real-time MPEG-2 video encoding,” *J. Parallel Distrib. Comput*, vol. 30, 1995, pp. 129-146.

[16] Ralph Ewerth, “Grid Services for Distributed Video Cut Detection,” Dec. 2004; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/ismse/2004/2217/00/2217toc.xml&DOI=10.1109/MMSE.2004.47.

[17] G. Kola, T. Kosar, and M. Livny, “A fully automated fault-tolerant system for distributed video processing and off-site replication,” *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, Cork, Ireland: ACM, 2004, pp. 122-126.

[18] F. Seinstra et al., “High-Performance Distributed Video Content Analysis with Parallel-Horus,” *Multimedia, IEEE*, vol. 14, 2007, pp. 64-75.

[19] Ping Li et al., “Design and implementation of parallel video encoding strategies using divisible load analysis,” *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 15, 2005, pp. 1098-1112.

[20] V. Li, V. Li, and Wanjiun Liao, “Distributed multimedia systems,” *Proceedings of the IEEE*, vol. 85, 1997, pp. 1063-1108.

[21] T. Little, T. Little, and A. Ghafoor, “Network considerations for distributed multimedia object composition and communication,” *Network, IEEE*, vol. 4, 1990, pp. 32-40, 45-49.

[22] L. Navarro et al., “Invasive patterns: aspect-based adaptation of distributed applications,” Nov. 2007.

[23] Yan Huang et al., “Wrapping legacy codes for Grid-based applications,” *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 7 pp.

[24] Taylor et al., “Triana Applications within Grid Computing and Peer to Peer Environments,” *Journal of Grid Computing*, vol. 1, Jun. 2003, pp. 199-217.

[25] F. Seinstra et al., “High-Performance Distributed Video Content Analysis with Parallel-Horus,” *Multimedia, IEEE*, vol. 14, 2007, pp. 64-75.

[26] Sneed, “Encapsulation of legacy software: A technique for reusing legacy software components,” *Annals of Software Engineering*, vol. 9, May. 2000, pp. 293-313.

[27] Harry M. Sneed, “Program Interface Reengineering for Wrapping,” Oct. 1997; http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/wcre/1997/8162/00/8162toc.xml&DOI=10.1109/WCRE.1997.624591.

[28] E.D. Lara et al., “Collaboration and multimedia authoring on mobile devices,” *Proceedings of the 1st international conference on Mobile systems, applications and services*, San Francisco, California: ACM, 2003, pp. 287-301; http://portal.acm.org/citation.cfm?id=1066116.1066126.

[29] E. de Lara et al., “Iterative adaptation for mobile clients using existing APIs,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, 2005, pp. 966-981.

[30] K. van der Raadt, “APST-DV: A Practical Framework for Scheduling and Deploying Divisible Loads on Grid Platforms.”

[31] Saxonica – XSLT and XQUERY processing (http://www.saxonica.com)

8. Anexos

1. Formato AVI

1.1. Gramática:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar format="avi" endianness="little" deterministic="false">

  <start>
    <ref name="list" chunk-id="RIFF" list-name="AVI "/>
    <interleave>
      <group>
        <ref name="hdrl"/>
        <optional>
          <ref name="info"/>
        </optional>
        <ref name="movi"/>
        <ref name="idx1"/>
      </group>
      <zeroOrMore>
        <ref name="junk"/>
      </zeroOrMore>
      <zeroOrMore>
        <data name="padding" size="1" type="binary"/>
      </zeroOrMore>
    </interleave>
  </start>

  <define name="chunk">
    <data name="chunk-id" type="string" size="4" equals="$chunk-id"/>
    <data name="size" type="uint32"/>
  </define>

  <define name="list" chunk-id="LIST">
    <block>
      <ref name="chunk"/>
      <data name="list-name" type="string" size="4" equals="$list-name"/>
    </block>
  </define>

  <define name="hdrl">
    <element name="hdrl">
      <ref name="list" list-name="hdrl"/>
      <ref name="avih"/>
      <sequence occurs="xpath:../avih/streams">
        <ref name="strl"/>
      </sequence>
      <optional>
        <ref name="odml"/>
      </optional>
    </element>
  </define>

  <define name="avih">
    <element name="avih">
      <ref name="chunk" chunk-id="avih"/>
```

```
      <data name="microsec-per-frame" type="uint32"/>
      <data name="max-bytes-per-sec" type="uint32"/>
      <data name="padding-granularity" type="uint32"/>
      <data name="flags" type="uint32"/>
      <data name="total-frames" type="uint32"/>
      <data name="initial-frames" type="uint32"/>
      <data name="streams" type="uint32"/>
      <data name="suggested-buffer-size" type="uint32"/>
      <data name="width" type="uint32"/>
      <data name="height" type="uint32"/>
      <sequence size="16">
        <data name="reserved" type="uint32"/>
      </sequence>
    </element>
  </define>

