



LABORATÓRIO NACIONAL
DE ENGENHARIA CIVIL

USO EFICIENTE DE MEMÓRIA EM COMPUTAÇÃO CIENTÍFICA

Núcleo de Tecnologias da Informação em Engenharia Civil

Lisboa • junho de 2016

I&D TECNOLOGIAS DA INFORMAÇÃO

RELATÓRIO 199/2016 – NTIEC

Título

USO EFICIENTE DE MEMÓRIA EM COMPUTAÇÃO CIENTÍFICA

Autoria

NÚCLEO DE TECNOLOGIAS DA INFORMAÇÃO EM ENGENHARIA CIVIL

João Rico

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 199/2016

Proc. 0109/112/20179

USO EFICIENTE DE MEMÓRIA EM COMPUTAÇÃO CIENTÍFICA

Resumo

A otimização de programas de computação científica é um problema extremamente relevante em engenharia pois uma otimização eficaz pode trazer ganhos de algumas ordens de grandeza em diversas dimensões: tempo de execução, memória utilizada, custo, precisão numérica, dimensão dos dados de entrada, tempo de programador, etc. Este relatório discute a otimização de memória e, em particular, pretende fornecer linhas de orientação para a otimização eficaz de um programa de computação científica em termos de memória. Descreve-se brevemente um modelo simples de um computador que permite compreender os conceitos mais relevantes da memória de um computador, e descrevem-se várias estratégias de otimização. Estas estratégias são discutidas e exemplificadas, e compiladas numa *checklist* com vista a ser percorrida e analisada aquando da otimização de memória de um programa de computação científica. É discutido um caso de estudo em que foi feita a análise dos objetivos da otimização e das características do programa. Feita esta análise foram escolhidas três estratégias da *checklist* que permitiram, com pouco esforço, resolver completamente o problema original.

Palavras-chave: Memória / Otimização de programas / Computação científica

THE EFFICIENT USE OF MEMORY IN SCIENTIFIC COMPUTATION

Abstract

The optimization of scientific computing programs is an extremely important problem in engineering since an effective optimization can bring gains of a few orders of magnitude in several dimensions: runtime, memory usage, cost, numerical precision, size of the input data, developer time, etc. This report discusses memory optimization and, in particular, aims to provide guidelines for the effective optimization of a scientific computing program in terms of memory. It describes briefly a simple model of a computer that allows the understanding of the most important concepts of computer memory, and describes various optimization strategies. These strategies are discussed and exemplified, and compiled in a *checklist* in order to be runned through and analyzed when attempting the memory optimization of a scientific computing program. A case study is discussed in which the analysis of the optimization objectives and program characteristics was carried out. Having completed this analysis three of strategies of the *checklist* were chosen that allowed, with little effort, to completely solve the original problem.

Keywords: Memory / Program optimization / Scientific computing

Índice

1	Introdução	1
2	Arquitetura e tipos de memória de um computador	2
2.1	Um modelo simples de um computador	2
2.2	Tamanho de palavra	3
2.3	Diferenças entre sistemas de 32 e 64 bits	4
3	Estratégias para uso eficiente da memória	6
3.1	Estratégias para uso eficiente de memória a nível do sistema	7
3.1.1	A transição de 32 bits para 64 bits	7
3.1.2	Adicionar mais memória	10
3.2	Estratégias para uso eficiente da memória a nível do código	10
3.2.1	Uso eficaz da memória dinâmica	10
3.2.2	Algoritmos eficientes	11
3.2.3	Estruturas de dados compactas	13
3.2.4	Entradas de dados adequadas	13
3.2.5	Opções de compilador	14
3.2.6	Debuggers de memória	14
3.2.7	Paralelizar	14
3.3	Uma <i>checklist</i> para otimizações de memória	16
4	Caso de estudo: cálculo estrutural viscoelástico de barragens de betão usando o método dos elementos finitos	17
4.1	Descrição do programa	17
4.2	Aplicação da <i>checklist</i>	18
5	Conclusões	21
	Agradecimentos	21
	Referências Bibliográficas	25
	ANEXO Lista de <i>debuggers</i> de memória	27

