LABORATÓRIO NACIONAL
DE ENGENHARIA CIVIL

PARFLUDAN: SOFTWARE PARALELO PARA ANÁLISE ESTRUTURAL DE ELEMENTOS FINITOS

Núcleo de Tecnologias da Informação em Engenharia Civil

Lisboa • junho de 2016

I&D TECNOLOGIAS DA INFORMAÇÃO

RELATÓRIO 198/**2016 – NTIEC**

Título

PARFLUDAN: SOFTWARE PARALELO PARA ANÁLISE ESTRUTURAL DE ELEMENTOS FINITOS

Autoria

NÚCLEO DE TECNOLOGIAS DA INFORMAÇÃO EM ENGENHARIA CIVIL

João Coelho
Bolseiro de Iniciação à Investigação Científica, NTIEC

António Silva
Investigador Principal, NTIEC

Copyright © LABORATÓRIO NACIONAL DE ENGENHARIA CIVIL, I. P.

AV DO BRASIL 101 • 1700-066 LISBOA

e-mail: lnec@lnec.pt

www.lnec.pt

Relatório 198/2016

Proc. 0109/112/20179

PARFLUDAN: SOFTWARE PARALELO PARA ANÁLISE ESTRUTURAL DE ELEMENTOS FINITOS

Resumo

Ao longo dos anos recentes, a atividade desenvolvida nas áreas de engenharia tem-se apoiado cada vez mais na utilização de ferramentas de *software* apropriadas. Estas ferramentas são muitas vezes desenvolvidas especificamente para dar resposta a um problema muito bem definido, faltando-lhes a capacidade para resolver de forma eficiente problemas semelhantes computacionalmente mais exigentes. A paralelização destas ferramentas é uma das abordagens existentes para permitir aumentar a sua eficiência e aumentar a complexidade dos problemas admissíveis.

Este relatório introduz o programa *ParFludan*, um programa paralelo para análise estrutural pelo método dos elementos finitos, que surgiu da necessidade de melhorar o desempenho do programa *Fludan-RAS*. Executado no *cluster Medusa* do LNEC, este novo programa permite tratar problemas de maior complexidade do que o original, com melhorias significativas ao nível dos tempos de execução. Além da descrição do processo de paralelização do programa original e das tecnologias para tal utilizadas, é também feita a análise do programa desenvolvido e são dadas instruções para a sua utilização, com o objetivo não só de facilitar a utilização do *ParFludan* como também de providenciar algumas linhas de orientação para a paralelização de outros programas em condições semelhantes.

Palavras-chave: Paralelização / Elementos finitos / Análise estrutural / *Medusa*

PARFLUDAN: PARALLEL SOFTWARE FOR STRUCTURAL ANALYSIS OF FINITE ELEMENTS

Abstract

In recent years, engineering practices have increasingly relied on suitable software tools to solve relevant challenges. Often these tools are developed specifically in order to solve a particular problem, lacking the ability to efficiently address other similar but computationally more demanding challenges. The parallelization of such tools is one possible approach to increase their efficiency and enhance their ability to solve more complex problems.

This report introduces the software program *ParFludan*, a parallel tool for structural behaviour analysis via the finite element method, which was born out of the need to improve the performance of the sequential program *Fludan-RAS*. Running on LNEC's *Medusa* cluster, the new program allows for dealing with more complex problems while managing to outperform the original program significantly, in terms of total running time. In addition to describing the paralellization process and the technologies used for that effect, this report includes an analysis of the newly-developed program and provides instructions on how to use it, its purpose being not only to allow for *ParFludan*'s employment to solve

appropriate problems but also to provide guidelines for the parallelization of other programs subject to similar conditions.

Keywords: Parallelization / Finite elements / Structural analysis / *Medusa*

Índice

1 | Introdução 1
1.1 Motivação 2
1.2 Condições iniciais 3
1.3 Proposta de solução 6
1.4 Estrutura do documento 7
2 | Desenvolvimento do programa 8
2.1 Integração de ficheiros de dados 8
2.2 Paralelização 9
2.2.1 Estruturas de dados globais e distribuídas 9
2.2.2 Partições de domínios 10
2.2.3 Criação das estruturas de dados 11
2.2.4 Resolução do sistema de equações 11
2.2.5 Implementação do modelo de dano 12
2.3 Modularização 12
2.4 Documentação 13
3 | Estrutura do programa 14
3.1 Visão geral 14
3.1.1 A pasta *data/* 14
3.1.2 A pasta *include/* 16
3.1.3 A pasta *results/* 16
3.1.4 A pasta *src/* 16
3.1.5 A pasta *src_seq/* 17
3.1.6 A pasta *tests/* 17
3.2 Organização dos módulos 18
3.2.1 Módulos do programa principal 18
3.2.2 Módulos auxiliares 19
3.3 Gestão das execuções 20
4 | Avaliação do programa 21
4.1 Exatidão dos resultados 21
4.1.1 Testes unitários 22
4.1.2 Testes de integração 23
4.2 Avaliação da solução 24
5 | Como utilizar 26
5.1 Instalação 26
5.1.1 Dependências 26
5.1.2 Programa 27
5.2 Interfaces 27
5.2.1 Ficheiros de entrada 27
5.2.2 Ficheiros de saída 32
5.3 Ficheiro de parametrizações 35
5.4 Executar uma simulação 35
5.4.1 Exemplo de execução de uma simulação 36
5.4.2 Parâmetros 36
5.4.3 Exemplos de utilização 37
5.5 Executar os testes 37

6 | Customização do programa 39
6.1 Como adicionar um cenário....................39
6.2 Como adicionar um teste....................39
6.3 Como adicionar um módulo....................40
7 | Conclusão 41
Agradecimentos....................42
Referências Bibliográficas 44
Anexos....................45
ANEXO I Exemplo do ficheiro de configurações *settings.c*....................47
ANEXO II Exemplo simplificado de ficheiro de dados *TDIN.DAD*....................51
ANEXO III Exemplo simplificado do ficheiro de dados *FACESM.DAD* 57
ANEXO IV Exemplo simplificado do ficheiro de dados *VARTERM.DAD* 61

Índice de figuras

Figura 1.1 – Malha de elementos finitos da barragem de Baixo-Sabor (escalão de montante).............2
Figura 1.2 – Algoritmo computacional simplificado para o cálculo estrutural. Adaptado de (Piteira Gomes, 2008)...5
Figura 3.1 – Esquema resumido do programa 15
Figura 5.1 – Visualização dos deslocamentos obtidos pela simulação (malha de Peti) 34

Índice de tabelas

Tabela 5.1 – Tipos de ficheiros de entrada admitidos 28
Tabela 5.2 – Designações e caraterísticas das várias solicitações disponíveis 30
Tabela 5.3 – Descrição de alguns parâmetros opcionais 36

1 | Introdução

Ao longo dos últimos anos, a utilização de *software* tem vindo a assumir um papel cada vez mais central no processo de investigação científica e desenvolvimento tecnológico, em particular na área da engenharia. Apesar de existirem numerosas soluções disponíveis, tanto comerciais como de utilização aberta, para resolver a maioria dos problemas, nos casos em que o problema em questão é especialmente singular é frequentemente necessário desenvolver um programa especificamente para tratar esse problema, o que é normalmente feito pelos próprios cientistas ou engenheiros por ele responsáveis.

Embora as soluções desenvolvidas resolvam o problema original, a sua aplicação a problemas semelhantes em diferentes condições é normalmente complicada ou mesmo impossível, como resultado do seu foco em resolver um problema específico e não em serem generalizáveis para uma classe de problemas ou para diferentes requisitos do utilizador. Exemplos típicos deste fenómeno são a necessidade de resolver de um problema do mesmo tipo mas de maior complexidade, que pode exceder os limites tolerados pelo programa, ou objetivo de resolver um problema semelhante em menos tempo, para o qual é frequente a solução não estar suficientemente otimizada.

Uma situação desta natureza esteve na origem do desenvolvimento do programa *ParFludan*, que consiste num programa de *software* para a análise do comportamento de estruturas homogéneas de grandes dimensões através do método dos elementos finitos, cujo desenvolvimento este relatório tem como objetivo reportar. A análise numérica do comportamento estrutural, sob diferentes condições, é uma componente essencial para a avaliação das condições de segurança de estruturas em engenharia civil, com aplicações no estudo de barragens, pontes ou viadutos, por exemplo. Este programa foi desenvolvido no LNEC com o objetivo de permitir a resolução numérica de problemas de complexidade crescente, tirando partido dos recursos computacionais disponíveis no laboratório, nomeadamente do *cluster Medusa*.

O *ParFludan* foi desenvolvido a partir do *Fludan-RAS*, um programa já existente no LNEC para a simulação do comportamento estrutural de barragens de betão, com particular destaque para a modelação do seu comportamento, em regime viscoelástico e com dano, quando sujeitas a ações expansivas com origem em reações químicas do betão. As limitações deste programa, nomeadamente o seu excessivo tempo de execução e a sua incapacidade para resolver problemas com a complexidade desejada, foram a principal motivação para o desenvolvimento do *ParFludan*.

Da leitura deste relatório devem ser claras as respostas às questões:

- Para que serve este programa?
- Quais foram a estratégia e as tecnologias usadas no seu desenvolvimento?
- Como está organizada?
- De que forma melhora a solução previamente existente?
- Como se usa?

1.1 Motivação

No contexto da engenharia de estuturas, a modelação de sistemas físicos através de formulações matemáticas com base nas variáveis relevantes para a caraterização do sistema é uma das principais ferramentas para a análise e compreensão da resposta estrutural. No LNEC, a modelação do comportamento macroscópico de estruturas de grandes dimensões, como barragens ou viadutos, com base nas caraterísticas microscópicas do sistema, como a sua geometria, as propriedades dos seus materiais e a influência de fatores ambientais, assume especial importância dado o papel central do LNEC no desenvolvimento e manutenção destas estruturas. Uma das áreas em que este tipo de abordagem tem maior relevo é na engenharia de barragens de betão, em particular no estudo do comportamento de barragens sujeitas a processos expansivos durante longos períodos de tempo.

A necessidade de um programa para simulação do comportamento estrutural de barragens no domínio do tempo levou ao desenvolvimento no LNEC do *Fludan-RAS*, um programa que considera a evolução do comportamento de uma barragem com recurso ao método de elementos finitos e de uma formulação que considera o comportamento diferido do betão, através de um modelo viscoelástico de maturação, bem como o surgimento e propagação de fenómenos de dano. A estrutura de elementos finitos da barragem é definida por meio de uma malha de elementos finitos, um conjunto de nós, elementos e ligações que estabelecem a geometria da estrutura e as interações entre os seus diferentes elementos. A figura 1.1 exibe uma visualização de uma destas malhas. O modelo estrutural em que o *Fludan-RAS* é baseado assume que o comportamento da estrutura é influenciado por cinco fatores: as suas caraterísticas geométricas, o tipo de comportamento dos seus materiais, o equilíbrio estrutural, as ligações exteriores e as ações, em particular as ações decorrentes de reações químicas de origem interna. A fundamentação e descrição do desenvolvimento deste programa estão descritos em detalhe em (Piteira Gomes, 2008).

Figura 1.1 – Malha de elementos finitos da barragem de Baixo-Sabor (escalão de montante)

Computacionalmente, porém, o *Fludan-RAS* apresenta algumas limitações. A principal dessas limitações é o tempo de execução de cada simulação, que para períodos de análise longos, da ordem dos 50 anos, chega a atingir valores na ordem das semanas. Outra limitação significativa do *Fludan-*

RAS reside no facto de impor limites relativamente baixos ao tamanho das malhas que pode processar, originando a necessidade de utilizar malhas grosseiras em que cada unidade elementar atinge facilmente dimensões da ordem das dezenas de metro cúbico. Por último, uma limitação menos visível mas igualmente significativa é o facto de não existir documentação adequada do programa, o que coloca obstáculos significativos à sua manutenção e desenvolvimento.

O desenvolvimento do *ParFludan* pretende dar resposta às limitações do *Fludan-RAS*, tirando partido de um novo paradigma computacional, o da computação paralela. Através da paralelização do programa, é possível distribuir a carga computacional e a ocupação de recursos por várias unidades computacionais (UC), em vez de concentrar tudo numa única, de forma a que o tempo total de execução das simulações seja reduzido e, simultaneamente, o programa seja capaz de lidar com malhas de maior dimensão. Esta estratégia é particularmente adequada ao contexto do LNEC, em que existe uma infraestrutura de computação paralela, nomeadamente o *cluster Medusa*, que pode ser aproveitada para executar as simulações (Coelho, Silva; 2013). Adicionalmente, a documentação do *ParFludan* será um contributo futuro para a sua utilização, generalização e manutenção.

Em resumo, o *ParFludan* foi criado resolver os problemas de tempo de execução e dimensão das malhas exibidos pelo *Fludan-RAS*, preservado a sua funcionalidade, em especial a modelação do comportamento viscoelástico de estruturas homogéneas de betão, solucionando a equação de equilíbrio na forma incremental no tempo, sendo cada passo resolvido em duas fases: uma primeira fase para contemplar a variação das ações no intervalo, admitindo que são aplicadas instantaneamente no intervalo – resposta instantânea – e uma segunda fase para fazer refletir a ação das cargas aplicadas durante o intervalo – resposta diferida. Tirando partido do novo paradigma de computação paralela, é possível minimizar as limitações do programa original e permitir o processamento de estruturas arbitrariamente grandes, com malhas arbitrariamente refinadas, em tempos de execução aceitáveis.

1.2 Condições iniciais

O desenvolvimento do *ParFludan* foi largamente baseado no *Fludan-RAS*, pelo que é importante apresentar com algum detalhe a sua estrutura e funcionalidade original, que foram em grande parte preservadas no *ParFludan*. A figura 1.2 descreve resumidamente o algoritmo computacional utilizado pelo *Fludan-RAS*, que foi também implementado no *ParFludan*, devidamente modificado para tirar partido da paralelização.

Este algoritmo compreende essencialmente duas fases:

1. Leitura dos ficheiros de entrada e preparação das estruturas de dados;
2. Execução da simulação.

Na primeira fase o programa lê os dados de entrada necessários para executar a simulação, nomeadamente a geometria da malha, as propriedades dos materiais, as condições de apoio e as ações a que a estrutura está sujeita, organizando essa informação nas estruturas de dados que serão utilizadas na fase de simulação. Esta informação está contida nos ficheiros de dados disponibilizados

pelo utilizador, de acordo com as especificações definidas na secção 5.2.1. Na segunda fase, o programa usa estes dados para executar a simulação do comportamento estrutural no domínio do tempo, com base no método dos elementos finitos e nos princípios descritos em (Piteira Gomes, 2008).

A diferença fundamental entre ambas estas fases é que a primeira é executada apenas uma vez, ao passo que a segunda é executada múltiplas vezes, uma vez que consiste num ciclo aos intervalos de tempo discretizados para simular a evolução temporal do comportamento da estrutura. Como consequência, a segunda fase é muito mais exigente do ponto de vista computacional.

Durante o ciclo temporal são tratadas duas fases em cada iteração, a fase instantânea e a fase diferida. A fase instantânea considera as variações das solicitações a que a estrutura está sujeita, como o peso próprio ou a pressão hidroestática, enquanto que a fase diferida serve para simular o efeito da história de carga. A alternância entre ambas as fases ao longo do tempo permite ao algoritmo simular o comportamento real da estrutura. Em ambas as fases, o programa monta uma matriz de rigidez K, cria um vetor de forças f, resolve o sistema $Kx=f$ para obter o vetor de deslocamentos x e calcula as restantes grandezas relevantes (tensões e extensões) a partir dos deslocamentos. Adicionalmente, considera ainda um modelo de dano, que só é ativado quando surge dano na estrutura – tipicamente, ao final de várias iterações no tempo – e consiste na resolução iterativa de múltiplos sistemas da forma $Kx=f$, com incremento de dano, até que o sistema atinja o equilíbrio.

Do ponto de vista computacional, o *bottleneck* deste algoritmo está na resolução iterativa do sistema de equações. A montagem da matriz de rigidez, que é feita em cada iteração, também contribui significativamente para o tempo de execução do programa, mas a resolução do sistema é o fator dominante. Nesse sentido, este *bottleneck* é especialmente agravado quando o modelo de dano é ativado, uma vez que consiste na resolução, em cada iteração no tempo, de múltiplas instâncias do sistema $Kx=f$.