  <define name="strl">
    <element name="strl">
      <ref name="list" list-name="strl"/>
      <element name="strh">
        <ref name="chunk" chunk-id="strh"/>
        <data name="fccType" type="string" size="4"/>
        <data name="fccHandler" type="string" size="4"/>
        <data name="flags" type="uint32"/>
        <data name="priority" type="uint16"/>
        <data name="language" type="uint16"/>
        <data name="initial-frames" type="uint32"/>
        <data name="scale" type="uint32"/>
        <data name="rate" type="uint32"/>
        <data name="start" type="uint32"/>
        <data name="length" type="uint32"/>
        <data name="suggested-buffer-size" type="uint32"/>
        <data name="quality" type="uint32"/>
        <data name="sample-size" type="uint32"/>
        <element name="rect">
          <element name="top-left">
            <data name="x" type="uint16"/>
            <data name="y" type="uint16"/>
          </element>
          <element name="bottom-right">
            <data name="x" type="uint16"/>
            <data name="y" type="uint16"/>
          </element>
        </element>
      </element>
      <element name="strf">
        <ref name="chunk" chunk-id="strf"/>
        <data name="content" type="binary" size="xpath:preceding-
sibling::size"/>
      </element>
      <optional>
        <element name="strn">
          <ref name="chunk" chunk-id="strn"/>
          <data name="content" type="binary" size="xpath:preceding-
sibling::size"/>
        </element>
      </optional>
    </element>
  </define>
```

```
  <define name="odml">
    <element name="odml">
      <ref name="list" list-name="odml"/>
      <element name="dmlh">
        <ref name="chunk" chunk-id="dmlh"/>
        <data name="total-frames" type="uint"/>
        <data name="other" type="binary" size="xpath:number(../size) - 4"/>
      </element>
    </element>
  </define>

  <define name="info">
    <element name="info">
      <ref name="list" list-name="INFO"/>
      <sequence size="xpath:number(../size) - 4">
        <element name="isft">
          <ref name="chunk" chunk-id="ISFT"/>
          <data name="content" type="string" size="xpath:preceding-
sibling::size"/>
          <zeroOrMore>
            <data name="unknown" type="binary" size="1"/>
          </zeroOrMore>
        </element>
      </sequence>
    </element>
  </define>

  <define name="movi">
    <element name="movi">
      <ref name="list" list-name="movi"/>
      <sequence size="xpath:number(../size) - 4">
        <interleave>
          <zeroOrMore>
            <element name="frame">
              <choice>
                <ref name="chunk" chunk-id="00dc"/>
                <ref name="chunk" chunk-id="00db"/>
                <ref name="chunk" chunk-id="01wb"/>
              </choice>
              <data name="content" type="binary" size="xpath:preceding-
sibling::size"/>
              <data name="padding" type="binary" size="xpath:preceding-
sibling::size mod 2"/>
            </element>
          </zeroOrMore>
        </interleave>
      </sequence>
    </element>
  </define>

  <define name="idx1">
    <element name="idx1">
      <ref name="chunk" chunk-id="idx1"/>
      <sequence size="xpath:number(../size)">
        <element name="index">
          <data name="chunk-id" type="string" size="4"/>
          <data name="flags" type="int"/>
          <data name="chunk-offset" type="int"/>
```

```
          <data name="chunk-length" type="int"/>
        </element>
      </sequence>
    </element>
  </define>

  <define name="junk">
    <element name="junk">
      <ref name="chunk" chunk-id="JUNK"/>
      <data name="junk-content" type="binary" size="xpath:preceding-
sibling::size"/>
    </element>
  </define>

</grammar>
```

1.2. Transformadores

```
<?xml version="1.0" encoding="UTF-8"?>
<transformations format="avi">

  <partition>

    <setup>
      <jobs min="1" max="$kfcount"/>
      <variable name="movi_data" select="/movi/list-name/@offset"/>
      <variable name="kfidxs" select="/idx1/index[flags=16][substring(chunk-
id,3,5)='dc']"/>
    </setup>

    <operations>
      <all>
         <insert into="/">
           <copy select="*"/>
         </insert>
         <variable name="start_index"
select="$kfidxs[round(gxf:curr()*count($kfidxs)) + 1]"/> <!-- keyframe index
for this job -->
         <variable name="start_frame"
select="/movi/frame[@offset=$movi_data+$start_index/chunk-offset]"/> <!--
keyframe  -->
      </all>

      <!-- bind variables to the last useful frame and index -->
      <last>
        <variable name="end_index" select="/idx1/index[last()]"/>
        <variable name="end_frame" select="/movi/frame[last()]"/>
      </last>
      <otherwise>
        <variable name="end_index"
select="$kfidxs[round(gxf:next()*count($kfidxs)) + 1]/preceding-
sibling::index[1]"/>
        <variable name="end_frame"
select="/movi/frame[@offset=$movi_data+$end_index/chunk-offset]"/>
      </otherwise>