Índice de figuras

Figura 2.1 – Esquema simples da arquitectura de um computador. A representação inclui o processador (CPU) e os seus registos, assim como a memória principal e os vários barramentos (de memória, de sistema e de entrada e saída). Fonte: Figura 1.4 de (Bryant; O'Hallaron, 2010).....	2
Figura 2.2 – Esquema dos registos x86-64. Destacado a vermelho: esquema dos registos x86. Fonte: adaptado da Figura 3.35 de (Bryant; O'Hallaron, 2010).....	4
Figura 3.1 – Painel de Informação de Sistema do Windows em que se pode verificar o tipo de processador do sistema.....	7
Figura 3.2 – Painel de Controlo do Windows onde se pode ver que versão do Windows está instalada	8
Figura 3.3 – Exemplo de instaladores Windows de 32 e 64 bits, na página de download do Git	9
Figura 3.4 – Código Fortran90 que declara e aloca uma matriz dinâmica	11
Figura 3.5 – Evolução do fator de <i>speed-up</i> para resolução de sistemas lineares esparsos devido a melhorias de algoritmos e de hardware. Fonte: (SIAM Working Group on CSE Education, 2001)	12
Figura 3.6 – Código Fortran que declara duas estruturas de dados para representar uma data e hora.....	13
Figura 3.7 – Ilustração da lei de Amdahl. Fator de <i>speed-up</i> em função de número de processadores, para diferentes percentagens da parte paralelizável do programa original. Fonte: https://en.wikipedia.org/wiki/Amdahl%27s_law (acedido em 24-06-2016)	15
Figura 3.8 – Ilustração da lei de Gustafson. A secção azul-claro do diagrama corresponde à parte paralelizável do programa que é a que beneficia do aumento do número de processadores. Fonte:	15
Figura 4.1 – Simulação da barragem de Alqueva. Fonte: (Castilho, 2013)	17
Figura 4.2 – Elemento hexahedral de 20 nós. Fonte: (Griffiths, Smith, Margetts; 2013)	18
Figura 4.3 – Matriz esparsa, simétrica e em banda	18
Figura 4.4 – Painel esquerdo: Fotografia aérea de barragem do Alqueva. Painel direito: malha de elementos finitos da barragem do Alqueva. Fonte: (Castilho, 2013).....	19

Índice de tabelas

Tabela 3.1 – Estratégias para uso eficiente de memória, sem alterar o código	16
Tabela 3.2 – Estratégias para uso eficiente de memória, alterando o código	16
Tabela 5.1 – Estratégias para uso eficiente de memória, sem alterar o código	21
Tabela 5.2 – Estratégias para uso eficiente de memória, alterando o código	22

1 | Introdução

A otimização de programas de computador pode ser levada a cabo em diversas dimensões: velocidade e tempo de execução, custo económico ou energético, testes e fiabilidade, latência, usabilidade, alterabilidade, qualidade de documentação, controlo de versões, etc.

Neste relatório, pretende-se discutir alguns dos problemas mais comuns dos programas de computação científica no que diz respeito à utilização da memória principal. Sugerem-se soluções para estes problemas e, em particular, discute-se como detetar e corrigir erros relacionados com a memória e também como otimizar um programa em termos de requisitos de memória. A discussão é dedicada essencialmente à memória principal de um computador, deixando de parte outros tipos de memória como, por exemplo, as caches do processador.

No capítulo 2, discutem-se os diferentes tipos de memória de um computador e explicam-se as diferenças entre sistemas de 32 e 64 bits, incluindo a mais importante: os sistemas de 32 bits estão limitados a um máximo de 4 GB de memória. O capítulo 3 descreve os problemas e restrições de memória mais comuns e mais importantes em computação científica, bem como diversas estratégias para a sua resolução. O capítulo conclui com uma *checklist* de estratégias a usar a fim de otimizar um programa em termos de memória - quer um já existente quer um novo, escrito de raiz. Descreve-se um caso de estudo no capítulo 4: um programa em Fortran de elementos finitos para o cálculo elástico linear da estrutura de uma barragem de betão. As conclusões deste relatório apresentam-se no capítulo 5.

2 | Arquitetura e tipos de memória de um computador

Neste capítulo descreve-se, brevemente, um modelo simples de um computador que permite perceber quais são os diferentes tipos de memória de um computador e quais as principais diferenças entre sistemas e programas de 32 e 64 bits.

A diferença mais importante é o facto de um sistema ou programa de 32 bits apenas conseguir usar 4 GB de memória, ao contrário dos de 64 bits cujo limite teórico é de 2^{64} bits de endereços o que corresponde a 16 milhões de TB de memória.

2.1 Um modelo simples de um computador

A maioria dos computadores atuais tem uma arquitetura muito semelhante: *motherboard*, processador, placa gráfica, memória principal e secundária, barramentos de dados e periféricos (como um teclado ou um monitor). Na Figura 2.1 mostra-se um esquema simplificado da arquitetura de um computador. De forma simplificada, o que um computador faz consiste em transportar dados e instruções da memória principal através dos barramentos para os registos do processador onde realiza algumas operações e, posteriormente, transportar os resultados dessas operações de volta à memória.