Ainda que muito simplificada, esta análise permite traçar três linhas gerais a ter em conta no desenvolvimento do *ParFludan*:

- É necessário otimizar a resolução do sistema $Kx=f$, uma vez que é o fator que mais contribui para o tempo de execução do programa;
- É conveniente preparar a solução para utilizar os recursos de computação paralela disponíveis no LNEC, nomeadamente o *cluster Medusa*;

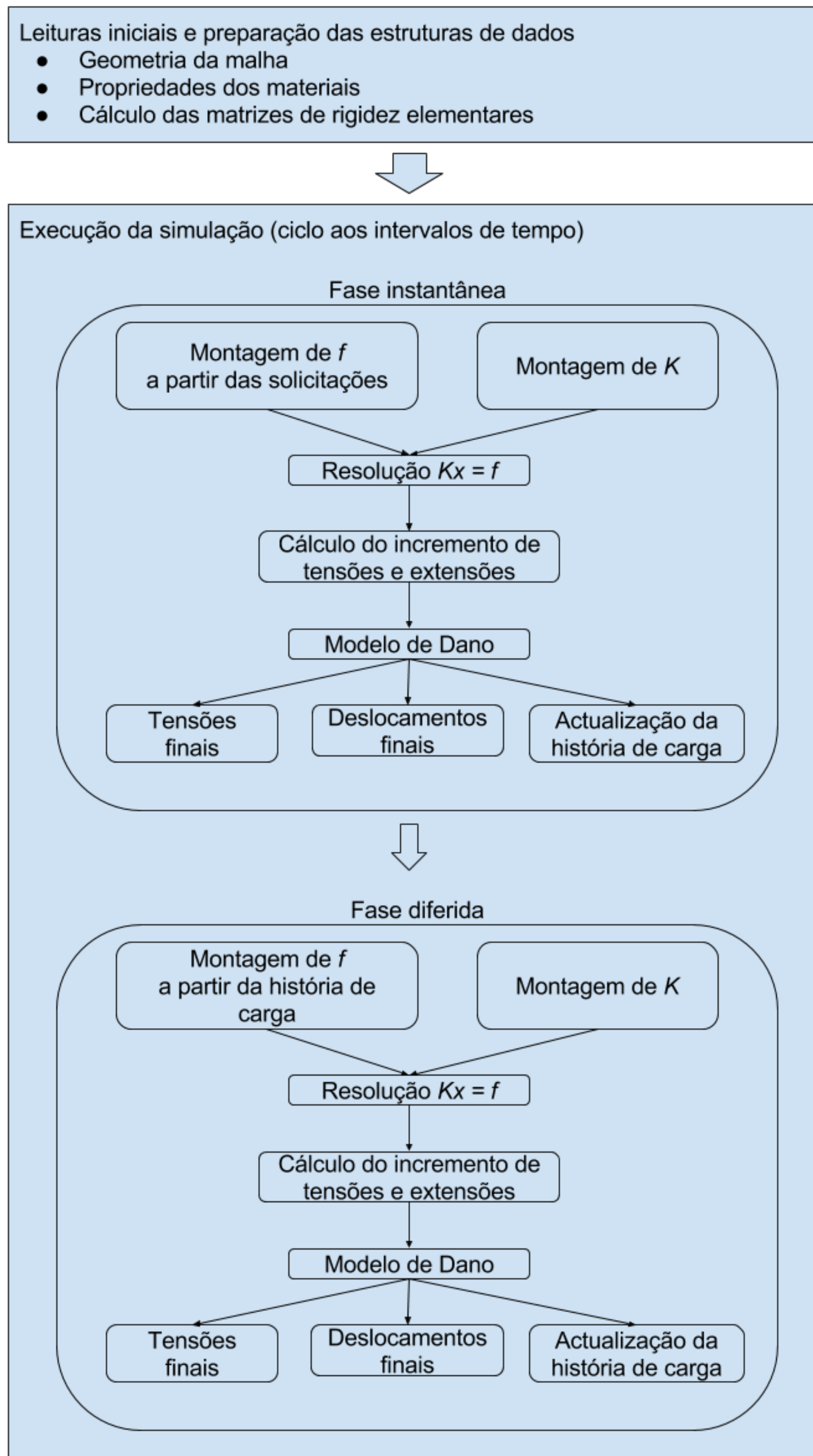


Figura 1.2 – Algoritmo computacional simplificado para o cálculo estrutural. Adaptado de (Piteira Gomes, 2008)

- É conveniente separar o programa em múltiplas partes independentes, seguindo o esquema lógico da figura 1.2, para que cada uma delas possa ser tratada de forma isolada.

1.3 Proposta de solução

Tendo em conta as particularidades do *Fludan-RAS* e as orientações retiradas da análise ao seu funcionamento, foi desenhada uma proposta de solução que permite dar resposta aos objetivos estabelecidos para o *ParFludan*.

No essencial, a estratégia adotada para esta solução baseou-se na paralelização da resolução do sistema de equações que constitui o principal *bottleneck* do *Fludan-RAS*, segundo o paradigma de memória distribuída, não só por permitir escalar o desempenho do programa como também por ser o mais adequado à utilização de um *cluster* como o *Medusa*; por sua vez, esta decisão implicou alterar a forma de preparação das estruturas de dados relevantes, de forma a serem adequadas ao processamento paralelo. Tendo em conta que para tirar benefício da paralelização todas as estruturas de dados envolvidas nas partes mais importantes do programa teriam de ser reorganizadas, esta estratégia acabou por implicar a reescrita integral do programa.

A necessidade de alterar a forma de armazenamento e, por conseguinte, de tratamento das estruturas de dados é evidente, por exemplo, no caso da matriz de rigidez K. Esta matriz tem dimensão $(N*3)^2$, onde N é o número de nós da malha; numa malha com 10 mil nós, a matriz de rigidez tem 9E+08 entradas, o que significa que, se cada elemento for representado por um tipo de dados *float* de 8 *bytes*, esta estrutura ocupa perto de 6,7GB de memória, o que rapidamente se torna incompatível com os limites de memória usuais se não for distribuída por várias UC. Desta forma, a definição das estruturas de dados limita o tamanho das malhas admissíveis pelo programa[1].

A nível tecnológico, a paralelização foi levada a cabo utilizando o modelo de comunicação padrão em computação distribuída, o *Message Passing Interface (MPI)*[2], através da implementação *OpenMPI* (Gabriel *et al*, 2004). Adicionalmente, uma vez identificada a resolução do sistema $Kx=f$ como o principal *bottleneck*, tornou-se imperativo encontrar uma biblioteca paralela de computação numérica para esse efeito, tendo a escolha recaído sobre o *toolkit PETSc*[3] (BALAY *et al*, 2016) (BALAY, GROPP, MCINNES, *et al*; 1997). Para a escrita do código do programa, foi adotada a linguagem de programação C, por permitir uma interface mais harmoniosa com o *MPI* e o *PETSc* do que o *Fortran*, a linguagem utilizada no *Fludan-RAS*. Para controlo de versões foi adotado o *Git*[4], enquanto que os *script*s para integrar as diferentes componentes do programa e gerir as submissões para execução foram escritos em *bash* e *python*.

[1] É evidente que outros esquemas de armazenamento permitem reduzir o espaço em memória exigido, em particular no caso de matrizes esparsas; no entanto, para malhas suficientemente grandes, essa solução não é suficiente para resolver o problema.

[2] http://www.mpi-forum.org/

[3] https://www.mcs.anl.gov/petsc/

[4] https://git-scm.com/documentation

1 | Introdução

Ao longo dos últimos anos, a utilização de *software* tem vindo a assumir um papel cada vez mais central no processo de investigação científica e desenvolvimento tecnológico, em particular na área da engenharia. Apesar de existirem numerosas soluções disponíveis, tanto comerciais como de utilização aberta, para resolver a maioria dos problemas, nos casos em que o problema em questão é especialmente singular é frequentemente necessário desenvolver um programa especificamente para tratar esse problema, o que é normalmente feito pelos próprios cientistas ou engenheiros por ele responsáveis.

Embora as soluções desenvolvidas resolvam o problema original, a sua aplicação a problemas semelhantes em diferentes condições é normalmente complicada ou mesmo impossível, como resultado do seu foco em resolver um problema específico e não em serem generalizáveis para uma classe de problemas ou para diferentes requisitos do utilizador. Exemplos típicos deste fenómeno são a necessidade de resolver de um problema do mesmo tipo mas de maior complexidade, que pode exceder os limites tolerados pelo programa, ou objetivo de resolver um problema semelhante em menos tempo, para o qual é frequente a solução não estar suficientemente otimizada.

Uma situação desta natureza esteve na origem do desenvolvimento do programa *ParFludan*, que consiste num programa de *software* para a análise do comportamento de estruturas homogéneas de grandes dimensões através do método dos elementos finitos, cujo desenvolvimento este relatório tem como objetivo reportar. A análise numérica do comportamento estrutural, sob diferentes condições, é uma componente essencial para a avaliação das condições de segurança de estruturas em engenharia civil, com aplicações no estudo de barragens, pontes ou viadutos, por exemplo. Este programa foi desenvolvido no LNEC com o objetivo de permitir a resolução numérica de problemas de complexidade crescente, tirando partido dos recursos computacionais disponíveis no laboratório, nomeadamente do *cluster Medusa*.

O *ParFludan* foi desenvolvido a partir do *Fludan-RAS*, um programa já existente no LNEC para a simulação do comportamento estrutural de barragens de betão, com particular destaque para a modelação do seu comportamento, em regime viscoelástico e com dano, quando sujeitas a ações expansivas com origem em reações químicas do betão. As limitações deste programa, nomeadamente o seu excessivo tempo de execução e a sua incapacidade para resolver problemas com a complexidade desejada, foram a principal motivação para o desenvolvimento do *ParFludan*.

Da leitura deste relatório devem ser claras as respostas às questões:

- Para que serve este programa?
- Quais foram a estratégia e as tecnologias usadas no seu desenvolvimento?
- Como está organizada?
- De que forma melhora a solução previamente existente?
- Como se usa?

1.1 Motivação

No contexto da engenharia de estuturas, a modelação de sistemas físicos através de formulações matemáticas com base nas variáveis relevantes para a caraterização do sistema é uma das principais ferramentas para a análise e compreensão da resposta estrutural. No LNEC, a modelação do comportamento macroscópico de estruturas de grandes dimensões, como barragens ou viadutos, com base nas caraterísticas microscópicas do sistema, como a sua geometria, as propriedades dos seus materiais e a influência de fatores ambientais, assume especial importância dado o papel central do LNEC no desenvolvimento e manutenção destas estruturas. Uma das áreas em que este tipo de abordagem tem maior relevo é na engenharia de barragens de betão, em particular no estudo do comportamento de barragens sujeitas a processos expansivos durante longos períodos de tempo.

A necessidade de um programa para simulação do comportamento estrutural de barragens no domínio do tempo levou ao desenvolvimento no LNEC do *Fludan-RAS*, um programa que considera a evolução do comportamento de uma barragem com recurso ao método de elementos finitos e de uma formulação que considera o comportamento diferido do betão, através de um modelo viscoelástico de maturação, bem como o surgimento e propagação de fenómenos de dano. A estrutura de elementos finitos da barragem é definida por meio de uma malha de elementos finitos, um conjunto de nós, elementos e ligações que estabelecem a geometria da estrutura e as interações entre os seus diferentes elementos. A figura 1.1 exibe uma visualização de uma destas malhas. O modelo estrutural em que o *Fludan-RAS* é baseado assume que o comportamento da estrutura é influenciado por cinco fatores: as suas caraterísticas geométricas, o tipo de comportamento dos seus materiais, o equilíbrio estrutural, as ligações exteriores e as ações, em particular as ações decorrentes de reações químicas de origem interna. A fundamentação e descrição do desenvolvimento deste programa estão descritos em detalhe em (Piteira Gomes, 2008).

Figura 1.1 – Malha de elementos finitos da barragem de Baixo-Sabor (escalão de montante)

Computacionalmente, porém, o *Fludan-RAS* apresenta algumas limitações. A principal dessas limitações é o tempo de execução de cada simulação, que para períodos de análise longos, da ordem dos 50 anos, chega a atingir valores na ordem das semanas. Outra limitação significativa do *Fludan-*

RAS reside no facto de impor limites relativamente baixos ao tamanho das malhas que pode processar, originando a necessidade de utilizar malhas grosseiras em que cada unidade elementar atinge facilmente dimensões da ordem das dezenas de metro cúbico. Por último, uma limitação menos visível mas igualmente significativa é o facto de não existir documentação adequada do programa, o que coloca obstáculos significativos à sua manutenção e desenvolvimento.

O desenvolvimento do *ParFludan* pretende dar resposta às limitações do *Fludan-RAS*, tirando partido de um novo paradigma computacional, o da computação paralela. Através da paralelização do programa, é possível distribuir a carga computacional e a ocupação de recursos por várias unidades computacionais (UC), em vez de concentrar tudo numa única, de forma a que o tempo total de execução das simulações seja reduzido e, simultaneamente, o programa seja capaz de lidar com malhas de maior dimensão. Esta estratégia é particularmente adequada ao contexto do LNEC, em que existe uma infraestrutura de computação paralela, nomeadamente o *cluster Medusa*, que pode ser aproveitada para executar as simulações (Coelho, Silva; 2013). Adicionalmente, a documentação do *ParFludan* será um contributo futuro para a sua utilização, generalização e manutenção.

Em resumo, o *ParFludan* foi criado resolver os problemas de tempo de execução e dimensão das malhas exibidos pelo *Fludan-RAS*, preservado a sua funcionalidade, em especial a modelação do comportamento viscoelástico de estruturas homogéneas de betão, solucionando a equação de equilíbrio na forma incremental no tempo, sendo cada passo resolvido em duas fases: uma primeira fase para contemplar a variação das ações no intervalo, admitindo que são aplicadas instantaneamente no intervalo – resposta instantânea – e uma segunda fase para fazer refletir a ação das cargas aplicadas durante o intervalo – resposta diferida. Tirando partido do novo paradigma de computação paralela, é possível minimizar as limitações do programa original e permitir o processamento de estruturas arbitrariamente grandes, com malhas arbitrariamente refinadas, em tempos de execução aceitáveis.

1.2 Condições iniciais

O desenvolvimento do *ParFludan* foi largamente baseado no *Fludan-RAS*, pelo que é importante apresentar com algum detalhe a sua estrutura e funcionalidade original, que foram em grande parte preservadas no *ParFludan*. A figura 1.2 descreve resumidamente o algoritmo computacional utilizado pelo *Fludan-RAS*, que foi também implementado no *ParFludan*, devidamente modificado para tirar partido da paralelização.

Este algoritmo compreende essencialmente duas fases:

1. Leitura dos ficheiros de entrada e preparação das estruturas de dados;
2. Execução da simulação.

Na primeira fase o programa lê os dados de entrada necessários para executar a simulação, nomeadamente a geometria da malha, as propriedades dos materiais, as condições de apoio e as ações a que a estrutura está sujeita, organizando essa informação nas estruturas de dados que serão utilizadas na fase de simulação. Esta informação está contida nos ficheiros de dados disponibilizados

pelo utilizador, de acordo com as especificações definidas na secção 5.2.1. Na segunda fase, o programa usa estes dados para executar a simulação do comportamento estrutural no domínio do tempo, com base no método dos elementos finitos e nos princípios descritos em (Piteira Gomes, 2008).

A diferença fundamental entre ambas estas fases é que a primeira é executada apenas uma vez, ao passo que a segunda é executada múltiplas vezes, uma vez que consiste num ciclo aos intervalos de tempo discretizados para simular a evolução temporal do comportamento da estrutura. Como consequência, a segunda fase é muito mais exigente do ponto de vista computacional.

Durante o ciclo temporal são tratadas duas fases em cada iteração, a fase instantânea e a fase diferida. A fase instantânea considera as variações das solicitações a que a estrutura está sujeita, como o peso próprio ou a pressão hidroestática, enquanto que a fase diferida serve para simular o efeito da história de carga. A alternância entre ambas as fases ao longo do tempo permite ao algoritmo simular o comportamento real da estrutura. Em ambas as fases, o programa monta uma matriz de rigidez K, cria um vetor de forças f, resolve o sistema $Kx=f$ para obter o vetor de deslocamentos x e calcula as restantes grandezas relevantes (tensões e extensões) a partir dos deslocamentos. Adicionalmente, considera ainda um modelo de dano, que só é ativado quando surge dano na estrutura – tipicamente, ao final de várias iterações no tempo – e consiste na resolução iterativa de múltiplos sistemas da forma $Kx=f$, com incremento de dano, até que o sistema atinja o equilíbrio.

Do ponto de vista computacional, o *bottleneck* deste algoritmo está na resolução iterativa do sistema de equações. A montagem da matriz de rigidez, que é feita em cada iteração, também contribui significativamente para o tempo de execução do programa, mas a resolução do sistema é o fator dominante. Nesse sentido, este *bottleneck* é especialmente agravado quando o modelo de dano é ativado, uma vez que consiste na resolução, em cada iteração no tempo, de múltiplas instâncias do sistema $Kx=f$.