      <all>
        <!-- delete unnecessary frames and indexes -->
        <variable name="rem_size" select="sum($start_frame/preceding-
```

```
sibling::frame/@size)"/>
        <delete context="$start_frame" select="preceding-sibling::frame"/>
        <delete context="$end_frame" select="following-sibling::frame"/>
        <delete context="$start_index" select="preceding-sibling::index"/>
        <delete context="$end_index" select="following-sibling::index"/>

        <!-- update indexes -->
        <update select="/idx1/index/chunk-offset" value="@value -
$rem_size"/>

        <!-- update list sizes -->
        <update select="/size" value="sum(/*/@size) - 8"/>
        <update select="/movi/size" value="sum(/movi/*/@size) - 8"/>
        <update select="/idx1/size" value="sum(/idx1/*/@size) - 8"/>

        <!-- update frame counts -->
        <variable name="vfcount" select="count(//frame[substring(chunk-
id,3,1)='d'])"/>
        <variable name="afcount" select="count(//frame[substring(chunk-
id,3,1)='w'])"/>
        <update select="//total-frames" value="$vfcount"/>
        <update select="//strh[fccType='vids']/length" value="$vfcount"/>
        <update select="//strh[fccType='auds']/length" value="$afcount"/>
      </all>

    </operations>
  </partition>

  <aggregation>

    <setup>
      <order by="sequence"/>
    </setup>

    <operations>
      <first>
        <insert into="/">
          <copy select="*"/>
        </insert>
      </first>
      <otherwise>
        <variable name="movi_end" select="/movi/*[last()]/@offset +
/movi/*[last()]/@size"/>
        <insert into="/movi">
          <copy select="/movi/list-name/following-sibling::*"/>
        </insert>
        <insert into="/idx1">
          <copy select="/idx1/size/following-sibling::*" var="indexes"/>
        </insert>
        <!-- update the indexes (the context is crucial) -->
        <update select="$indexes/chunk-offset" value="@value + $movi_end -
gxf:srcdoc()/movi/@offset - 12"/>
      </otherwise>

      <!-- when the last job reply arrives, correct the headers -->
      <last>
        <update select="/movi/size" value="sum(/*/@size) - 8"/>
        <update select="/movi/size" value="sum(/movi/*/@size) - 8"/>
        <update select="/idx1/size" value="sum(/idx1/*/@size) - 8"/>
```

```
        <variable name="vfcount" select="count(//frame[substring(chunk-
id,3,4)='d'])"/>
        <variable name="afcount" select="count(//frame[substring(chunk-
id,3,4)='w'])"/>
        <update select="//total-frames" value="$vfcount"/>
        <update select="//strh[fccType='vids']/length" value="$vfcount"/>
        <update select="//strh[fccType='auds']/length" value="$afcount"/>
      </last>

    </operations>
  </aggregation>
</transformations>
```

2. Formato Pov-Ray INI

2.1 Gramática

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar format="povray-ini" deterministic="false">

  <start>
    <interleave>
      <ref name="property" pname="Initial_Frame"/>
      <ref name="property" pname="Final_Frame"/>
      <ref name="property" pname="Initial_Clock"/>
      <ref name="property" pname="Final_Clock"/>
      <zeroOrMore>
        <ref name="property"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="comment"/>
      </zeroOrMore>
      <zeroOrMore>
        <ref name="newline"/>
      </zeroOrMore>
    </interleave>
  </start>

  <define name="property" pname="\w+">
    <element name="property">
      <data name="name" type="string" regex="$pname"/>
      <data name="equals" type="char" equals="="/>
      <data name="value" type="string" regex=".+"/>
    </element>
  </define>

  <define name="comment">
    <block>
      <data name="comment_start" type="string" regex="\;\s*"/>
      <data name="comment" type="string" regex=".*"/>
    </block>
  </define>

  <define name="newline">
    <data name="newline" type="string" regex="((\r)?\n)+"/>
  </define>

</grammar>
```

2.2 Transformadores

```
<?xml version="1.0" encoding="UTF-8"?>
<transformations format="povray-ini">

  <!-- povray ini files only make sense to partition, not to aggregate -->

  <partition>

    <setup>
```

```
      <jobs min="1" max="/property[name='Final_Frame'] -
/property[name='Initial_Frame'] + 1"/>
    </setup>

    <operations>
      <all>
        <!-- remove the properties just in case they already exist -->
        <delete select="/property[name='Subset_Start_Frame' or
name='Subset_End_Frame']"/>
        <insert into="/">
          <data name="subset_start" type="string"
value="concat('Subset_Start_Frame=', gxf:curr(), '\n\r')"/>
          <data name="subset_end" type="string"
value="concat('Subset_End_Frame=', gxf:next())"/>
        </insert>
      </all>
    </operations>

  </partition>

</transformations>
```