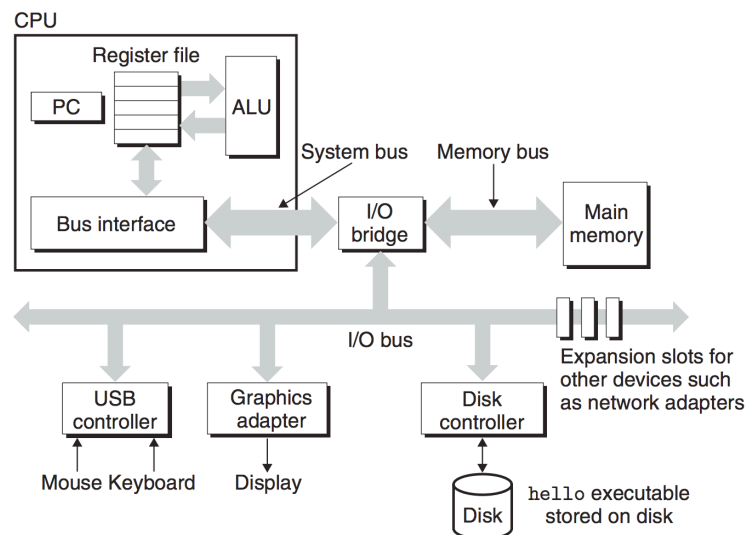


Figura 2.1 – Esquema simples da arquitectura de um computador. A representação inclui o processador (CPU) e os seus registos, assim como a memória principal e os vários barramentos (de memória, de sistema e de entrada e saída). Fonte: Figura 1.4 de (Bryant; O’Hallaron, 2010)

Os registos de um processador são o tipo mais rápido de memória de um computador e são responsáveis por guardar os dados que o CPU vai processar. Estes dados (uma sequência de bits) representam diretamente o tipo de dados que o processador manipula e, em particular, é a um registo que o processador irá buscar um endereço de memória de que necessite. A memória principal é uma

memória mais lenta, mas maior, e que corresponde à memória que o processador pode endereçar diretamente. Nela são armazenados os dados de que o processador necessita como, por exemplo, as instruções ou as variáveis de um programa. A memória principal faz a ponte entre o processador e as memórias secundárias como discos rígidos ou CDs e DVDs. Hoje em dia, a memória *RAM* (do inglês, *random access memory*) é o tipo de memória mais usada como memória principal. Um barramento é um dispositivo de transferência de dados entre as várias componentes internas de um computador (por exemplo, o CPU e a memória principal) ou entre o computador e dispositivos externos (por exemplo, um cabo USB, *Universal Serial Bus*). Os processadores possuem ainda outro tipo de memória denominadas de cache e que se localizam entre os registos e a memória principal. As caches, sendo mais rápidas e mais pequenas do que a memória principal mas maiores e quase tão rápidas como a memória dos registos, servem para diminuir o tempo de uma nova leitura ou escrita de dados da memória principal.

2.2 Tamanho de palavra

O **tamanho de palavra** é o número de bits das sequências de bits que um registo geral de um processador consegue guardar e é uma das características mais importantes de um sistema informático (hoje em dia, tipicamente, o tamanho de palavra é 32 ou 64 bits). Este valor geralmente coincide com a largura do barramento do sistema. Se uma sequência de bits num registo representar um endereço de memória, o número de endereços possíveis de endereçar com esta sequência é dramaticamente diferente, caso o tamanho da sequência seja 32 ou 64 bits. No primeiro caso, o número de endereços é 2^{32} , e cada endereço correspondendo a um *byte*, perfaz um total de 4 GB de memória endereçável, enquanto que com 64 bits o número de endereços é 2^{64} , o que corresponde a um total de 16 EB¹, cerca de 10^9 mais que os sistemas típicos hoje em dia. Na prática, o sistema *x86-64* ainda implementa apenas endereços de memória de 48 bits, o que perfaz um total de 256 TB de memória endereçável². Quanto aos sistemas de 32 bits, na prática, cada processo não chega sequer a ter acesso a 4 GB de memória porque o sistema operativo e outros processos também usam parte da memória. Por esta razão, por exemplo, em Windows tipicamente o limite será mais próximo dos 3.5 GB do que dos 4 GB.

¹ EB é o símbolo de exabyte, que corresponde a 10^{18} bytes ou um milhão de terabytes.

² Por serem os sistemas mais comuns hoje em dia, a discussão restringir-se-á aos sistemas *x86* vs *x86-64*, de 32 e 64 bits, respectivamente.