Ainda que muito simplificada, esta análise permite traçar três linhas gerais a ter em conta no desenvolvimento do *ParFludan*:

- É necessário otimizar a resolução do sistema $Kx=f$, uma vez que é o fator que mais contribui para o tempo de execução do programa;
- É conveniente preparar a solução para utilizar os recursos de computação paralela disponíveis no LNEC, nomeadamente o *cluster Medusa*;

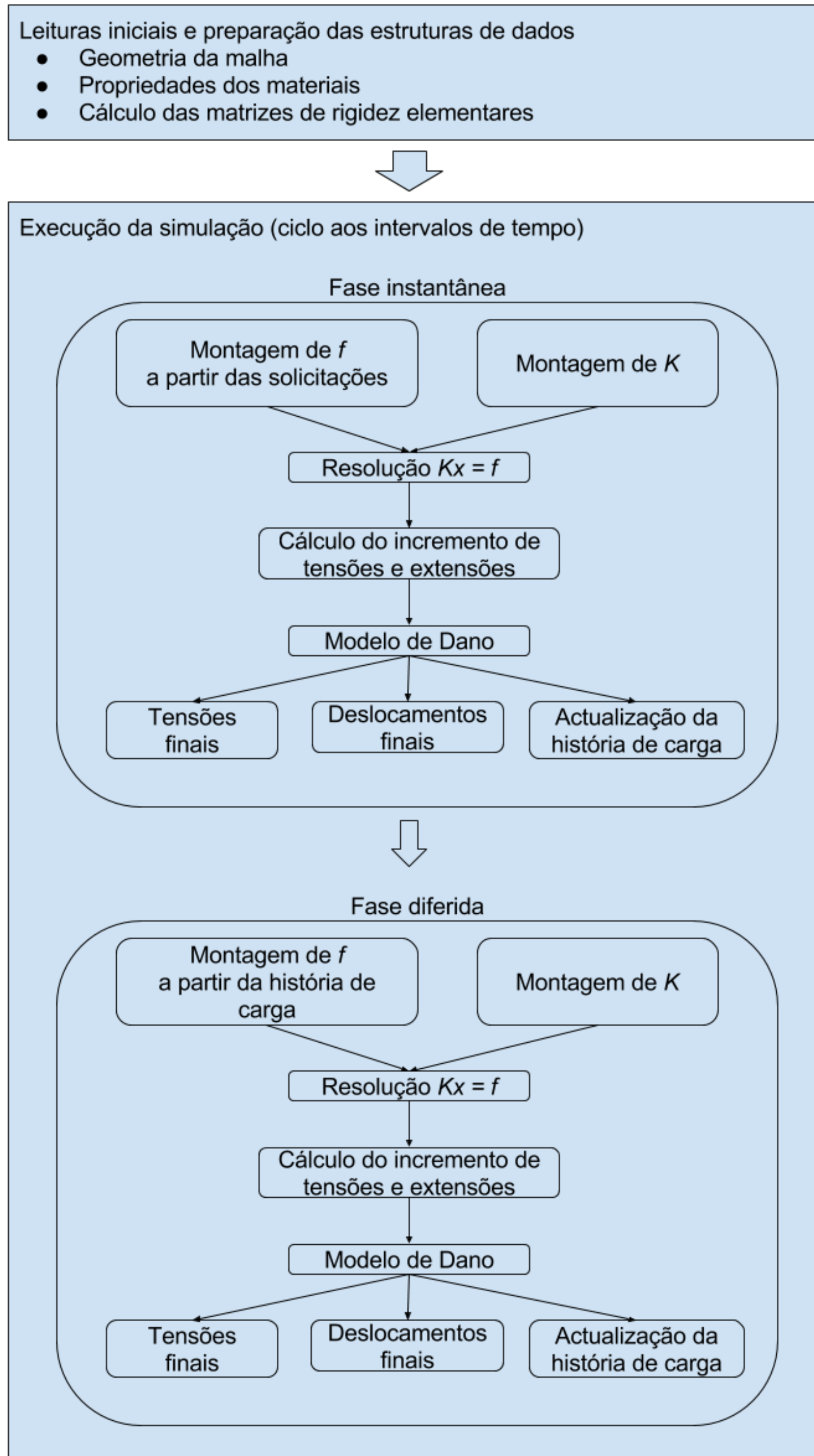


Figura 1.2 – Algoritmo computacional simplificado para o cálculo estrutural. Adaptado de (Piteira Gomes, 2008)

- É conveniente separar o programa em múltiplas partes independentes, seguindo o esquema lógico da figura 1.2, para que cada uma delas possa ser tratada de forma isolada.

1.3 Proposta de solução

Tendo em conta as particularidades do *Fludan-RAS* e as orientações retiradas da análise ao seu funcionamento, foi desenhada uma proposta de solução que permite dar resposta aos objetivos estabelecidos para o *ParFludan*.

No essencial, a estratégia adotada para esta solução baseou-se na paralelização da resolução do sistema de equações que constitui o principal *bottleneck* do *Fludan-RAS*, segundo o paradigma de memória distribuída, não só por permitir escalar o desempenho do programa como também por ser o mais adequado à utilização de um *cluster* como o *Medusa*; por sua vez, esta decisão implicou alterar a forma de preparação das estruturas de dados relevantes, de forma a serem adequadas ao processamento paralelo. Tendo em conta que para tirar benefício da paralelização todas as estruturas de dados envolvidas nas partes mais importantes do programa teriam de ser reorganizadas, esta estratégia acabou por implicar a reescrita integral do programa.

A necessidade de alterar a forma de armazenamento e, por conseguinte, de tratamento das estruturas de dados é evidente, por exemplo, no caso da matriz de rigidez *K*. Esta matriz tem dimensão $(N*3)^2$, onde *N* é o número de nós da malha; numa malha com 10 mil nós, a matriz de rigidez tem 9E+08 entradas, o que significa que, se cada elemento for representado por um tipo de dados *float* de 8 *bytes*, esta estrutura ocupa perto de 6,7GB de memória, o que rapidamente se torna incompatível com os limites de memória usuais se não for distribuída por várias UC. Desta forma, a definição das estruturas de dados limita o tamanho das malhas admissíveis pelo programa[1].

A nível tecnológico, a paralelização foi levada a cabo utilizando o modelo de comunicação padrão em computação distribuída, o *Message Passing Interface (MPI)*[2], através da implementação *OpenMPI* (Gabriel *et al*, 2004). Adicionalmente, uma vez identificada a resolução do sistema *Kx=f* como o principal *bottleneck*, tornou-se imperativo encontrar uma biblioteca paralela de computação numérica para esse efeito, tendo a escolha recaído sobre o *toolkit PETSc*[3] (BALAY *et al*, 2016) (BALAY, GROPP, MCINNES, *et al*; 1997). Para a escrita do código do programa, foi adotada a linguagem de programação C, por permitir uma interface mais harmoniosa com o *MPI* e o *PETSc* do que o *Fortran*, a linguagem utilizada no *Fludan-RAS*. Para controlo de versões foi adotado o *Git*[4], enquanto que os *script*s para integrar as diferentes componentes do programa e gerir as submissões para execução foram escritos em *bash* e *python*.

---

[1] É evidente que outros esquemas de armazenamento permitem reduzir o espaço em memória exigido, em particular no caso de matrizes esparsas; no entanto, para malhas suficientemente grandes, essa solução não é suficiente para resolver o problema.

[2] http://www.mpi-forum.org/

[3] https://www.mcs.anl.gov/petsc/

[4] https://git-scm.com/documentation

1.4 Estrutura do documento

Este relatório está estruturado em sete capítulos. No primeiro, de introdução, é descrita a motivação para a construção do *ParFludan*, as condições iniciais que serviram de base ao seu desenvolvimento e as considerações estratégicas e tecnológicas que deram origem à proposta de solução.

No segundo capítulo é detalhado o processo de desenvolvimento do *ParFludan*, sendo destacados os processos de integração das interfaces com os ficheiros de dados existentes, de paralelização, modularização e documentação do código.

O terceiro capítulo descreve a estrutura da solução obtida. O quarto capítulo detalha a avaliação da solução a nível de exatidão dos resultados e da qualidade da resposta ao desafio inicial.

O quinto capítulo contém instruções sobre a utilização do *ParFludan*, cuja funcionalidade pode ser customizada seguindo as recomendações incluídas no sexto capítulo.

Por último, o sétimo capítulo resume os pontos principais do relatório e elabora algumas considerações finais.

2 | Desenvolvimento do programa

Neste capítulo são descritas as linhas de desenvolvimento seguidas que, com base na estratégia e nas tecnologias definidas na secção 1.3, permitiram chegar à solução atualmente disponível. São detalhadas as abordagens à integração dos ficheiros de entrada e saída provenientes do *Fludan-RAS*, à paralelização do código, à modularização do código e finalmente à documentação do novo programa.

2.1 Integração de ficheiros de dados

Uma das primeiras decisões que foi necessário tomar durante o desenvolvimento do programa foi a de como lidar com os ficheiros de dados, tanto de entrada como de saída. No ambiente original do *Fludan-RAS*, os ficheiros de dados de entrada consistem maioritariamente em ficheiros de texto (em certos casos, também em ficheiros binários) que o programa lê integralmente na fase inicial da sua operação, conforme descrito na secção 1.2, criando e armazenando as estruturas de dados correspondentes para utilização ao longo do programa.

No contexto da paralelização, torna-se relevante distribuir as estruturas de dados correspondentes aos dados de entrada pelos recursos disponíveis, sendo necessário decidir entre ler todos os dados num único processador para depois os distribuir pelos diferentes processadores ou, em alternativa, ler em cada UC apenas a fração dos dados de entrada relevante para as operações que são realizadas por essa mesma UC. De um modo geral, a primeira opção é mais simples mas menos eficiente que a segunda, permitindo porém manter o formato original dos ficheiros sem necessidade de os adaptar para processamento paralelo[5]; como um dos objetivos do desenvolvimento foi manter o novo programa tão próximo quanto possível do original, optou-se por manter o formato dos ficheiros de entrada e recorrer à primeira opção, isto é, ler todos os ficheiros de dados na fase inicial do programa e, posteriormente, distribuir esses dados adequadamente pelos recursos a utilizar.

Já em relação aos ficheiros de saída, no programa original era criado um único ficheiro para cada execução, contendo os vários resultados relevantes - tipicamente deslocamentos, tensões, extensões e dano, nos vários nós, elementos ou pontos de gauss da malha. Isto tornava difícil a análise individual de um certo resultado, ou de uma certa fase temporal, bem como a comparação de resultados ao longo do tempo para diferentes execuções. Foi por isso tomada a decisão de criar não apenas um único ficheiro de saída por execução, mas um ficheiro por cada resultado (deslocamentos, tensões, extensões ou danos), iteração temporal e fase (instantânea ou diferida). Este formato discretizado permite facilmente comparar ou analisar individualmente cada resultado, mantendo-se a possibilidade de reverter ao formato original através da concatenação dos vários ficheiros de saída.

[5] A leitura de ficheiros de texto por vários processadores em simultâneo não é trivial.

A orientação central durante este processo de integração dos ficheiros de dados foi a de manter estes ficheiros tão próximos quanto possível da sua configuração original; idealmente, como resultado desta abordagem, um utilizador do *Fludan-RAS* dispondo de ficheiros de dados para um determinado cenário será capaz de executar o mesmo cenário no *ParFludan*, obtendo resultados num formato semelhante ao original. Os detalhes relativos aos formatos de cada tipo de ficheiro de dados estão disponíveis na secção 5.2.

2.2 Paralelização

Esta secção destina-se a descrever em detalhe o processo de paralelização e as decisões tomadas durante o seu curso. Recapitulando, a paralelização do programa original *Fludan-RAS*, segundo um paradigma de computação distribuída, foi identificada como a melhor solução para resolver os dois desafios colocados pelo *Fludan-RAS*: tempos de execução demasiado longos e incapacidade de tratar malhas de grande dimensão. Para reduzir o tempo de execução foram identificados os dois principais *bottlenecks* computacionais do programa, a resolução do sistema de equações e a montagem da matriz de rigidez; no sentido de aumentar a complexidade das malhas com que o programa é capaz de lidar, a distribuição das estruturas de dados pelas várias UC disponíveis foi reconhecida como a principal via a seguir.

2.2.1 Estruturas de dados globais e distribuídas

Durante o processo de paralelização do programa, em particular na criação das estruturas de dados, foi frequentemente necessário decidir criar estruturas de dados globais ou distribuídas. As estruturas de dados distribuídas são repartidas pelas várias UC, ficando cada uma apenas com uma parte da estrutura de dados original e não havendo duplicação da informação, pelo que em cada UC apenas é usada memória local na proporção da fracção da estrutura original que lhe está alocada.

Em contraste, uma estrutura global é replicada em cada processador e, portanto, requer tantas vezes mais memória quanto o número de processadores em uso; porém, a vantagem de ter estruturas de dados globais é que não é necessário haver comunicação entre processadores – visto que a informação está replicada em todos eles –, o que não só reduz a complexidade da programação como permite eliminar os tempos de comunicação (*overheads*), que limitam a eficiência do programa.

Atendendo a que o objetivo de reduzir o tempo de execução foi identificado como prioritário em relação ao de aumentar a complexidade das malhas aceitável, o processo da paralelização focou-se em minimizar os tempos de comunicação entre processadores em vez de minimizar a memória utilizada por cada processador. Isto levou a que na maioria dos casos fossem criadas estruturas de dados globais em vez de distribuídas, existindo portanto uma réplica em cada processador da estrutura de dados completa. Existem, contudo, estruturas de dados em que tal abordagem não é ou necessária, porque nem toda a sua informação é requerida por todos os processadores, ou exequível, por comprometer a capacidade do programa de lidar com malhas de maior dimensão – caso, por exemplo, da matriz de rigidez – e nesses casos foram criadas estruturas de dados distribuídas para armazenar essa informação.

Consequentemente, o programa contém estruturas de dados dos dois tipos:

- globais, ou seja, transversais a todas as UC, no caso de estruturas de dados de pequena dimensão e frequentemente utilizadas por vários processadores;
- distribuídas, ou seja, repartidas pelas várias UC sem duplicação de informação, no caso de estruturas de dados de grande dimensão e cujo processamento é normalmente feito em paralelo por todos os processadores.

Para lidar com estruturas de dados distribuídas, é importante definir partições do domínio de dados a que cada processador tem acesso.

2.2.2 Partições de domínios

A definição adequada de partições dos dados para cada processador é necessária para garantir que a carga computacional é distribuída aproximadamente de igual forma por todas as UC, e portanto que o tempo de execução é reduzido ao mínimo. O tipo de partições que definimos é diretamente influenciado pela utilização do método dos elementos finitos no programa principal; este método compreende dois tipos essenciais de operações: operações orientadas aos nós e operações orientadas aos elementos. É, portanto, conveniente definir uma partição em cada um desses domínios.

A partição nos nós é naturalmente assumida pelo *PETSc* e consiste unicamente em repartir o número de nós da malha pelo número de processadores disponíveis[6]. Para cada processador, é definido um número de nós locais e o intervalo de nós globais a que esse processador tem acesso, e que lhe são portanto locais. Por exemplo, para uma malha de 400 nós a ser executada em 8 processadores, são atribuídos ao primeiro processador os nós 1 a 50, ao segundo os nós 51 a 100, e assim sucessivamente. Esta partição é depois convenientemente utilizada sempre que uma operação orientada aos nós, como por exemplo a montagem da matriz de rigidez, é executada, restringindo a iteração de cada processador somente à sua partição local de nós e agregando no final os resultados de cada processador.

Por outro lado, a partição nos elementos é criada manualmente com base na minimização das comunicações entre processadores. Assim, cada elemento é associado ao processador que contenha mais nós que façam parte desse mesmo elemento, de tal forma que quando seja feita uma operação de iteração sobre todos os nós de cada elemento, a maioria desses nós residam no mesmo processador. Esta partição é utilizada nas operações orientadas aos elementos por meio de um filtro que indica a cada processador que apenas trabalhe sobre os elementos que lhe pertencem.

Adicionalmente, para cada processador é criada uma região-fantasma (*ghost region*) a que pertencem todos os nós ligados a elementos desse processador, mas não residentes nesse processador; por exemplo, se o elemento 10 contém 15 nós atribuídos ao processador 1, 2 nós atribuídos ao processador 2 e 3 nós atribuídos ao processador 4, então o elemento 10 pertencerá ao

---

[6] A rotina de *PETSc* usada é a *PetscSplitOwnershipBlock*.

processador 1 e a região-fantasma desse processador incluirá os 5 nós que estão ligado ao elemento 10 mas não estão atribuídos ao processador 1. O propósito desta estrutura é garantir que em cada processador exista uma cópia de todos os nós ligados aos elementos locais, mesmo que os próprios nós pertençam a outro processador; isto resulta na duplicação de alguma informação nodal, logo em maior consumo de memória, mas em contrapartida as comunicações entre processadores nas operações orientadas aos nós são eliminadas, reduzindo os tempos de execução.

2.2.3 Criação das estruturas de dados

A definição destas partições permite redefinir os algoritmos de criação das estruturas de dados necessárias de forma eficiente e em paralelo. No caso da criação da matriz de rigidez, que é um dos dois principais *bottlenecks* do programa, é criada uma estrutura paralelizada através da função do *PETSc MatCreateSBAIJ*, que é depois preenchida em paralelo reproduzindo o algoritmo inicial de montagem desta matriz, que envolve iterar sobre todos os nós de cada elemento da malha, mas limitando cada processador aos seus próprios elementos locais. Desta forma, conseguem-se dois benefícios:

- A carga computacional é repartida pelos vários processadores, reduzindo o tempo de execução;
- A estrutura de dados correspondente à matriz de rigidez passa a existir distribuída pelas várias UC disponíveis, reduzindo o espaço de memória necessário.

Uma estratégia idêntica é seguida no caso das restantes estruturas de dados que estão envolvidas nos principais cálculos computacionais. O vetor das forças *f*, por exemplo, também é uma estrutura paralelizada criada através de uma rotina especializada do *PETSc*, de tal forma que se encontra distribuída pelos vários processadores e preparada para ser utilizada pelas rotinas paralelas disponíveis na biblioteca. Tal como no caso da matriz de rigidez, também aqui a montagem deste vetor, que compreende a análise das várias solicitações presentes nos dados (ver secção 5.2.1), é feita em paralelo recorrendo às partições de domínio definidas.

A criação destas estruturas de dados paralelizadas é fundamental para permitir a aplicação das funções especializadas da biblioteca *PETSc*, como no caso da resolução do sistema $Kx=f$.

2.2.4 Resolução do sistema de equações

Tal como descrito na secção 1.2, a resolução do sistema $Kx=f$ é o principal *bottleneck* do programa e o foco principal da abordagem de paralelização proposta. Na verdade, utilizando o *PETSc*, esta operação é relativamente simples e limita-se a chamar a função apropriada (*KSPSolve*), passando-lhe como parâmetros as estruturas de dados relevantes e devidamente paralelizadas.

Esta etapa do programa resume-se, portanto, à criação adequada das estruturas de dados tal como descrito na secção 2.2.3, ficando a paralelização da resolução do sistema integralmente a cargo da biblioteca numérica utilizada.

Neste contexto, a escolha do método de resolução do sistema (escolha de pré-condicionador, escolha de algoritmo, métodos diretos ou iterativos, etc) pode ser facilmente modificada, bastando para isso alterar os parâmetros relevantes na chamada da função.

2.2.5 Implementação do modelo de dano

Finalmente, é importante mencionar a abordagem adotada no caso do modelo de dano, que é não só uma das principais funcionalidades do *ParFludan* como também um das principais etapas contribuintes para o tempo total de execução, a partir do momento em que é ativado.

Na verdade, em termos computacionais, o modelo de dano consiste na resolução iterativa de múltiplos sistemas de equações até que se atinja uma determinada condição de convergência, tipicamente expressa em termos das normas dos vetores das forças elástica e real.

Na prática, as técnicas adotadas para tornar a montagem da matriz de rigidez e a resolução do sistema principal de equações mais eficientes traduzem-se automaticamente na melhoria da eficiência deste processo. Por meio da paralelização da montagem da matriz de rigidez e da resolução do sistema, é então possível ao modelo de dano tirar vantagem dos recursos paralelos disponíveis, reduzindo o tempo de execução e aumentando a capacidade de lidar com malhas de maior dimensão, à semelhança da parte viscoelástica.

2.3 Modularização

Na sequência das orientações gerais delineadas na secção 1.2, é conveniente repartir o programa completo em unidades menos complexas que possam ser desenvolvidas, testadas e alteradas independentemente umas das outras e sem influenciar mais nenhuma componente do programa. Definimos este processo como a modularização do código, designando-se cada uma das unidades por módulo.

Concetualmente, um módulo é composto por duas estruturas: uma interface e uma implementação. A interface define os elementos requeridos e providenciados pelo módulo, enquanto a implementação corresponde ao código que implementa esses elementos de funcionalidade. Esta distinção é importante porque um módulo pode ter múltiplas implementações sem que isso interfira com a sua ligação aos restantes elementos do programa, desde que cada implementação respeite a interface. Em particular, esta abordagem permite que a funcionalidade do módulo seja alterada tendo apenas em atenção a interface definida, sem necessidade de rever a sua interação com o resto do programa.

Adicionalmente, o facto de cada módulo ser independente permite que a sua validação seja levada a cabo sem dependências externas associadas, o que é útil não só para isolar potenciais causas de erro como também para experimentar diferentes implementações da mesma funcionalidade. Na prática, esta propriedade é explorada através da definição de testes aos módulos, que permitem em qualquer momento verificar que a funcionalidade dos módulos é a esperada, independentemente do estado do resto do programa.

Com base nestas vantagens, foi decidido adotar para o *ParFludan* uma estratégia modular de desenvolvimento. A descrição detalhada da organização dos vários módulos do *ParFludan* e da sua interação está incluída na secção 3.2, e a definição de testes unitários aos módulos na secção 4.1.1.

2.4 Documentação

Por último, é importante descrever a abordagem adotada para documentar o funcionamento do programa, no sentido de facilitar a sua utilização, verificação e extensão no futuro. A documentação do *ParFludan* encontra-se desenvolvida a três níveis: código, repositório e publicações.

A documentação a nível do código consiste nos comentários estruturados ao funcionamento das funções e dos módulos definidos no programa. Cada função contém informação sobre o seu propósito, os seus dados de entrada, dados de saída, efeitos colaterais, variáveis globais utilizadas e comentários adicionais. O objetivo desta documentação é garantir que qualquer utilizador dispõe a cada momento de toda a informação necessária para utilizar uma determinada função. Do mesmo modo, cada módulo contém informação sobre o seu propósito e notas adicionais sobre o seu funcionamento.

A documentação a nível do repositório consiste em três ficheiros definidos na *home* do programa: *BUGS.md*, *INSTALL.md* e *README.md*. O primeiro ficheiro contém informação sobre *bugs* encontrados e eliminados durante o desenvolvimento, o segundo contém instruções detalhadas sobre como instalar o programa e o terceiro descreve brevemente como utilizar o programa. Estes ficheiros são relativamente pequenos e necessariamente incompletos, mas como são atualizados frequentemente constituem uma fonte dinâmica de informação prática relevante sobre o programa.

A documentação publicada consiste nos artigos e/ou relatórios que descrevem o desenvolvimento, a utilização ou outro aspeto relevante do funcionamento do programa. Estas são as fontes mais detalhadas de informação e as mais adequadas a um novo utilizador. Exemplos de documentação deste tipo são a publicação (Coelho, Silva, Piteira Gomes; 2014) e este próprio documento.

Este esforço de documentação não acompanhou o desenvolvimento do programa desde o início, pelo que existem várias funções que ainda não dispõem de documentação adequada; nesses casos, existem normalmente comentários no código, dentro das próprias funções, que explicam as partes menos claras da sua funcionalidade.

3 | Estrutura do programa

Este capítulo introduz o utilizador aos elementos essenciais da estrutura do programa, de forma a permitir-lhe utilizar e modificar o programa conforme as suas necessidades. O capítulo inclui uma visão geral da estrutura do programa, com a descrição do sistema de pastas em que o programa está organizado e de cada uma das suas componentes. Posteriormente descreve, em detalhe, a organização do programa em módulos e a forma de gestão das submissões de execuções através do *script run.py*.

No final do capítulo deverão ser claras as respostas às seguintes questões:

- Como está organizado o programa?
- Onde estão definidos os cenários para execução?
- Onde são colocados os resultados de uma execução?
- Onde estão definidos os cenários de teste?
- Onde e como são definidos os vários módulos do programa?
- Como são associadas e utilizadas as diferentes componentes do programa?

3.1 Visão geral

O programa está organizado em seis componentes, correspondendo a seis pastas diferentes, que funcionam em conjunto para suportar a funcionalidade completa do programa. A cada uma destas componentes corresponde uma subdiretoria do programa principal: *data*, *include*, *results*, *src*, *src_seq* e *tests*. A figura 3.1 apresenta um esquema resumido desta organização.

A utilização mais simples do programa exige apenas uma compreensão do funcionamento das componentes *data/*, *results/* e *src/*. Porém, a alteração do programa com vista a acrescentar funcionalidade ou casos de teste já requer uma compreensão mais detalhada de todas as componentes. Esta estrutura não é rígida e o programa suporta alterações à estrutura original – por exemplo, se o utilizador desejar ter duas pastas *data1/* e *data2/* de forma a separar cenários, poderá fazê-lo; o suporte para tal é dado através do módulo *run.py* e descrito na secção 5.4.2.

3.1.1 A pasta *data/*

A pasta *data/* inclui a definição dos cenários de execução do programa. Para cada cenário que o utiizador deseje ter disponível para executar, deve ser criada uma subpasta com o nome do cenário que contém todos os ficheiros de dados necessários para executar o cenário (a descrição dos ficheiros necessários para executar um cenário é feita na secção 5.2.1). A pasta *data/* pode conter um número ilimitado de subpastas relativas a diferentes cenários.

Considere-se como exemplo o cenário descrito na figura 3.1. Neste caso estão definidos dois cenários, *barragem1* e *barragem2*, que contêm os ficheiros de entrada relevantes e estão assim

disponíveis para serem invocados pelo programa para execução. Caso fosse necessário acrescentar um novo cenário, por exemplo *barragem3*, deveria ser criada uma nova pasta *barragem3/* dentro da pasta *data/*, contendo todos os ficheiros de dados necessários.

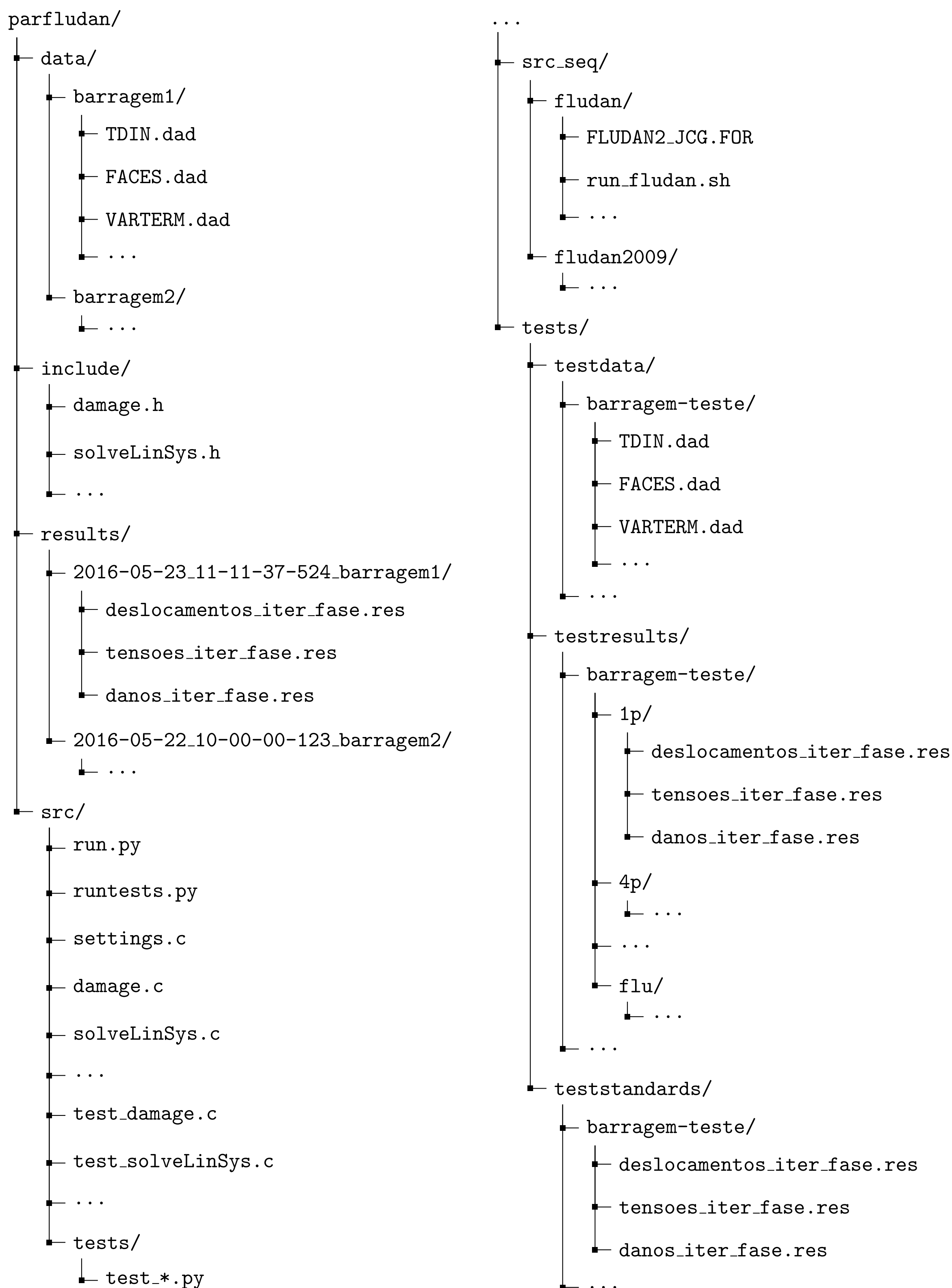


Figura 3.1 – Esquema resumido do programa

3.1.2 A pasta *include/*

A pasta *include/* é onde pertencem os *header files (.h)* referentes aos diferentes módulos do programa. Estes ficheiros contêm as declarações de funções e estruturas de dados que são depois implementadas no correspondente ficheiro de código (*.c*). Esta componente não é relevante quando se pretende apenas executar uma simulação e deve apenas ser considerada quando o utilizador pretender introduzir novos módulos no programa ou alterar os já existentes. A secção 3.2 descreve em detalhe a organização modular do programa.

3.1.3 A pasta *results/*

Esta pasta guarda os resultados das execuções do programa. Para cada simulação é criada uma nova subpasta com um nome único[7], para onde o programa copia os resultados da simulação logo que esta termina. O facto de os resultados de cada simulação serem guardados numa nova pasta permite que o utilizador faça múltiplas execuções, até de um mesmo cenário, sem se preocupar com sobreposição dos ficheiros de resultados e só no final trate da sua análise.

O formato e a organização dos ficheiros de saída guardados pelo programa são descritos em detalhe na secção 5.2.2.

3.1.4 A pasta *src/*

Na pasta src estão incluídos todos os ficheiros de código que constituem o programa, nomeadamente os módulos *bash* e *python* para controlar a execução do programa (em particular os *script*s *run.py*, *runtests.py* e *processing.sh*), os ficheiros de código (*.c*) que constituem os módulos do programa, os ficheiros de teste correspondentes a cada um desses módulos, *script*s *python* auxiliares que complementam a execução do programa (por exemplo, para comparação ou visualização de resultados) e ainda, dentro da subpasta *tests/*, os testes unitários aos *script*s e módulos auxiliares (ver secção 4.1.1).

Para utilizações simples, que envolvam apenas executar simulações, os únicos ficheiros a ter em conta são o *run.py* e o *settings.c*. O *script run.py* faz a gestão de submissões (ver secção 3.3), enquanto que o ficheiro *settings.c* contém algumas parametrizações do programa (ver secção 5.3). Para modificar ou alargar a funcionalidade do programa, torna-se necessário modificar os restantes ficheiros, embora esta interação deva, em princípio, estar limitada à adição, remoção ou alteração de módulos independentes (ver secção 3.2). No caso de ser preciso fazer alterações a outros ficheiros que não os módulos, a principal referência é a documentação contida nesses próprios ficheiros.

Como resultado de incluir as peças fundamentais do programa, esta pasta é a que mais frequentemente é modificada e como tal está significativamente menos bem organizada do que as restantes. Apesar de isto não comprometer a funcionalidade do programa, faz parte do plano de

---

[7] A unicidade é garantida pelo formato do nome: YYYY-MM-DD_HH-mm-ss-uuu_cenario.

desenvolvimento futuro do *ParFludan* uma reorganização desta componente de forma a separar mais claramente todas as partes que a compõem, em particular os módulos e respetivos testes.

3.1.5 A pasta *src_seq/*

Esta pasta contém versões funcionais do programa original, o *Fludan-RAS*. O propósito de manter o programa original disponível é comparar os resultados obtidos com o *ParFludan* e os originais; isto é feito sobretudo quando o utilizador executa os testes completos (ver secção 4.1.2), mas também é possível executar uma simulação apenas com a versão original.

Dentro da pasta src_seq existe uma subpasta para cada versão do programa original disponível(no exemplo da figura 3.1, as subpastas *fludan/* e *fludan2009/*). Em cada subpasta devem estar todos os ficheiros necessários para executar o programa original e ainda o *script run_fludan.sh*, através do qual é feita a ligação com o programa principal; detalhes sobre como funciona este *script* estão disponíveis dentro do próprio *script*.

A necessidade de haver várias versões do programa original tem a ver com o facto de nem todas as versões usadas como referência para a construção do programa terem a mesma funcionalidade, pelo que diferentes exemplos poderão ser compatíveis com uma das versões mas não nas outras; por omissão é usada a versão *fludan* e outras versões podem ser invocadas através do *script run.py*. Caso o utilizador pretenda adicionar uma versão do *Fludan-RAS*, basta criar uma nova subpasta em *src_seq/* com os ficheiros necessários e acrescentar uma cópia do *script run_fludan.sh*, modificando-o apropriadamente.

Visto que é usada apenas para validação de resultados, esta parte do programa não é relevante para o utlizador que pretenda apenas executar uma simulação com o *ParFludan*.

3.1.6 A pasta *tests/*

Aqui estão alojados os dados necessários para executar os testes de integração, isto é, os testes que simulam cenários com resultados conhecidos para verificar a exatidão do programa (ver secção 4.1.2). Tal como mostra a figura 3.1, a pasta *tests/* contém três subpastas: *testdata*, *testresults* e *teststandards*.