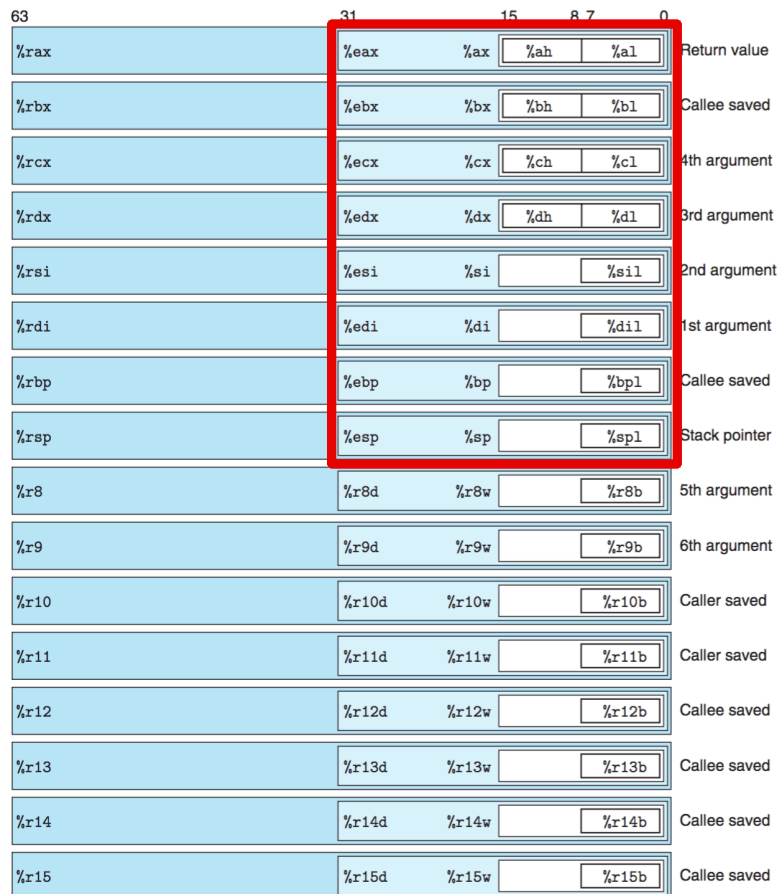


Figura 2.2 – Esquema dos registos x86-64. Destacado a vermelho: esquema dos registos x86. Fonte: adaptado da Figura 3.35 de (Bryant; O'Hallaron, 2010)

Na Figura 2.2 apresenta-se um esquema dos registos x86-64. Há 16 registos de 64 bits, enquanto que a vermelho estão marcados os 8 registos de 32 bits dos processadores x86. O sistema de 64 bits tem o dobro do número de registos, e cada registo é duas vezes maior. O sistema x86-64 inclui o sistema x86 como subsistema, sendo retrocompatível com programas de 32 bits e até 16 de bits.

2.3 Diferenças entre sistemas de 32 e 64 bits

Como já referido, a maior diferença entre os sistemas de 32 e 64 bits é o facto de os sistemas de 32 bits terem um limite máximo de 4 GB de memória, enquanto que o limite dos sistemas de 64 bits é várias ordens de grandeza superior. Para além desta diferença crucial, existem outras que, apesar de comparativamente menores, interessa mencionar.

Enquanto que um programa de 32 bits (e até um de 16 bits) pode ser executado num sistema de 64 bits, o contrário não é verdade: um programa de 64 bits não é executável num sistema de 32 bits.

Em termos de desempenho, apesar de na maioria dos casos ser preferível usar uma versão de 64 bits, existem algumas desvantagens. Em 64 bits, os programas ocupam mais espaço, já que os ponteiros duplicam de tamanho, assim como o tamanho padrão de alguns tipos de dados (por exemplo, em C, os *long int* passam de 32 a 64 bits na maioria dos compiladores para *linux*). No

entanto, em geral, os programas de 64 bits podem ser mais otimizados. Como já referido (e ilustrado Figura 2.2), os registos duplicam em tamanho e em número na passagem dos 32 para os 64 bits. O dobro do tamanho nos registos quer dizer a aritmética não está limitada a 32 bits (operações como *add*, *subtract*, *and*, etc), e que as instruções SIMD (em inglês, *single instruction, multiple data*) podem operar em mais dados por ciclo de relógio. O dobro do número de registos possibilita menos transportes de dados de e para a memória do que um programa equivalente em 32 bits, já que existe espaço perto do CPU para mais dados.

3 | Estratégias para uso eficiente da memória

Neste capítulo descrevem-se estratégias de resolução para os problemas mais comuns relacionados com a utilização de memória em computação científica, quer a nível de otimização dos requisitos quer de minimização de erros.

A otimização de um programa deve geralmente ser tentada apenas após o programa funcionar corretamente, etapa após a qual, como em qualquer cenário de otimização, duas questões se impõem: (1) é necessário otimizar?, (2) se sim, quais os desafios que vale a pena atacar, e por que ordem? A otimização é um dos melhores exemplos do princípio de Pareto³ que diz que, aproximadamente, 80% dos efeitos são criados por 20% das causas. Isto é, podemos escolher otimizar totalmente uma determinada parte do programa e tal não ter impacto praticamente nenhum no rendimento do programa, pois esta é dominada por outra componente do programa. Da mesma forma, é possível modificar apenas uma pequena parte do programa e conseguir uma excelente otimização na performance geral do programa. Para além deste aspeto, a otimização envolve frequentemente um investimento de tempo significativo e alguns compromissos, isto é, melhorar a performance de um programa numa certa dimensão (por exemplo, tempo de execução) pode levar a uma deterioração noutra dimensão (por exemplo, memória utilizada). Os fatores envolvidos nos compromissos da otimização de memória podem incluir o tempo de execução, o tempo de programador, a complexidade e legibilidade do código, o custo, os dados de entrada e de saída, etc. É também especialmente importante ter uma noção clara do que se pretende que o programa faça como, por exemplo, que precisão numérica é necessária, qual o tamanho máximo dos dados de entrada e qual o tempo de execução máximo aceitável. Tal permite ter objetivos e restrições específicos com os quais trabalhar, sendo assim possível estimar os requisitos e pesar as diferentes estratégias. Se, por exemplo, um programa de computação científica deixar de funcionar para estruturas de dados de grande dimensão, a solução mais fácil, e possivelmente a única necessária, pode passar por aumentar a memória do sistema (por exemplo, de 8 GB para 16 GB). Evita-se assim todo o tempo e complexidade associados a outras abordagens. As estratégias a usar dependem também do facto de ser um programa já existente ou um novo, ainda em desenvolvimento. Em geral, por exemplo, fará mais sentido procurar otimizar os algoritmos e estruturas de dados num programa que se escreve de raiz do que num já existente devido à complexidade que tal pode envolver.

As primeiras secções deste capítulo dividem as estratégias entre as que ocorrem a nível do sistema e a nível do código, isto é, as que não envolvem alterar o código-fonte e as que envolvem. A última secção resume todas as estratégias abordadas numa *checklist*.

³ https://en.wikipedia.org/wiki/Pareto_principle (consultado a 24-06-2016).

3.1 Estratégias para uso eficiente de memória a nível do sistema

Nesta secção, descrevem-se diversas estratégias de otimização de memória ao nível do sistema, isto é, que não envolvem alterar o código-fonte.

3.1.1 A transição de 32 bits para 64 bits

Hoje em dia, a maioria dos computadores são sistemas de 32 ou 64 bits (as exceções incluem micro-controladores e *firmware* especializado para dispositivos com poucos requisitos de memória.). No capítulo 2, abordou-se este tema e foram explicadas as diferenças entre os dois sistemas. Em termos práticos, a maior diferença é o facto de os sistemas e programas de 32 bits estarem limitados a usar um máximo de 4 GB de memória. Abaixo, descrevemos como ver se um processador, um sistema operativo ou um programa são de 32 ou 64 bits, e como obter as versões de 64 bits dos programas.

3.1.1.1 Como ver se um processador é de 32 bits ou 64 bits

Uma das maneiras de ver qual o tipo de processador num sistema *Windows* é abrir *Menu Iniciar > Todos os Programas > Acessórios > Ferramentas de Sistema > Informações do Sistema* e ver qual a entrada em *Tipo de Sistema*, conforme ilustrado na Figura 3.1. Em sistemas *linux*, basta ver a entrada 'flags' do ficheiro '/proc/cpuinfo': se a *flag* 'lm' estiver presente é de 64 bits, caso contrário será de 32 bits.

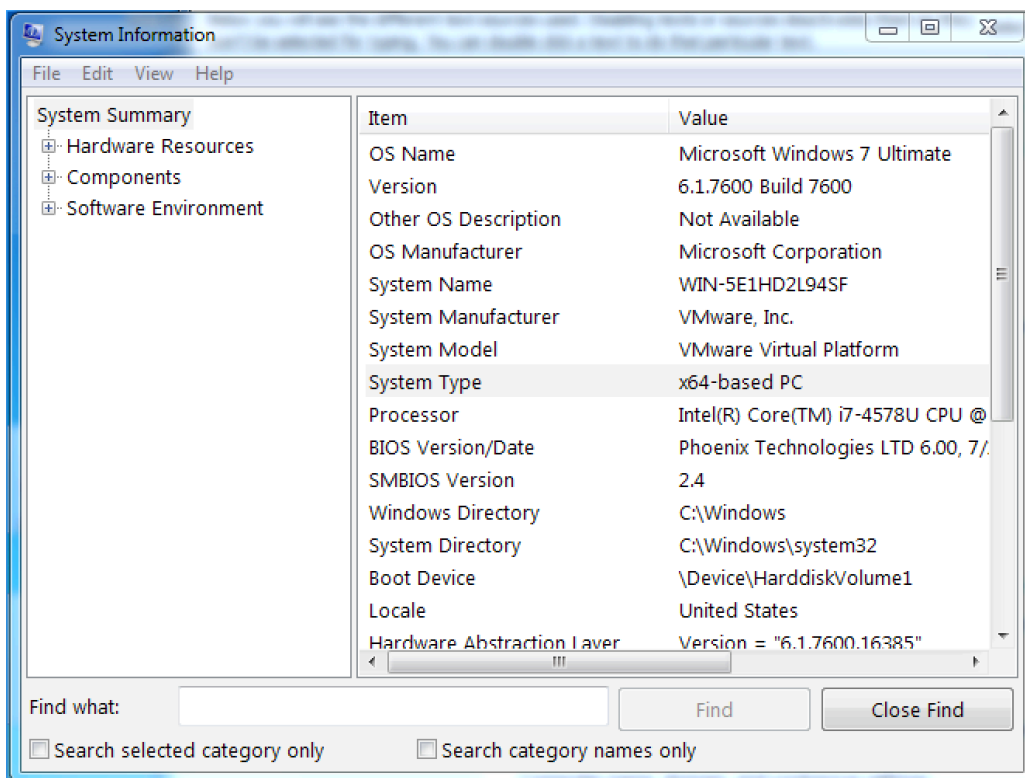


Figura 3.1 – Painel de Informação de Sistema do Windows em que se pode verificar o tipo de processador do sistema