A pasta *testdata/* replica a estrutura da pasta *data/*, mas apenas para os cenários de exemplo; no exemplo da figura existe apenas um cenário de teste, *barragem-teste*, mas tal como no caso dos cenários para execução podem ser acrescentadas tantas novas instâncias quanto desejado (ver secção 6.1).

A pasta *testresults/* também é equivalente à pasta *results/*: cada vez que é executado um cenário de teste, os resultados são guardados numa nova subpasta criada no momento. No entanto, esta tem a particularidade de, no caso de estarem a ser executados os testes completos (através do *script runtests.py*, ver secção 4.1), os novos resultados serem guardados numa pasta com o nome do cenário apenas (por oposição ao formato *timestamp+nome* na pasta *results/*), eventualmente apagando resultados de testes anteriores. Além disso, resultados de execuções com diferente

número de processadores são guardadas em sub-subpastas diferentes de um mesmo cenário, de forma a permitir a comparação posterior dos resultados[8]. A figura 3.1 ilustra esta descrição.

É na pasta *testresults/* que são guardados os resultados da execução quando se utiliza a opção *–T* com o *script run.py* (ver secção 5.4.2); nesse caso, os resultados são simplesmente guardados numa subpasta de *testresults/* de forma análoga ao que acontece com os resultados de uma execução normal e a pasta *results*, tal como descrito na secção 3.1.3).

Finalmente, a pasta *teststandards/* contém, para cada cenário de teste disponível, os ficheiros de resultados correspondentes aos resultados de referência, obtidos a partir de uma execução considerada correta. Estes resultados são úteis como *baseline* para comparação com resultados posteriores, permitindo detetar se alterações feitas ao programa dão origem a resultados diferentes em cenários estabelecidos e considerados corretos. A descrição detalhada dos mecanismos de teste que usam esta informação é feita na secção 4.1.2.

3.2 Organização dos módulos

Na construção do programa procurou-se, a partir de determinado momento, seguir uma lógica modular de desenvolvimento (ver secção 2.3). Este tipo de abordagem consiste em dividir a funcionalidade do programa em pequenas partes, independentes entre si, de tal forma que cada uma delas contém toda a informação necessária para executar a sua função; cada uma destas pequenas partes constitui um módulo do programa. Em contraste com abordagens não-estruturadas, em que o código consiste num conjunto de instruções altamente interdependentes entre si, a modularização do código contribui para que este seja mais estruturado e como tal mais fácil de compreender e modificar.

O *ParFludan* contém tanto módulos que implementam funcionalidades que fazem parte do programa principal de análise de elementos finitos como módulos auxiliares que servem para complementar os resultados dessa análise. Esta secção descreve como estão organizados esses módulos.

3.2.1 Módulos do programa principal

Na linguagem principal do programa, cada módulo é definido por dois ficheiros – um ficheiro *.h*, que define a interface, e um ficheiro *.c*, que define a implementação. Estes dois ficheiros devem ter o mesmo nome, sendo o ficheiro *.h* guardado na pasta *include/* e o ficheiro *.c* na pasta *src/*.

O principal ficheiro de código do programa, *tf2.c*, inclui um único ficheiro de interface, o *header.h*, que por sua vez deve incluir todos os ficheiros de interface dos módulos que implementam

[8] O mesmo se passa caso seja executada a versão sequencial do programa, sendo nesse caso os resultados guardados na sub-subpasta *flu*.

funcionalidades relevantes para o programa principal. Desta forma, para que a funcionalidade de um determinado módulo fique disponível, basta incluir o respetivo ficheiro de interface em *header.h*[9].

Por exemplo, no *ParFludan* existe um módulo cuja funcionalidade é executar os cálculos de dano na estrutura, chamado *damage*; este módulo é implementado em dois ficheiros, o ficheiro de interface *damage.h*, na pasta *include/*, e o ficheiro de implementação *damage.c*, na pasta *src/*. Esta funcionalidade fica depois disponível ao acrescentar ao *header.h* a instrução

#include "damage.h"

É ainda importante explicar o papel dos ficheiros *header.h* e *header.c*. A lógica modular não foi implementada logo desde o início do desenvolvimento do programa, pelo que até um ponto relativamente avançado existia apenas um ficheiro, o *header.h*, complementar ao corpo principal do programa; este ficheiro incluía tanto as interfaces como as implementações de toda a funcionalidade do programa, o que rapidamente se tornou inviável. A partir do momento que se decidiu modularizar o código, o *header.h* foi restruturado em duas fases: em primeiro lugar as interfaces foram separadas das implementações, sendo as primeiras mantidas no *header.h* e as segundas incluídas no novo ficheiro *header.c*; em segundo lugar, o ficheiro foi sendo progressivamente desconstruído e separado noutros módulos, mais pequenos e autocontidos. Desta forma, o estado atual de ambos estes ficheiros é um resquício de estratégias anteriores já abandonadas e o reflexo de um esforço de modularização ainda não concluído; no cenário ideal, existirá apenas o ficheiro *header.h*, garantindo a interface com todos os módulos relevantes, e não será necessário qualquer ficheiro *header.c*.

3.2.2 Módulos auxiliares

Além dos módulos que compõem o programa principal, o *ParFludan* dispõe ainda de um conjunto de módulos auxiliares para complementar a execução do programa principal. Estes módulos são escritos em *python* ou *bash*, ao contrário dos módulos usados no programa principal, e tal como estes mantêm a caraterística de serem autocontidos e independentes. Todos os módulos auxiliares existem na pasta *src/* ou em suas subpastas, dependendo da sua complexidade.

Os módulos auxiliares disponíveis podem ser executados autonomamente, isto é, invocando-os explicitamente sem nenhuma ligação com o resto do programa, ou então podem ser executados em conjunto com o programa principal se forem invocados a partir do *script processing.sh*. A maioria destes módulos destina-se a ser usada na fase de pós-processamento dos resultados obtidos a partir da simulação[10].

Um exemplo de um módulo auxiliar é o módulo *vizmesh.py*, localizado na pasta *src/*, cuja função é criar um ficheiro de visualização dos resultados obtidos na simulação, em formato *VTK*[11], a partir do

[9] Na verdade é também necessário incluir o ficheiro de implementação no processo de compilação (ver secção 6.3).

[10] Faria também sentido incluir alguns módulos de pré-processamento para tratar de aspetos comuns a qualquer simulação, como o processamento dos dados iniciais; porém, isto ainda não foi implementado e este processamento é para já feito no programa principal.

[11] http://www.vtk.org/

ficheiro de dados inicial, que descreve a malha, e de um ficheiro de resultados. Este componente é útil na fase de pós-processamento e pode ser invocada tanto no *flow* do programa como isoladamente, para qualquer par *(malha,resultados)* disponível. A descrição completa de todos os módulos auxiliares usados no *ParFludan* será o foco de um relatório futuro.

3.3 Gestão das execuções

A existência e interação de todas estas componentes resulta num programa relativamente complexo, pelo que se torna necessário manter uma forma simples de utilizar o programa para as funções mais comuns. Essa interface simples é garantida pelo módulo *run.py*, que encapsula as diversas componentes do programa e providencia uma maneira fácil de executar o programa em diferentes condições. Os detalhes de como usar o *run.py* para executar uma simulação são descritos na secção 5.4; aqui, pretende-se apenas dar uma visão geral da funcionalidade deste módulo e do seu papel na estrutura do programa.

Este *script* é responsável pelo pré-processamento dos dados até à submissão do programa para execução, e pela própria submissão. Na fase de pré-processamento este *script* garante automaticamente um conjunto de operações necessárias ao bom funcionamento do programa, como a definição da estrutura de pastas a ser considerada, incluindo eventuais alterações à original tal como indicadas pelo utilizador, a criação da nova pasta de resultados para a execução específica ou a verificação dos requisitos necessários para o programa executar sem problemas.

Na fase de submissão, o *script* lê em que plataforma está a ser executado (por exemplo, se num computador local ou no *cluster Medusa*) e chama o *script processing.sh* adequadamente, tendo em conta as definições estabelecidas na fase de pré-processamento (e.g. a estrutura de pastas ou a ativação do modo verboso). O *script processing.sh* é, depois de submetido, responsável pela compilação e execução do programa principal, bem como pela fase de pós-processamento.

O ponto central é que, desta forma, o programa é usado sempre da mesma forma, independentemente de onde esteja a ser executado. No entanto, é natural que a certa altura se torne desejável ajustar a estrutura dos ficheiros às necessidades específicas de cada utilizador e não ficar preso a uma certa configuração; um exemplo disso seria, por exemplo, usar a *pasta data_2016/* como fonte dos cenários de simulação em vez da pré-definida pasta *data/*. Para permitir essa adaptabilidade o módulo *run.py* admite que um conjunto de parâmetros sejam redefinidos quando o programa é chamado através de *flags*, sobrepondo-se aos valores pré-definidos; neste exemplo, fazer essa alteração seria o equivalente a chamar o módulo como:

$ python run.py --datafolder data_2016/

É através da configuração destes parâmetros que o programa suporta diferentes funcionalidades como múltiplas fontes de dados, a definição do número de processadores a usar no processamento paralelo ou a execução do programa sequencial original (*Fludan-RAS*). Uma descrição mais detalhada destes parâmetros está disponível na secção 5.4.2; para consultar todas as opções disponíveis, sugere-se a consulta da documentação do próprio *script run.py*.

4 | Avaliação do programa

Para verificar a qualidade do programa é essencial fazer a sua avaliação, que compreende duas partes: a avaliação funcional, em que o objetivo é garantir que os resultados que o programa produz estão corretos, e a avaliação não-funcional, em que o objetivo é analisar o comportamento do programa (por exemplo, quanto tempo demora a executar, que memória ocupa, etc.). Esta secção foca-se na avaliação funcional do programa, uma vez que foi a que se considerou prioritária no processo de desenvolvimento.

O objetivo principal desta avaliação foi assegurar a exatidão dos resultados obtidos pelo programa. Embora a definição de "correto" não seja absolutamente precisa, uma vez que o desempenho do método dos elementos finitos depende de fatores externos ao programa, tais como a definição da malha ou o hardware em que o programa é executado, é imprescindível definir uma forma razoável de avaliar os resultados, sendo a forma encontrada para o fazer descrita neste capítulo. Além da exatidão dos resultados, é também importante compreender de que forma o novo programa dá resposta às necessidades que motivaram originalmente o seu desenvolvimento; a avaliação da solução desenvolvida sob esse ponto de vista é também feita neste capítulo.

É expetável que no final do capítulo sejam claras as respostas às seguintes questões:

- Como se garante que cada função/módulo funciona corretamente?
- Como se garante que os resultados obtidos pelo programa estão certos?
- Como se garante que uma alteração do código não interfere com a funcionalidade do programa já desenvolvida?
- Como é que a nova solução se compara com a solução original?
- Quais são os limites da nova solução?

4.1 Exatidão dos resultados

O primeiro passo para possibilitar a utilização do *ParFludan* é garantir a exatidão dos seus resultados, através de testes funcionais. Essa tarefa nem sempre é trivial, uma vez que em muitos dos casos que se pretende estudar não existe termo de comparação disponível e a análise dos resultados tem de ser feita exclusivamente com base no conhecimento e experiência dos engenheiros de barragens. Ainda assim, foi definida uma estratégia de teste do programa baseada na ideia de considerar casos simples, aumentando progressivamente a sua complexidade, admitindo que um conjunto de peças simples garantidamente funcionais resulta numa peça mais complexa mas igualmente funcional.

No campo dos testes funcionais foram desenvolvidos e implementados dois tipos de testes: testes unitários e testes de integração. Os testes unitários destinam-se a avaliar a funcionalidade de cada peça do programa, isto é, cada função ou módulo. Por sua vez, os testes de integração destinam-se a avaliar a funcionalidade do programa completo, isto é, do conjunto das múltiplas peças, perante

diferentes cenários de simulação; de entre estes, são considerados tanto cenários teóricos, cujos resultados expetáveis podem ser consultados na literatura, como cenários reais cujos resultados são conhecidos por via numérica, quando disponível, ou experimental.

4.1.1 Testes unitários

Os testes unitários são usados para testar cada componente do programa durante o processo de desenvolvimento, com o propósito de verificar o comportamento dessa componente específica, antes de ser reunida com as restantes componentes para formar o programa completo. Estes testes devem ser tão simples quanto possível e independentes entre si e de outras componentes do programa, de forma a que seja possível executar os testes de uma componente em particular sem quaisquer dependências de outras componentes.

A garantia de que os testes unitários definidos para uma certa componente estão corretos é da responsabilidade do programador dessa componente, que conhece o seu funcionamento expetável. É boa prática de programação definir pelo menos um pequeno conjunto de testes unitários para cada peça do programa que se está a construir. Uma vez disponível, o conjunto de testes unitários deve ser executado sempre que é feita uma alteração no código ou uma compilação para execução; desta forma, é garantido que as alterações introduzidas não produziram efeitos imprevistos.

Como exemplo, consideremos a função que soma dois números *a* e *b*, divide por um terceiro *c* e retorna o resultado, f(a,b,c)=a+bc. Como testes unitários para esta função poderiam ser definidas as seguintes asserções:

- f(1,2,3)==1
- f(-1,2,5)==0.2
- f(10,-10,8218)==0
- f(1,2,0)==ERRO (indeterminação)

Ao serem executados o testes unitários, estas igualdades seriam testadas e caso não fossem verificadas o programa altertaria o utilizador para o facto de alguns dos testes unitários estarem a falhar.

No contexto do *ParFludan*, os testes unitários estão definidos dentro da pasta *src/*. No caso dos módulos do programa principal, para cada módulo existe um ficheiro *.c* com o mesmo nome acrescentado do prefixo "test_", onde estão definidos os seus testes unitários. O *script runCtests.sh* reúne e executa todos estes testes, falhando com uma mensagem de erro caso algum dos testes não passe; este *script* é executado sempre que o *script runtests.py* é chamado (ver secção 5.5) e pode também ser executado isoladamente.

No caso dos módulos auxiliares, os testes estão definidos dentro da pasta *src/tests.* Estes testes são também executados quando o *script runtests.py* é chamado, e para executar todos os testes aos módulos auxiliares (e, mais geralmente, todos os testes a módulos *python*), basta executar na pasta *src/* o comando

$ python –m unittest discover[12]

É possível e desejável que o próprio utilizador acrescente testes unitários, em especial no caso de desenvolver uma nova funcionalidade para o programa; os detalhes sobre como o fazer estão disponíveis na secção 6.1.

4.1.2 Testes de integração

Mesmo que todas as componentes do programa tenham testes unitários bem definidos e que todos eles passem sem erros, não é garantido que o conjunto das várias componentes, uma vez reunidas, funcione da maneira esperada. Para fazer essa verificação é necessário definir testes de integração, onde o programa completo é submetido a um cenário de execução e os resultados obtidos são comparados com os esperados. Os testes de integração são na verdade o principal teste à funcionalidade do programa e uma etapa indispensável no processo de verificar a integridade dos resultados obtidos.

Para definir um teste de integração são necessárias duas partes: por um lado, é preciso definir um cenário de execução correspondente aos dados de entrada para o programa; por outro, é necessário dar alguma medida dos resultados esperados de forma a que se possa verificar a exatidão dos resultados obtidos. No âmbito do *ParFludan* são distinguidos dois tipos de testes de integração, considerando casos teóricos e casos reais. Os casos teóricos correspondem a cenários ideais, descritos tipicamente na literatura de engenharia de estruturas, onde tanto os dados de entrada (a malha, as ações, etc) como os resultados (deslocamentos, tensões, etc) estão definidos a nível teórico. Os casos reais correspondem a cenários não-ideais, tipicamente provenientes de estruturas estudadas no âmbito de outros trabalhos, onde os dados de entrada estão bem definidos mas os resultados, em vez de terem origem teórica, provêm de dados experimentais ou de simulações do mesmo cenário feitas previamente com algum outro programa, e dos quais existe consequentemente um conhecimento menos exato do que nos casos com soluções analíticas.

Os cenários para testes de integração são definidos na pasta *tests/testdata*, que replica a estrutura da pasta *data/*, isto é, para cada cenário é necessário criar uma pasta em *tests/testdata* com o nome do cenário e incluir todos os ficheiros necessários para o executar, tal como se fosse um cenário normal para execução. A nível do programa não é feita qualquer distinção entre casos teóricos e casos reais.

Quando é definido um cenário para teste de integração, devem também ser definidos na pasta *tests/teststandards/* os resultados esperados caso a simulação decorra como previsto. Para cada cenário existe uma subpasta dentro desta com o nome do cenário e dentro da qual estão os resultados "corretos", ou pelo menos esperados, na mesma forma em que vão ser produzidos pela simulação, para que depois possam ser comparados.

Para executar os testes de integração existem duas hipóteses: usando o *script run.py* ou o *script runtests.py*. Por um lado, a forma mais simples é usar o *script run.py* para executar um dos cenários

[12] Esta funcionalidade é uma consequência dos testes serem definidos com o nome "test_*".

exatamente da mesma forma que se faria se fosse um cenário real e não um teste – a única particularidade a ter em conta é que deve ser escolhida a pasta *tests/testdata* como fonte de dados, em vez da pasta *data/* definida por omissão (ver secção 5.4).