3.1.1.2 Como ver se um sistema operativo é de 32 ou 64 bits

Se o sistema operativo for *Windows* (Vista, 7, 8 ou 10), basta abrir o *Painel de Controlo > Sistema e Segurança > Sistema* e ver qual o valor na entrada *Tipo de sistema*, conforme ilustrado na Figura 3.2. Em sistemas *linux*, basta abrir o ficheiro `/proc/cpuinfo` ou usar o comando `'uname -a'`.

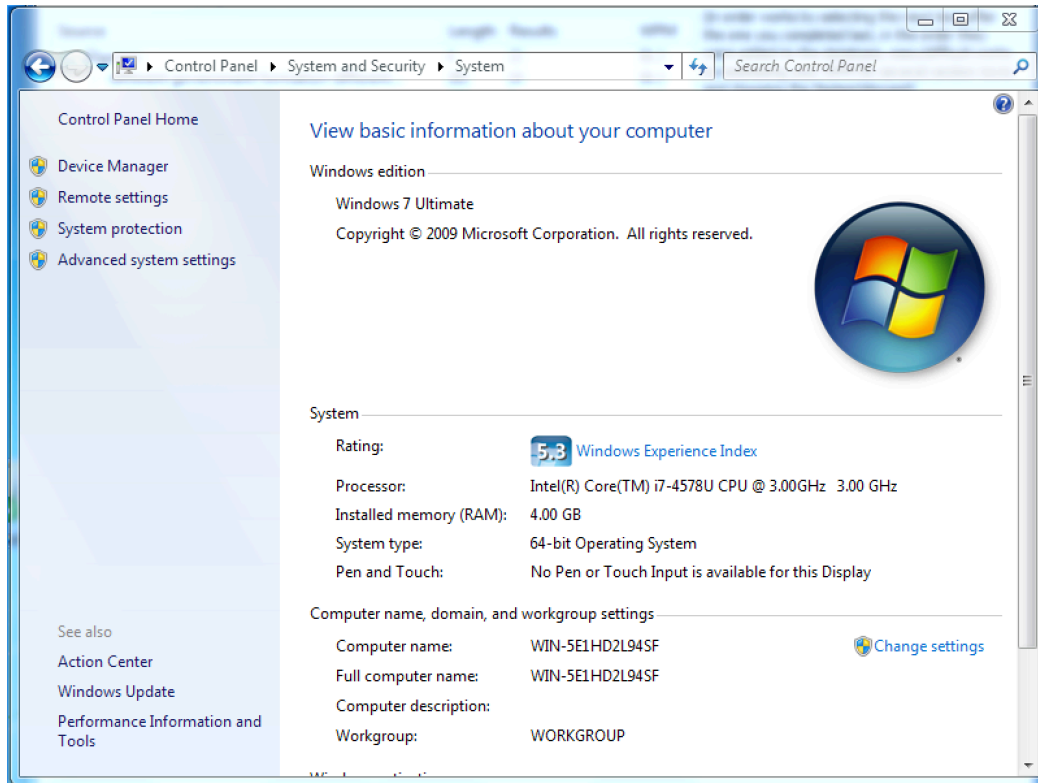


Figura 3.2 – Painel de Controlo do Windows onde se pode ver que versão do Windows está instalada

3.1.1.3 Como ver se um programa é de 32 ou 64 bits

Em sistemas *Windows*, existem vários métodos de verificar se um programa é de 32 ou 64 bits. Ao instalar uma aplicação, o *Windows* guarda os programas de 32 bits na pasta 'Programas (x86)' e guarda na pasta 'Programas' os programas de 64-bit (no entanto, este método não cobre a totalidade dos casos, já que depende de algumas configurações na instalação de cada programa). No *Gestor de Tarefas*, os processos de 32 bits têm geralmente um sufixo '*32', ao contrário dos processos de 64 bits que não têm este sufixo. Este método tem no entanto a desvantagem de obrigar a correr o executável (algo eventualmente não desejável) e, para processos muito rápidos, pode não demorar tempo suficiente para ver o programa na lista de processos. Existem no entanto vários utilitários que o utilizador pode usar para analisar um executável e ver se é de 32 ou 64 bits, como por exemplo o *GNU File for Windows*⁴ ou o *sigcheck*⁵ da SysInternals. Em sistemas *linux*, basta usar no terminal o comando `'file [path do ficheiro]'` e no *output* deste comando estará '32-bit' ou '64-bit', conforme o caso.

⁴ GNU File for Windows, <http://gnuwin32.sourceforge.net/packages/file.htm>.

⁵ sigcheck, <https://technet.microsoft.com/en-us/sysinternals/bb897441.aspx>.