Por outro lado, para executar um cenário e comparar os resultados obtidos com os resultados de referência deve ser usado o *script runtests.py*, que inclui três operações:

1. Execução da simulação com o *ParFludan*;
2. Execução da simulação com o *Fludan-RAS* (sequencial);
3. Comparação dos resultados das várias simulações entre si e com os de referência.

O primeiro passo é equivalente a executar o *run.py*, com a vantagem de que pode ser chamado múltiplas vezes para diferente número de processadores – o que é importante para garantir que os resultados são consistentes entre si independentemente do número de processadores usado. O segundo passo corresponde a executar a simulação com uma versão do programa sequencial original (disponível em *src_seq/*), em vez do *ParFludan*; a definição de que versão deve ser executada para cada cenário, se alguma[13], é feita dentro do *script runtests.py*, sendo usada por omissão a versão disponível em *src_seq/fludan/*. Por último, o *runtests.py* indexa os resultados criados nos passos 1 e 2 e compara-os entre si e com os resultados de referência definidos (caso existam), lançando avisos e eventualmente abortando a execução caso as diferenças encontradas excedam a tolerância definida[14].

Além dos cenários de teste disponíveis, o utilizador pode adicionar novos cenários para verificar novas funcionalidades ou simplesmente diversificar o conjunto de testes (ver secção 6.1). Por omissão o *runtests.py* executa todos os testes de integração disponíveis, mas é possível restringir a execução apenas a um subconjunto dos cenários existentes. A secção 5.5 descreve os detalhes de como utilizar esta e outras funcionalidades do *script runtests.py*.

4.2 Avaliação da solução

Além da validação dos resultados obtidos com o *ParFludan*, é importante avaliar se a nova solução responde adequadamente aos problemas que motivaram o seu desenvolvimento, nomeadamente a necessidade de reduzir o tempo de simulação de uma execução e de processar malhas de maior dimensão relativamente às capacidades do *Fludan-RAS*.

Relativamente ao processamento de malhas mais complexas, o *ParFludan* revela-se capaz de lidar com malhas de dimensões impossíveis para o *Fludan-RAS*; um exemplo é a malha disponível para o cenário da barragem de Peti, com cerca de 13 mil nós (o *Fludan-RAS* não é capaz de lidar com esta malha, mas o *ParFludan* executa corretamente). Esta nova capacidade surge como resultado da

[13] Em certos casos o programa sequencial pode não ser capaz de lidar com o cenário definido e como tal não interessa incluí-lo no processo de testes, como acontece no caso de malhas de grandes dimensões. Neste caso o processo de teste resume-se a comparar os resultados das execuções do *ParFludan*, eventualmente para vários processadores, entre si e com os resultados de referência.

[14] As tolerâncias para comparação de resultados podem ser definidas no próprio *script runtests.py*.

distribuição das estruturas de dados do programa por vários processadores, o que permite evitar as limitações de memória associadas a um único processador. Neste aspeto, o *ParFludan* responde positivamente ao desafio original e permite alargar a aplicabilidade do *Fludan-RAS* para malhas de maior dimensão, tal como pretendido.

No que diz respeito ao desempenho do programa, o *ParFludan* também permite acelerar a execução das simulações. Em (Coelho, Silva, Piteira Gomes; 2014) são reportados *speedups* de até 4 vezes, mesmo considerando como referência o *ParFludan* para 1 processador, e não o *Fludan-RAS* (caso contrário, é expetável que fossem registados *speedups* ainda maiores). Em termos absolutos, cenários que no *Fludan-RAS* demoravam na ordem das semanas a executar podem agora ser executados em dezenas de horas. Este aumento do desempenho é sobretudo consequência da paralelização implementada, em particular da resolução em paralelo do sistema de equações principal. Também neste aspeto o *ParFludan* responde adequadamente ao objetivo inicial de reduzir significativamente o tempo de execução.

5 | Como utilizar

Esta secção visa explicar como utilizar o programa, nomeadamente como funciona o processo de instalação, quais os ficheiros de entrada que o programa espera, quais os ficheiros resultantes da sua execução, qual o papel do ficheiro de parametrizações, qual o processo para executar uma simulação e como podem ser executados os testes disponíveis.

O projeto está alojado no repositório do LNEC e pode ser acedido através do link *https://sourcerepo.lnec.pt/parfludan/parfludan* por qualquer utilizador com permissão. Os ficheiros *INSTALL.md* e *README.md*, disponíveis na *home* do projeto, contêm instruções detalhadas sobre como instalar e como utiizar o programa, que podem servir como complemento a este relatório.

É usada como referência a versão v1.0 do programa, de 30 de junho de 2016. Em versões subsequentes, detalhes como as dependências do programa, as instruções precisas de instalação ou a forma de executar um cálculo podem ser alteradas. Os ficheiros *INSTALL.md* e *README.md* serão atualizados à medida que o projeto evolua, pelo que serão a referência caso exista uma versão mais recente que a v1.0.

5.1 Instalação

A instalação do programa está dividida em duas partes. Em primeiro lugar é necessário garantir que as dependências do programa estão disponíveis, e em segundo lugar é preciso fazer o *download* do programa propriamente dito. As instalações tanto das dependências como do programa devem ser feitas a partir do local onde se pretende executar o programa; por exemplo, no caso de o utilizador pretender usar o *ParFludan* no *Medusa*, então estas instalações devem ser feitas a partir da sua área pessoal no *Medusa*.

5.1.1 Dependências

Para funcionar corretamente, o programa precisa de utilizar outros programas dos quais depende. Antes de instalar o programa, é necessário garantir que estas dependências estão disponíveis. O ficheiro *README.md* contém a lista das dependências do programa, enquanto que o ficheiro *INSTALL.md* contém instruções detalhadas sobre como instalar cada uma destas dependências.

No caso de uma instalação no *cluster Medusa*, as dependências encontram-se automaticamente satisfeitas e basta seguir as instruções para instalação do programa (ver secção 5.1.2). É porém necessário garantir que o utilizador dispõe de algum programa de transferência de ficheiros no seu computador local (como o *Filezilla*[15] ou o *WinSCP*[16]), de forma a poder enviar e extrair ficheiros do *Medusa*.

[15] https://filezilla-project.org/

5.1.2 Programa

O programa é instalado simplesmente fazendo um clone do repositório. É recomendado que seja feito apenas o clone da última versão do programa, para evitar o download de ficheiros históricos irrelevantes. Para isso, basta executar o comando

$ git clone --depth=1 https://nome_de_utilizador@sourcerepo.lnec.pt/parfludan/parfludan.git[17]

Para testar se a instalação foi concluída com sucesso, basta executar os testes disponíveis (ver secção 5.5).

Caso surjam dificuldades durante este processo, os ficheiros *README.md* e *INSTALL.md*, disponíveis no repositório, contêm instruções mais detalhadas e atualizadas que poderão ser úteis.

Concluídos estes passos, o programa estará pronto a ser utilizado.

5.2 Interfaces

Para executar uma simulação com o *ParFludan* é necessário fornecer ao programa os dados de entrada que definem o cenário de simulação; da mesma forma, no final da execução o programa irá produzir ficheiros de saída que deverão corresponder aos resultados da simulação que são úteis para o utilizador. Esta secção define o tipo e formato tanto dos dados de entrada esperados pelo programa como dos resultados por ele gerados.

5.2.1 Ficheiros de entrada

Uma parte essencial ao bom funcionamento de qualquer programa de *software* é a definição precisa e sem ambiguidades dos tipos e formatos de dados com que está preparado para lidar. Uma das principais causas de erros na execução do programa, muitas vezes difíceis de descobrir e corrigir, advém da tentativa de lhe passar dados num formato diferente do esperado. Em geral, é desejável que um programa seja flexível o suficiente para lidar com algumas variações nos tipos de dados, lançando avisos ao utilizador ou mesmo estando preparado para lidar com pequenos desvios do formato base. Não obstante, é da responsabilidade do utilizador garantir que os dados usados no programa estão de acordo com as especificações definidas.

No desenvolvimento do *ParFludan*, foi decidido manter a nomenclatura e o formato original dos ficheiros de entrada, tal como usados no *Fludan-RAS*. Por conveniência, é aqui feita uma descrição resumida dos formatos de cada um deles e das circunstâncias em que devem ser definidos. A tabela 5.1 resume os cinco tipos de ficheiro admitidos pelo *ParFludan*. Em anexo podem ser encontrados exemplos simplificados de cada um destes ficheiros.

---

[16] https://winscp.net/

[17] Em alternativa a *https*, pode ser feito o clone através de *ssh*.

Tabela 5.1 – Tipos de ficheiros de entrada admitidos

Nome	Tipo de dados	Quando necessário?
TDIN.DAD	Alfanumérico	Sempre
FACESM.DAD	Alfanumérico	Quando existe pressão hidroestática
VARTERM.DAD	Alfanumérico	Quando existem variações de temperatura
EXPAPG.BIN	Binário	Quando existem deformações impostas
RIGFUND.BIN	Binário	Quando a fundação é considerada separadamente

Os nomes presentes na tabela 5.1 são aqueles que o programa utiliza por omissão, e devem ser mantidos independentemente do cenário a considerar; é também possível usar outros nomes para estes ficheiros, mas nesse caso é necessário indicá-lo explicitamente quando o *script run.py* é chamado (ver secção 5.4). É ainda importante notar que no caso dos ficheiros de entrada todos os índices são indexados a 1, isto é, a numeração dos nós e elementos começa em 1.

5.2.1.1 O ficheiro de dados TDIN.DAD

O ficheiro *TDIN.DAD* é o principal ficheiro de dados utilizado pelo *ParFludan*, tendo sempre de estar presente independentemente das especificidades do cenário a executar. Neste ficheiro está incluída a informação sobre a malha, as ações a que a estrutura está sujeita e as caraterísticas dos seus materiais. Está dividido em 6 partes:

1. Cabeçalho;
2. Propriedades da malha;
3. Coordenadas dos nós;
4. Incidências;
5. Apoios;
6. Solicitações.

O cabeçalho consiste apenas em três linhas que identificam o ficheiro de dados em questão, contendo tipicamente o nome da barragem, o tipo de elementos finitos considerados e outras informações relevantes.

Na parte 2 estão definidas informações importantes sobre a propriedades da malha, tais como número de nós e elementos a considerar, e sobre a estrutura propriamente dita, tais como a informação temporal relevante (para determinar a idade do betão e o início e fim do período em análise) e as propriedades dos materiais da estrutura.

Nas partes 3 e 4 é feito o essencial da definição da malha. Na parte 3 são definidas as coordenadas dos pontos nodais, através de uma lista com tantas linhas quanto o número de nós indicado no cabeçalho, tendo cada linha quatro valores no formato *x y z id*, em que *x*, *y* e *z* (números reais) são as coordenadas do ponto *id* (número inteiro).

Na parte 4 são definidas as incidências da malha, isto é, os pontos nodais que constituem cada elemento, bem como o tipo de cada elemento. Cada linha deve ter o seguinte formato:

nó_1 nó_2 nó_3 .... nó_19 nó_20 tipo_elemento id_elemento

Nesta parte devem existir tantas linhas quanto o número de elementos da malha, cada uma com 22 valores[18].

Na parte 5 são descritos os apoios da estrutura. O *ParFludan* admite dois tipos de apoios: apoios rígidos ou apoios de inserção. Os apoios rígidos correspondem a fixar alguns dos nós da estrutura, correspondentes à fundação, em uma ou mais direções, enquanto que os apoios de inserção correspondem a considerar a fundação como um único elemento finito não estritamente fixo mas antes com uma rigidez finita.

No caso de apoios rígidos, a parte 5 do ficheiro *TDIN.DAD* consiste numa lista de formato *id Cx Cy Cz,* em que *id* é o número do nó definido como fixo e *Cx, Cy, Cz* são as várias direções segundo as quais o nó pode ser definido como fixo (valor 1) ou livre (valor 0). Esta lista deve ter o número de linhas correspondente ao número de apoios rígidos definidos no cabeçalho.

Já no caso dos apoios de inserção, esta parte consiste numa primeira linha com um único valor, correspondente ao número de nós de inserção considerados, seguida de tantas linhas quanto necessário para incluir os números de todos os nós definidos como de inserção. A utilização de apoios de inserção pressupõe igualmente a utilização do ficheiro *RIGFUND.BIN* para dar os valores da rigidez dos nós de inserção.

Finalmente, a parte 6 do ficheiro *TDIN.DAD* diz respeito à definição das solicitações, isto é, do conjunto de ações a que a estrutura é sujeita ao longo do período de simulação. A primeira linha nesta secção serve para definir quais as solicitações ativas e toma a seguinte forma:

pp ph vt di fcn fce fd assap

Cada uma destas variáveis é binária e indica se uma solicitação de certo tipo deve ou não ser considerada no cálculo. Adicionalmente, a computação das solicitações exige um conjunto de dados adicionais para especificar a contribuição de cada ação para o vetor das forças em cada iteração, que podem ser incluídos no *TDIN.DAD*, especificados à parte através de um ficheiro de dados auxiliar ou ambos. A tabela 5.2 contém a chave de cada um destes termos e os dados e ficheiros adicionais que cada solicitação requer.

A seguir à primeira linha, é necessário descrever a contribuição de cada ação para o vetor das forças, em cada iteração. Na prática isto significa que a cada ação corresponde um bloco de dados e para cada iteração os blocos de dados de cada ação devem ser concatenados – necessariamente, pela ordem da tabela 5.2 – e escritos no ficheiro de dados.

---

[18] Consequência da única geometria de elementos finitos considerada ser a de hexaedro de 20 nós.

Tabela 5.2 – Designações e caraterísticas das várias solicitações disponíveis

Designação	Ação que representa	Dados adicionais	Ficheiro auxiliar
pp	Peso próprio	1. Multiplicador	-
ph	Pressão hidroestática	1. Peso específico da água 2. Cota da água 3. Nº de faces sujeitas a PH 4. Faces sujeitas a PH (via *FACESM.DAD*)	*FACESM.DAD*
vt	Variações térmicas	1. Nº de nós com variações de temperatura 2. Valor das variações de temperatura (via *VARTERM.DAD*)	*VARTERM.DAD*
di	Deformações impostas	1. Nº de elementos com deformações impostas 2. Valor das deformações impostas (via *EXPAPG.BIN*)	*EXPAPG.BIN*
fcn	Forças concentradas nos nós	1. Nº de nós com forças concentradas 2. Vetor das forças concentradas nodais	-
fce, fd, assap	Não consideradas (preservadas apenas para referência)		

A seguir à primeira linha, é necessário descrever a contribuição de cada ação para o vetor das forças, em cada iteração. Na prática isto significa que a cada ação corresponde um bloco de dados e para cada iteração os blocos de dados de cada ação devem ser concatenados – necessariamente, pela ordem da tabela 5.2 – e escritos no ficheiro de dados.

5.2.1.2 Ficheiros de dados auxiliares

Além do *TDIN.DAD* o *ParFludan* está preparado para ler outros quatro tipos de ficheiros: *FACESM.DAD*, *VARTERM.DAD*, *EXPAPG.BIN* e *RIGFUND.BIN*. Estes ficheiros só são necessários em casos específicos de utilização e não são tão gerais quanto o *TDIN.DAD*, mas são essenciais para conseguir tratar todas os cenários relevantes.

O ficheiro *FACESM.DAD* é necessário sempre que a ação da pressão hidroestática é considerada. É um ficheiro simples, apenas com dados alfanuméricos, com as seguintes caraterísticas:

- A primeira linha deve ter apenas um número inteiro, correspondente ao número de faces sujeitas a pressão hidroestática;

Por sua vez, o ficheiro *VARTERM.DAD* é necessário quando existem variações por ação da temperatura. É também um ficheiro estritamente alfanumérico com as seguintes caraterísticas:

- É constituído por um conjunto contíguo de *M* blocos, em que *M* é o número de iterações no tempo consideradas;
- Para cada bloco, a primeira linha deve ser a data correspondente à iteração representada nesse bloco, no formato "Ano mês dia" (e.g. 2014 1 1);
- As restantes linhas de cada bloco, devem ser constituídas por pares de números, em que o primeiro é um inteiro correspondente ao número do nó sujeito a variações térmicas e o segundo é um valor real que representa a variação de temperatura nesse nó para essa iteração.

No total este ficheiro deve ter *M*(N+1)* linhas, sendo *N* o número de nós sujeitos a variações de temperatura.

O ficheiro *EXPAPG.BIN* é necessário sempre que se considerem deformações impostas, nomeadamente as devidas a reações expansivas do betão. É um ficheiro binário, com as seguintes caraterísticas:

- É constituído por uma sequência de números reais, escritos em formato binário, cada um com 8 *bytes*;
- A orientação dos números segue o formato *IT * N * G*, sendo *IT* o número de iterações no tempo, *N* o número de elementos sujeitos a deformações impostas e *G* o número de pontos de gauss por elemento. Ou seja:
  - O valor correspondente ao ponto de gauss 1 do elemento 1 na iteração 1 está na posição *1* do ficheiro;
  - O valor correspondente ao ponto de gauss 1 do elemento 2 na iteração 1 está na posição *G + 1*;
  - O valor correspondente ao ponto de gauss 1 do elemento 1 na iteração 2 está na posição *N * G + 1*;
  - Em geral, o valor correspondente ao ponto de gauss g do elemento *n* na iteração *it* está localizado na posição *(it-1) * N * G + (n-1) * G + g*.