3.1.1.4 Como obter programas de 64 bits

Para correr um programa em 64 bits é necessário que o *hardware*, o sistema operativo e o programa sejam de 64 bits.

Se o processador for de 32 bits, não é possível instalar um sistema operativo de 64 bits. O inverso - instalar um sistema operativo de 32 bits num processador de 64 - é possível (e até relativamente comum), mas não é vantajoso: não será possível utilizar programas de 64 bits, e o sistema operativo e os programas apenas utilizarão 4 GB, mesmo que o computador tenha mais memória disponível.

Tendo instalado um sistema operativo de 64 bits (necessariamente num processador de 64 bits), o programa terá de ser compilado para 64 bits. Na prática, ocorre uma de duas situações: ou o utilizador instala um programa alheio (geralmente, usando um instalador ou um binário) ou o utilizador compila um programa a partir do código-fonte.

Downloading Git

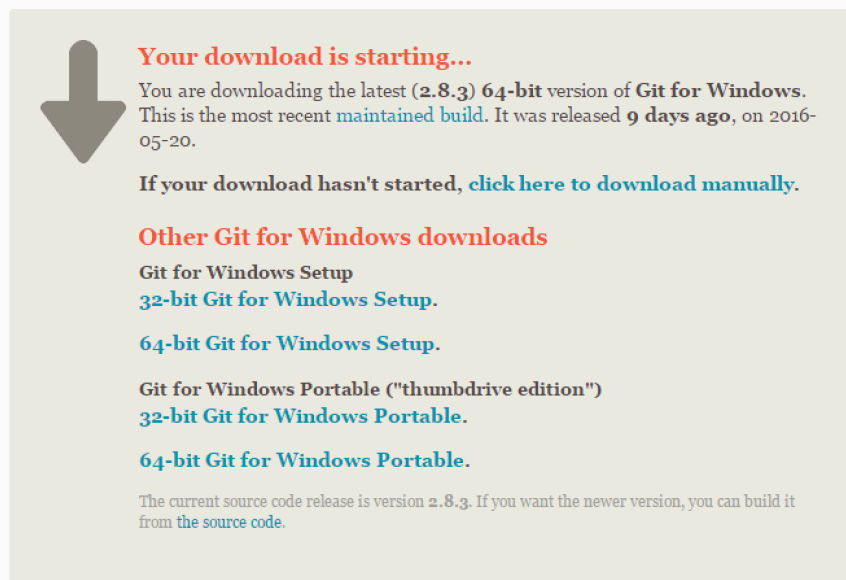


Figura 3.3 – Exemplo de instaladores Windows de 32 e 64 bits, na página de download do Git⁶

No primeiro caso, cabe ao utilizador escolher a opção de 64 bits, caso esta exista. A Figura 3.3 mostra o exemplo de uma página de transferência dos instaladores de um programa em que ambas as versões estão disponíveis.

No caso de ser um programa compilado pelo utilizador a partir do código-fonte, é necessário que o compilador suporte a compilação de 64 bits. Nalguns IDEs⁷, por exemplo o Microsoft Visual Studio⁸, é

⁶ Git, <https://git-scm.com/download/win>.

⁷ IDE – Ambiente de Desenvolvimento Integrado (Integrated Development Environment).

⁸ Microsoft Visual Studio, <https://www.visualstudio.com/>.

possível escolher a versão para a qual queremos compilar, pois o IDE integra vários compiladores. Regra geral, se um programa compilar com sucesso para uma das versões compilará com sucesso para o outra, sem ser necessário alterar o código-fonte.

3.1.2 Adicionar mais memória

Naturalmente, uma solução simples para resolver problemas de restrições de memória pode ser adquirir mais memória, quer correndo o programa nouro sistema com mais memória quer fazendo uma atualização ao sistema original. Hoje em dia, computadores portáteis ou desktops com 16 GB e até 32 GB são cada vez mais comuns e acessíveis, e estações de trabalho com, por exemplo, 128 GB de memória também já são relativamente acessíveis num contexto de computação científica.

3.2 Estratégias para uso eficiente da memória a nível do código

Nesta secção, descrevem-se diversas estratégias de otimização de memória que envolvem alterar o código-fonte.

3.2.1 Uso eficaz da memória dinâmica

Existem dois tipos de memória principal que uma aplicação utiliza quando é executada: a que é determinada no processo de compilação e a chamada memória dinâmica que só é determinada no processo de execução.

O uso adequado da memória dinâmica pode reduzir significativamente os requisitos de memória de um programa, enquanto que o seu uso incorreto pode levar a programas incorretos, inseguros e a um uso excessivo de memória.