No total este ficheiro deve ter *IT * N * G* elementos; como cada elemento tem 8 *bytes*, o tamanho total do ficheiro deve ser exatamente *IT * N * G * 8 bytes*.

Por último, o ficheiro *RIGFUND.BIN* é necessário sempre que se considera a fundação como um único elemento, por oposição a simular o comportamento da estrutura completa, incluindo fundação, com o método dos elementos finitos. Esta técnica é utilizada quando se torna necessário reduzir o tamanho do problema. Neste caso são identificados os nós correspondentes ao assentamento da estrutura principal e os valores correspondentes às suas contribuições para a matriz de rigidez são calculados externamente e fornecidos ao *ParFludan* através do ficheiro *RIGFUND.BIN*. Este ficheiro é também binário, e tem as seguintes caraterísticas:

- É constituído por uma sequência de números reais, escritos em formato binário, cada um com 8 *bytes*;
- A orientação dos números segue o formato $I * 3 * J * 3$, sendo $I = J = N$ o número de nós de inserção e 3 o número assumido de graus de liberdade em cada nó. Assim, por exemplo:
  - O valor da rigidez correspondente à interação do nó de inserção 1, na dimensão *X*, com o próprio nó de inserção 1, na dimensão *X*, está na posição 1;
  - O valor da rigidez correspondente à interação do nó de inserção 1, na dimensão *X*, com o próprio nó de inserção 1, na dimensão *Y*, está na posição 2;
  - O valor da rigidez correspondente à interação do nó de inserção 1, na dimensão *X*, com o nó de inserção 2, na dimensão *X*, está na posição 4;
  - O valor da rigidez correspondente à interação do nó de inserção 1, na dimensão *Y*, com o próprio nó de inserção 1, na dimensão *X*, está na posição $N*3 + 1$;
  - O valor da rigidez correspondente à interação do nó de inserção 2, na dimensão *X*, com o nó de inserção 1, na dimensão *X*, está na posição $3*N*3 + 1$;
  - Em geral o valor da rigidez correspondente à interação do nó de inserção *I*, na dimensão *a*, com o nó de inserção *J*, na dimensão *b*, está na posição $(I-1)*3*N*3 + (a-1)*N*3 + (J-1)*3 + b$.

No total, este ficheiro deve ter $(N * 3)^2$ elementos; como cada elemento tem 8 *bytes*, o tamanho total do ficheiro deve ser exatamente $(N * 3)^2 * 8$ *bytes*. Note-se que o ficheiro contém todos os valores em duplicado, à exceção dos da diagonal, uma vez que as contribuições para a rigidez são simétricas.

O *ParFludan* admite ainda um formato alternativo para este ficheiro, em que além da sequência de valores reais de 8 *bytes* existem dois inteiros de 4 *bytes*, um no início do ficheiro e o outro no final; estes valores correspondem ao número total de *bytes* do ficheiro, a menos desses 4+4 *bytes*, e não são utilizados pelo programa[19].

Em todos estes casos, a consulta dos exemplos disponíveis na pasta *tests/testdata/* é útil como complemento a esta descrição; no caso particular do *TDIN.DAD*, é especialmente proveitoso consultar os exemplos para perceber como construir novos ficheiros de dados.

5.2.2 Ficheiros de saída

No final da execução o programa cria vários ficheiros de resultados, sendo os mais relevantes os ficheiros de resultados (com extensão *.res*) e o ficheiro de log (com extensão *.log*). Todos os ficheiros criados pelo programa são guardados na nova pasta de resultados criada especificamente para a execução em questão (ver secção 3.1.3).

Os principais ficheiros gerados pelo programa são os ficheiros de resultados. Por omissão são escritos ficheiros respeitantes a quatro grandezas: deslocamentos totais, tensões, extensões e danos.

---

[19] Este formato é permitido para facilitar a compatibilidade com os ficheiros escritos a partir de Fortran, que em certas condições acrescenta 4 bytes ao início e final do ficheiro.

Cada uma destas grandezas é escrita no final de cada fase (instantânea ou diferida), isto é, são escritos dois ficheiros por grandeza por cada iteração no tempo.

Por exemplo, suponha-se que determinado cenário considera ambas as fases instantânea e diferida do cálculo ao longo de 10 iterações temporais. Nesse caso, o programa irá produzir 80 ficheiros de resultados:

dis_0_0.res	ten_0_0.res	ext_0_0.res	dan_0_0.res
dis_0_1.res	ten_0_1.res	ext _0_1.res	dan_0_1.res
dis_1_0.res	ten_1_0.res	ext_1_0.res	dan_1_0.res
dis_1_1.res	ten_1_1.res	ext_1_1.res	dan_1_1.res
dis_2_0.res	ten_2_0.res	ext_2_0.res	dan_2_0.res
[...]	[...]	[...]	[...]
dis_9_0.res	ten_9_0.res	ext_9_0.res	dan_9_0.res
dis_9_0.res	ten_9_0.res	ext _9_0.res	dan_9_0.res

Em relação ao formato destes ficheiros, à semelhança dos ficheiros de entrada foi decidido manter o formato original, fazendo apenas a separação em vários ficheiros por oposição a ter um único contendo todos os resultados. Assim, todos têm um cabeçalho de duas linhas, e nas linhas seguintes o formato varia em função da grandeza em questão. No caso dos deslocamentos, cada linha tem o seguinte formato:

Nó dX dY dZ

em que *Nó* é o número do nó e *dK* é o valor do deslocamento na direção *K*. Cada ficheiro tem portanto *N+2* linhas, em que *N* é o número de nós na malha.

No caso das tensões e extensões é necessário escrever um tensor simétrico *S* para cada ponto de gauss de cada elemento. Cada linha tem o seguinte formato:

Elem PG S11 S22 S33 S23 S31 S12

em que *Elem* é o número do elemento, *PG* é o número do ponto de gauss, e *Sij* é a componente *(i,j)* do tensor *S*. Cada ficheiro deste tipo tem assim *(M*G + 2)* linhas, em que *M* é o número de elementos da malha e *G* o número de pontos de gauss por elemento.

Finalmente, no caso dos danos são escritas duas quantidades, correspondentes ao dano de tração e ao de compressão, por cada ponto de gauss de cada elemento. Assim, cada linha terá o formato:

Elem PG DT DC

em que *Elem* é o número do elemento, *PG* é o número do ponto de gauss, e *DT* e *DC* são os valores de dano à tração e à compressão, respetivamente. Como resultado, cada ficheiro deste tipo tem

também *(M*G + 2)* linhas, em que *M* é o número de elementos da malha e *G* o número de pontos de gauss por elemento.

Em todos os ficheiros de resultados os nós, elementos ou pontos de gauss são indexados em 0; por exemplo, ao nó 1 na malha inicial corresponde o nó 0 nos resultados, e ao elemento 173 na malha inicial corresponde o elemento 172 nos resultados. Na pasta *tests/test_results* estão disponíveis alguns exemplos de ficheiros de resultados, obtidos para os cenários de teste.

Em geral, os resultados gerados pela simulação não serão analisados no seu formato original, sendo antes sujeito a algum tipo de pós-processamento de acordo com os requisitos do utilizador. Uma operação frequente de pós-processamento dos resultados é a sua visualização, tal como mostra a figura 5.1. A definição e integração no programa principal de métodos de pós-processamento dos resultados pode ser feita recorrendo ao conceito de módulos, tal como descrito na secção 6.3.

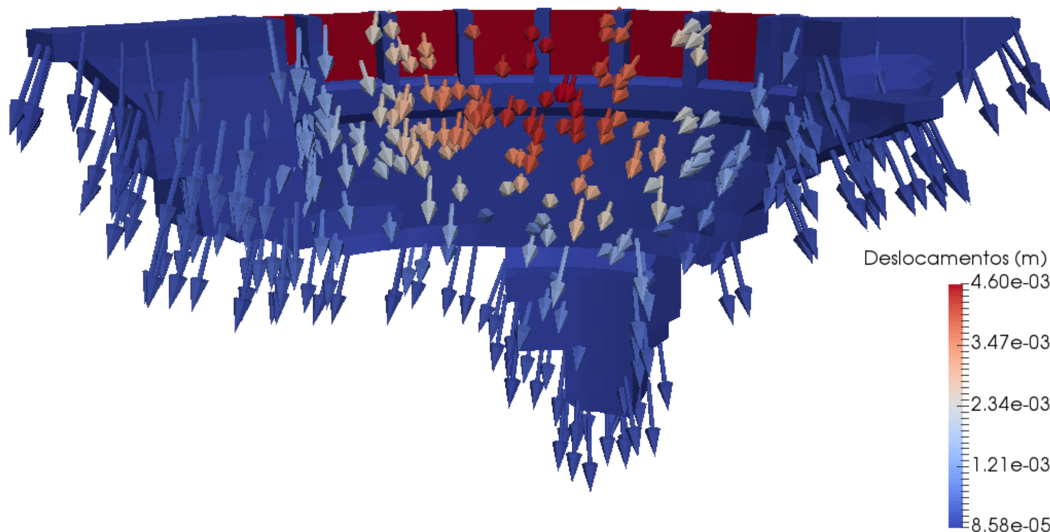


Figura 5.1 – Visualização dos deslocamentos obtidos pela simulação (malha de Peti)

Complementarmente aos ficheiros de resultados, o ficheiro de *log* contém um registo de informação importante sobre a execução, incluindo o tempo que o programa demorou a executar, a versão do programa utilizada, o utilizador que submeteu a execução, o comando utilizado para submeter o programa e a data, hora e local da execução. Este ficheiro serve dois propósitos: por um lado, identifica os restantes ficheiros presentes na pasta de forma a não haver ambiguidade sobre a que cenário dizem respeito ou em que programa foram gerados; por outro lado, contribui para a reprodutibilidade do ensaio ao dar informação sobre todas as condições importantes em que foi realizada.

5.3 Ficheiro de parametrizações

Para concluir a descrição dos ficheiros necessários para executar o programa falta mencionar o ficheiro de parametrizações incluído na pasta *src/* (*settings.c)*. Este ficheiro inclui algumas definições que permitem alternar rapidamente entre configurações frequentes do programa. É especialmente útil na fase de desenvolvimento, mas pode também ser conveniente para algum utilizador regular.

Este ficheiro funciona da seguinte forma: são definidas variáveis como macros de C, que são depois importadas para o programa principal; o programa ativa depois comportamentos diferentes em função das parametrizações definidas. A principal função destas variáveis é permitir modificar os dados de entrada rapidamente, sem necessidade de editar o ficheiro de dados propriamente dito.

Por exemplo, as variáveis *IGNORE_X*, em que *X* é uma das ações disponíveis, permitem ignorar o efeito dessas ações mesmo que elas estejam presentes no ficheiro de dados; por sua vez, a variável *OVERRIDE_OnlyViscoelasticBehaviour* permite desativar o cálculo de dano mesmo quando este faz parte do cenário definido[20]. Todas as parametrizações disponíveis são explicadas nos comentários do ficheiro *settings.c*.

Estas parametrizações são especialmente úteis na fase de desenvolvimento, para *debug* e experimentação. Para o funcionamento normal do programa, não é necessário interagir com este ficheiro; todas as variáveis devem ser mantidas com o valor 0, à exceção de *PRINTRESULTS* (caso contrário o programa não escreve os resultados) e *RESTRICT_FREE_EXP*, que devem ambas ser mantidas a 1.

5.4 Executar uma simulação

Nesta secção pretende-se mostrar como executar uma simulação, quer seja um dos exemplos incluídos com o programa, quer seja um novo cenário. O procedimento descrito aplica-se para simulações executadas tanto localmente (isto é, num portátil ou *desktop*) como no *cluster Medusa*.

Todas as submissões de simulações para execução são feitas através do módulo de *python run.py*, que abstrai o utilizador de detalhes como a localização dos ficheiros ou a plataforma onde a simulação é executada; o único argumento necessário é o nome do cenário a considerar.

A utilização deste módulo pode ser refinada através da passagem de parâmetros relevantes como argumentos opcionais, como por exemplo o número de processadores a utilizar ou a localização dos ficheiros de entrada (se for diferente da pré-definida).

Para todos os exemplos descritos nesta secção, assume-se que a estrutura de ficheiros do programa não foi alterada, i.e. que segue o esquema descrito na secção 3, e que os comandos indicados são executados a partir da diretoria *parfludan/src*.

---

[20] Note-se que a leitura do ficheiro de dados feita pelo programa vai ser diferente consoante este parâmetro esteja ativado ou não, o que pode originar erros caso o ficheiro de dados e o ficheiro de parametrizações não sejam consistentes.

5.4.1 Exemplo de execução de uma simulação

Admita-se que a diretoria *data* contém a subdiretoria *baixosabor*, que por sua vez contém todos os ficheiros necessários para especificar um cenário de simulação (tal como descrito na secção 5.2.1).

Nesse caso, para executar a simulação do cenário baixosabor, basta ao utilizador usar o comando

$ python run.py baixosabor

Logo que a simulação terminar, os resultados estarão disponíveis na pasta *results/*, tal como descrito na secção 5.2.2.

5.4.2 Parâmetros

O módulo *run.py* admite receber como argumento um conjunto de parâmetros opcionais (*flags*) que permitem adaptar a execução do programa às necessidades do utilizador. A tabela 5.3 apresenta alguns dos mais relevantes.

Tabela 5.3 – Descrição de alguns parâmetros opcionais

Flag	Descrição	Exemplo de utilização
-np	Definir número de processadores	*python run.py baixosabor -np 16*
-v	Ativar modo verboso	*python run.py baixosabor -v*
-T	Ativar configurações de teste	*python run.py baixosabor -T*
-flu	Executar *Fludan-RAS*	*python run.py baixosabor -flu*

O parâmetro *-np* permite indicar em quantos processadores se pretende executar a simulação, aproveitando o paralelismo do programa. O parâmetro *-v* permite ativar o modo verboso, que vai escrevendo o estado do programa em diferentes momentos para a linha de comando. O parâmetro *-T* permite ativar as configurações de teste, o que pode ser útil se o utilizador pretender executar apenas um dos exemplos de teste. Finalmente, a opção *-flu* permite chamar o programa *Fludan-RAS*, em vez do *ParFludan*, para executar a simulação, o que pode ser útil para compação de resultados.

Outras *flags* permitem ajustar o módulo *run.py* para lidar com estruturas de ficheiros diferentes das que estão pré-definidas. Por exemplo, se se pretender escrever os resultados para a pasta *parfludan/Resultados_das_corridas*, em vez da pré-definida *parfludan/results*, então basta passar a *flag --resultsfolder* com o nome da nova pasta:

$ python run.py baixosabor --resultsfolder Resultados_das_corridas

Neste caso, os resultados serão escritos nesta pasta, em vez da pasta pré-definida, para esta simulação apenas (i.e., em execuções futuras, a pasta por omissão continuará a ser *parfludan/results*). Se a estrutura de ficheiros for mantida no estado original, nunca será necessário usar estes parâmetros.

Em geral, todas as *flags* podem ser usadas em simultâneo, embora em alguns casos tal não faça sentido e algumas das opções sejam ignoradas (por exemplo, ao tentar combinar *-np* com *-flu*). A ordem em que as *flags* são inseridas não é importante. A documentação do módulo *run.py* contém uma explicação detalhada de todos os parâmetros disponíveis. Para a consultar basta executar o comando *python run.py --help.*

5.4.3 Exemplos de utilização

Na prática, a maioria dos utilizadores terá interesse em combinar diferentes parâmetros ao executar uma simulação. Esta secção apresenta alguns exemplos de utilizações mais complexas do programa e como as executar.

1. Executar o cenário *baixosabor* em 20 processadores, em modo verboso:

   $ python run.py baixosabor -np 20 -v

2. Executar o cenário *baixosabor* em 20 processadores, em modo verboso, e escrever os resultados na pasta *parfludan/novos_resultados*:

   $ python run.py baixosabor -np 20 -v --resultsfolder novos_resultados

3. Executar o cenário de teste *vigapp* em 20 processadores:

   $ python run.py vigapp -np 20 -T

4. Executar o cenário *novabarragem*, definido em *parfludan/cenarios_novos/*, cujo ficheiro de entrada tem o nome *TDIN_novabarragem.dad*, em 20 processadores:

   $ python run.py novabarragem -np 20 --inputfile TDIN_novabarragem.dad --datafolder cenarios_novos

5. Executar o cenário *novabarragem*, definido em *parfludan/cenarios_novos/*, cujo ficheiro de entrada tem o nome *TDIN_novabarragem.dad*, com o programa *fludan2009*:

   $ python run.py novabarragem --inputfile TDIN_novabarragem.dad --datafolder cenarios_novos -flu --srcseqfolder fludan2009

5.5 Executar os testes

No capítulo 4 foram apresentados os testes introduzidos para orientar o desenvolvimento do programa e verificar a sua funcionalidade. Mesmo não sendo o seu principal objetivo, pode ser útil ao utilizador executar os testes disponíveis, por exemplo para confirmar que a instalação inicial foi bem sucedida ou para verificar a integridade do programa após introduzir modificações no código ou na estrutura de ficheiros. Esta secção descreve como executar estes testes e analisar os seus resultados.

Estão definidos três tipos de testes: testes unitários aos módulos *python*, testes unitários aos módulos *.c*, e testes de integração. O módulo *python runtests.py* permite executar todos os testes disponíveis em sequência. Para o utilizar basta executar o comando:

$ python runtests.py

Os resultados são escritos para a linha de comando e dividem-se em três partes, uma para cada tipo de testes. Estes resultados incluem mensagens autoexplicativas de sucesso ou insucesso, pelo que a inspeção da saída do comando basta para compreender se o programa passa todos os testes.

Por omissão, o módulo *runtests.py* executa todos os testes de integração disponíveis, usando 1 processador. Em alternativa, estão disponíveis algumas *flags* que permitem modificar este comportamento, nomeadamente:

1. *-np*: definir o número de processadores com que os testes de integração serão executados. Exemplo: *python runtests.py –np 2*
2. *-tc*: definir apenas um subconjunto dos testes de integração para execução.
   Exemplo: *python runtests.py -tc vigapp vigaph vigavt*
3. *-td*: definir a pasta que contém os diferentes cenários de teste (incluídos por omissão em *parfludan/tests/testdata*).
   Exemplo: *python runtests.py -td tests/testdata-novo.*

6 | Customização do programa

O *ParFludan* permite ao utilizador alguma flexibilidade para alargar a funcionalidade do programa e o adaptar às suas necessidades específicas. Neste capítulo é identificado um conjunto de áreas em que é provável que o utilizador queira intervir, de forma a personalizar o funcionamento do programa, e descrito de que forma pode levar a cabo essa intervenção.

6.1 Como adicionar um cenário

Porventura a primeira necessidade de um utilizador do *ParFludan* será definir o seu próprio cenário de simulação. Para o fazer basta ter em atenção que:

1. Os cenários para execução são definidos na pasta *data/* (secção 3.1.4);
2. Para definir um cenário é preciso criar os ficheiros de dados relevantes (secção 5.2.1).

A partir do momento em que exista na pasta *data/* uma subpasta com o nome do novo cenário, e essa pasta contenha o ficheiro *TDIN.DAD* e os ficheiros auxiliares relativos às ações relevantes, o cenário está pronto a ser executado pelo programa.

6.2 Como adicionar um teste

É também provável que a certa altura seja necessário adicionar testes ao programa, tanto unitários como de integração. Os testes unitários terão de ser explicitamente programados pelo utilizador, enquanto que os testes de integração poderão ser simplesmente cenários que já não são interessantes para execução e cujos resultados estão bastante bem estabelecidos.

Para adicionar um teste unitário é preciso, em primeiro lugar, criá-lo seguindo as instruções da secção 4.1.1. No caso de se tratar de um teste a um módulo de *python*, deve ser acrescentado à pasta *src/tests*; caso seja um teste a um módulo *.c*, são necessários dois passos:

1. Acrescentar o ficheiro de teste à pasta *src/*;
2. Editar o ficheiro *runCtests.sh* de forma a incluir este novo teste no conjunto de testes já definido (seguindo a lógica implícita nesse ficheiro).

Para adicionar testes de integração, são necessários três passos:

1. Copiar os conteúdos da pasta onde o cenário está definido, contendo o *TDIN.DAD* e outros ficheiros de dados, para a pasta *tests/testdata/nome_do_novo_cenário*; isto garante que o cenário fica disponível para execução pelo *ParFludan* e pelo *script* de testes *runtests.py*;
2. Editar o ficheiro *run_fludan.sh* (da versão relevante), adicionando o nome do novo cenário ao conjunto de cenários processados pelo programa (seguindo a lógica implícita nesse ficheiro); isto garante que o cenário fica disponível para esta versão do *Fludan-RAS*, e só é relevante caso o utilizador deseje comparar os resultados com os obtidos com a versão sequencial

original (por oposição a comparar apenas os resultados do programa entre si e/ou com os resultados de referência);

3. Copiar os ficheiros de resultados da última execução do cenário, assumindo que estão corretos, para a pasta *tests/teststandards/nome_do_novo_cenário*; isto garante que o programa dispõe do conjunto de resultados base com os quais pode comparar os resultados obtidos em cada execução dos testes, e só é relevante caso o utilizador deseje fazer essa comparação (por oposição a comparar apenas os resultados do programa entre si e/ou com os do *Fludan-RAS*).

6.3 Como adicionar um módulo

Uma das vantagens da organização modular do programa é permitir que a sua funcionalidade seja alargada através da criação e inclusão de novos módulos, minimizando a necessidade de intervir no código já existente. Tal como na secção 3.2, distinguimos entre módulos do programa principal e módulos auxiliares.

Para adicionar um módulo do programa principal, o utilizador deve criar três novos ficheiros:

- O ficheiro *novo_modulo.h*, na pasta *include/*, onde define a interface;
- O ficheiro *novo_modulo.c*, na pasta *src/*, onde define a implementação;
- O ficheiro *test_novo_modulo.c*, na pasta *src/*, onde define os testes unitários ao novo módulo.

Neste ponto os conteúdos do módulo estarão prontos a ser usados no programa principal, bastando para isso acrescentar o novo módulo à lista de *includes* no ficheiro *header.h* e editar o código fonte para fazer uso dos novos conteúdos como desejado.

Como o processo de compilação não está totalmente automatizado, é depois necessário incluir esse módulo na compilação do programa; para tal, é preciso editar o ficheiro *processing.sh* e incluir o novo módulo seguindo a lógica implícita no ficheiro. Para incluir os testes ao novo módulo na bateria de testes definida, basta seguir as instruções da secção 6.2.

Para adicionar um módulo auxiliar, basta criar o novo módulo e incluí-lo na pasta *src/*; ficará automaticamente pronto para ser usado pelo programa. No caso de se pretender acrescentar o novo *script* à cadeia de execução normal do programa, o comando para o invocar deve ser incluído no *script processing.sh*. Esta abordagem pode ser especialmente útil no caso de se pretender definir módulos de pós-processamento dos resultados.

7 | Conclusão

Este relatório descreve o programa *ParFludan*, um programa paralelo para a simulação do comportamento de estruturas homgéneas de grandes dimensões através do método dos elementos finitos, criado com base no *Fludan-RAS*, um programa sequencial para o mesmo efeito desenvolvido no LNEC. A motivação para o desenvolvimento do *ParFludan* provém sobretudo das limitações exibidas pelo *Fludan-RAS*, em particular em termos do tempo de execução das simulações e ao nível do processamento de malhas de grandes dimensões.

Para dar resposta a estes desafios, foi adotada uma estratégia baseada na paralelização do programa original, tendo em vista a utilização do *cluster Medusa*. Este processo exigiu a escrita de um novo programa, com as estruturas de dados adequadas ao modelo de computação paralela distribuída, cujo processo de desenvolvimento envolveu, além da paralelização, a modularização e documentação do código.

O *ParFludan* foi submetido a um processo de avaliação de qualidade, destinado a verificar o cumprimento tanto dos requisitos funcionais como dos não-funcionais. A avaliação funcional, isto é, a verificação da exatidão dos resultados obtidos pelo *ParFludan*, foi desenhada com base em baterias de testes unitários, para validar o comportamento de cada componente isoladamente, e em testes de integração, para validar o comportamento das várias componentes em conjunto. A validação dos resultados obtidos teve por base a comparação com resultados analíticos ou experimentais previamente conhecidos e foi concluída com sucesso.

No caso dos requisitos não-funcionais, isto é, os requisitos relacionados com o funcionamento do sistema (tais como o tempo de execução ou a memória ocupada, por exemplo), a avaliação realizada permitiu concluir que o novo programa os cumpre adequadamente, verificando-se uma redução no tempo de cálculo da ordem das semanas para as dezenas de horas, bem como o aumento da complexidade dos problemas que o programa é capaz de tratar corretamente, tendo sido obtidos resultados para malhas maiores do que as que o programa original admitia (por exemplo no caso da barragem de Peti, uma malha com cerca de 13 mil nós).

Os resultados da avaliação mostram claramente que a estratégia de paralelização adotada teve sucesso e permitiu eliminar as limitações que estiveram na origem do desenvolvimento do *ParFludan,* tornando assim possível a análise de estruturas mais complexas em menos tempo. Parelelamente, a simplificação da interface do programa e a sua generalização para diferentes ambientes de execução permite que um utilizador consiga mais facilmente executar o programa, independentemente da plataforma computacional que esteja a usar. Finalmente, a estratégia de modularização e documentação do código resultou num programa que poderá mais facilmente ser compreendido e utilizado, e cuja funcionalidade pode agora ser alargada para se adequar às necessidades específicas de cada utilizador.

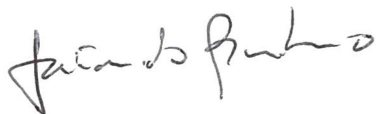
Agradecimentos

Os autores agradecem a colaboração do Investigador Auxiliar José Piteira Gomes, do Núcleo de Observação do Departamento de Barragens de Betão, pela disponibilização do programa original e da sua documentação, pelo apoio na sua compreensão e exploração e pelas contribuições para o desenvolvimento e validação do *ParFludan*.

Lisboa, LNEC, junho de 2016

VISTO

O Chefe do Núcleo de Tecnologias da Informação em Engenharia Civil



José Barateiro

AUTORIA

João Coelho

Bolseiro de iniciação à investigação científica

António Inês Silva

Investigador principal

Referências Bibliográficas

BALAY, Satish; *et al*, 2016 – **PETSc Web Page.** URL: http://www.mcs.anl.gov/petsc.

BALAY, Satish; GROPP, William; MCINNES, Lois; SMITH, Barry, 1997 – **Efficient Management of Parallelism in Object Oriented Numerical Software Libraries**. Modern Software Tools in Efficient Computing, pp. 163-202. ISBN: 978-1-4612-1986-6.

COELHO, João; SILVA, António; PITEIRA GOMES, José, 2014 – **Parallelization of a tridimensional finite element program for structural behaviour analysis**. Springer Lecture Notes in Computer Science, vol. 8805, pp. 36-47. Porto, Portugal.

COELHO, João; SILVA, António, 2013 – **Computação paralela no LNEC – Guia introdutório e estudo de caso**. LNEC - Proc. 1302/044. Relatório 237/2013 – CD/NTIEC.

GABRIEL, Edgar; *et al*, 2004 – **Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation**. Proceedings, 11th European PVM/MPI Users' Group Meeting. Budapeste, Hungria.

PITEIRA GOMES, José, 2008 – **Modelação do comportamento estrutural de barragens de betão sujeitas a reacções expansivas**. LNEC, Lisboa. ISBN: 978-972-49-2143-3.

Anexos

ANEXO I
Exemplo do ficheiro de configurações *settings.c*

```
/* Settings file */

// Force vector components - set to 1 to ignore, 0 to keep default

#define IGNORE_npp 0

#define IGNORE_nph 0

#define IGNORE_nvt 0

#define IGNORE_ndefi 0


// Delayed response - set to 1 to ignore, 0 to keep default

#define IGNORE_DelayedResponse 0


// Timesteps - N to use N timesteps, 0 to keep default,

#define OVERRIDE_Timesteps 1


// PrintTests - set to 1 to print test statements

#define PRINTTESTS 0


// PrintResults - set to 1 to print results

#define PRINTRESULTS 1


// Additional OVERRIDE options

// if set to 1, overrides default behaviour as set from input data


#define OVERRIDE_DamAlwaysFull 0


#define OVERRIDE_OnlyViscoelasticBehaviour 0 // set to 1 for all testcases in tests/


#define RESTRICT_FREE_EXP 1 // if set to 0, doesn't restrict expansion of finite elements. Set to 1 to restrict expansions as a consequence of previous tensions (normal usage)
```

ANEXO II
Exemplo simplificado de ficheiro de dados *TDIN.DAD*

```
BARRAGEM DE PETI  -  MALHA DA BARRAGEM
ELEMENTOS FINITOS TRIDIMENSIONAIS
TIPO CUBO ISOPARAMETRICOS DO 2§ GRAU
13135  2366  0  0  1   1 !NPONT NELEM NAPRI NAPEL NAFUN  IASC
4933 1326 14    60   0.10  100000   !Not Nelt NPGt Mniter Stest TDIV
Exponencial
2 3  0.18     !NGRUP  NPGAUSS COEFHIST[=0. a .18]
1945 01 01  !ANO0 MES0 DIA0  - Data origem dos tempos (Presa do betao)
1945 10 11  !ANOI MESI DIAI  - Data de inicio das solicitacoes
2011 12 08  !ANOF MESF DIAF  - Data do final do periodo em analise
1720 1 1  !NINT ICNL ICVISCO     - Numero de intervalos adotados na discr.
do tempo
i ! Grupo 1
20.0E6   0.2            !E0  P0
0.95    2000.    1.     !
-26000000. -28000000. -30000000.    !
-.002   -2626900.6  -.0021 -2400000.    !
-.006 -12000000. !(b=0.)
27e6 5.6 0.46 0.05 0.118 ! Emaj FI1 XM BET XN
24. 0. 0. -1. !
1.10e-5  1.10e-5   1.10e-5  !
i ! Grupo 2
210.0E6   0.3            !E0  P0
0.95    2000.    1. !
-26000000. -28000000. -30000000. !
-.002   -2626900.6  -.0021 -2400000. !
-.006 -12000000. !(b=0.)
27e6 5.6 0.46 0.05 0.118 ! Emaj FI1 XM BET XN
24. 0. 0. -1. !
0.0   0.0   0.0    ! 1.10e-5  1.10e-5   1.10e-5  !
COORDENADAS DOS PONTOS NODAIS
41.571      24.880      713.000     1
41.571      25.380      713.000     2
41.571      24.880      712.500     3
40.971      24.880      713.000     4
[...]
-40.383     14.159      711.900     13134
-40.383     14.159      710.800     13135
INCIDENCIAS
7708  7579  7539  7667  7638  7351  7335  7620  7635  7556  7602  7684  7505
      7340  7488  7625  7676  7465  7437  7645  1     1
```

```
7621  7388  7351  7579  7755  7672  7638  7708  7512  7367  7465  7601  7714
      7652  7676  7729  7697  7536  7505  7635  1     2
[...]
1701  1801  1960  1923  1603  1697  1879  1838  1741  1883  1944  1815  1635
      1792  1859  1725  1646  1748  1924  1881  1     2365
1868  1701  1603  1790  1857  1673  1586  1775  1782  1646  1702  1827  1769
      1627  1679  1817  1863  1690  1593  1780  1     2366
APOIOS RIGIDOS
1     1     1     1
25    1     1     1
26    1     1     1
50    1     1     1
[...]
13102 1     1     1
13104 1     1     1
13106 1     1     1
SOLICITACOES
1  1  0  1   0  0  0  0
PESO PROPRIO --->    1
1.0
PRESSAO HIDROSTATICA -->    1         1945 10 11
10.000
674.00
0
EXPANSOES ----->    1
1
2366
PESO PROPRIO --->    2
1.0
PRESSAO HIDROSTATICA -->    2         1945 10 25
10.000
688.00
0
EXPANSOES ----->    2
1
2366
[...]
PESO PROPRIO ---> 1719
1.0
PRESSAO HIDROSTATICA --> 1719         2011 11  8
10.000
```

```
712.11
0
EXPANSOES ----->    1719
1
2366
PESO PROPRIO ---> 1720
1.0
PRESSAO HIDROSTATICA --> 1720         2011 11 22
10.000
712.11
0
EXPANSOES ----->    1720
1
2366
```

ANEXO III
Exemplo simplificado do ficheiro de dados *FACESM.DAD*

```
   704
8     1
11    6
12    6
19    1
20    6
27    6
28    6
[...]
536   6
539   1
542   6
545   6
[...]
2357  6
2358  6
2359  6
2361  3
2363  4
2364  6
```

ANEXO IV
Exemplo simplificado do ficheiro de dados *VARTERM.DAD*

```
1945 10 11      674.00
     1       0.000
     2       0.000
     3       0.000
     4       0.000
 [...]       [...]
 13133       0.000
 13134       0.000
 13135       0.000
 1945 10 25      688.00
     1       0.740
     2       0.740
     3       0.740
     4       0.740
 [...]       [...]
13133        0.550
13134        0.530
13135        0.570
 1945 11  8      698.54
     1       -0.550
     2       -0.590
     3       -0.350
     4       -0.590
 [...]       [...]
13133       -0.590
13134       -0.470
13135       -0.130
[...]       [...]
[...]       [...]
[...]       [...]
```

```
[...]        [...]
[...]        [...]
 2011 11 22      712.11
     1       0.320
     2       0.320
     3       0.320
     4       0.320
 [...]       [...]
13133        0.530
13134        0.530
13135        0.490
 2011 12  6      712.11
     1       0.740
     2       0.740
     3       0.740
     4       0.740
 [...]       [...]
13133        0.550
13134        0.530
13135        0.570
```

Divisão de Divulgação Científica e Técnica - LNEC