Veja-se, como exemplo, o caso das matrizes aloáveis em Fortran. Em Fortran77, o tamanho das matrizes é fixo e tem de ser declarado no código. Como em muitos casos não se sabe o tamanho da matriz, é necessário declarar uma matriz "suficientemente grande". Tal leva a um desperdício de memória na maioria dos casos ou, alternativamente, a recompilações frequentes. No entanto, com o Fortran90 passou a ser possível usar as chamadas matrizes aloáveis (ou dinâmicas). Estas matrizes são também declaradas no código mas a sua dimensão pode ser computada ou dada como entrada, já na execução do programa. A Figura 3.4 mostra um exemplo em Fortran.

```
integer :: n
real, dimension(:), allocatable :: a
[...]

allocate (a(n), stat = AllocationStatus)
if (AllocationStatus /= 0) stop "insufficient memory"
[...]

deallocate (a, stat = DeallocationStatus)
```

Figura 3.4 – Código Fortran90 que declara e aloca uma matriz dinâmica

As duas últimas linhas são especialmente importantes para um programa correto e eficiente. É essencial o utilizador verificar que a alocação foi bem-sucedida, assim como não deixar de libertar a memória dinâmica de que já não necessita (usando a função 'deallocate' em Fortran, por exemplo).

3.2.2 Algoritmos eficientes

A eficiência dos algoritmos é extremamente importante em computação científica, quer seja em termos de memória ou velocidade, e chega a rivalizar, nalgumas aplicações, com o impacto da lei de Moore⁹. Como exemplo, a Figura 3.5 ilustra o impacto da melhoria de *hardware* e de algoritmos na resolução de alguns sistemas lineares esparsos.

⁹ A Lei de Moore (Moore, 1965) diz que “o número de transístores num chip duplica a cada 18 meses mantendo-se o custo constante”, e encapsula o enorme aumento do poder de computação das últimas décadas.

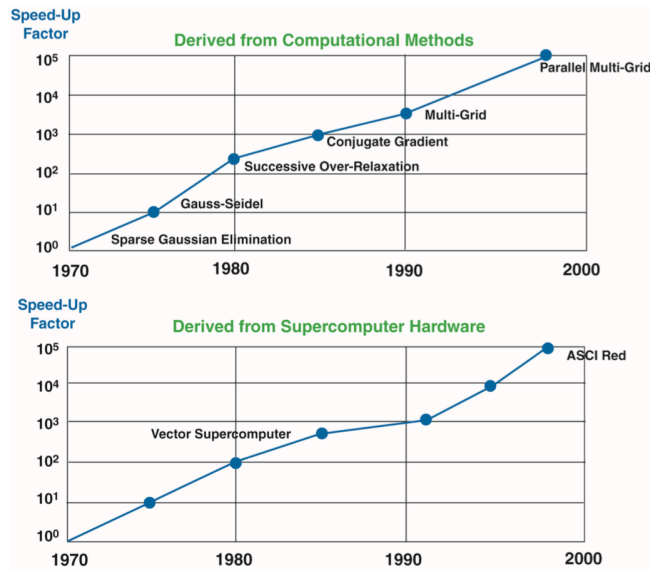


Figura 3.5 – Evolução do fator de *speed-up* para resolução de sistemas lineares esparsos devido a melhorias de algoritmos e de hardware. Fonte: (SIAM Working Group on CSE Education, 2001)

Note-se como, no período representado, o fator de *speed-up* aumentou essencialmente da mesma maneira devido a melhorias nos algoritmos e no *hardware* dos supercomputadores.

O comportamento assintótico de um algoritmo é uma parte importante da análise da eficiência de um algoritmo. Se um algoritmo tiver um tempo de execução proporcional a uma determinada função $f(n)$, em que n representa o tamanho dos dados de entrada, diz-se que um algoritmo corre em tempo $O(f(n))$. Nesta notação, ignoram-se os fatores constantes, isto é, em vez de se dizer, por exemplo, que um programa corre em tempo $3kn^3$ (em que k é uma determinada constante), diz-se apenas que corre em $O(n^3)$. O mesmo se aplica para a memória utilizada pelo programa, isto é, diz-se que um algoritmo ocupa espaço $O(f(n))$. No resto desta secção, abordam-se alguns exemplos específicos que demonstram a importância da eficiência algorítmica.

Considere-se um algoritmo para somar todos os números inteiros de 1 a n . Talvez o algoritmo mais simples seja um ciclo de 1 a n em que se incrementa uma variável de 1 a n e em que, em cada iteração, se adiciona essa variável a uma outra variável que representa a soma cumulativa. Este ciclo demora n iterações, e corre assim em tempo $O(n)$. Isto é, o tempo é assintoticamente linear com o número n : espera-se que se se duplicar o número n , o tempo de corrida seja aproximadamente o dobro. No entanto, existe um algoritmo mais eficiente, já que a soma dos números de 1 a n pode ser expressa na fórmula analítica $n(n+1)/2$. Esta fórmula é obviamente um algoritmo que corre em tempo $O(1)$, isto é, em que o tempo de execução não depende do número n .

No entanto, nem sempre é conveniente considerar apenas o comportamento assintótico de um algoritmo. Efetivamente, uma vez escolhido o algoritmo com melhor comportamento assintótico interessa tentar reduzir o valor das constantes. Considere-se outro exemplo muito simples como ilustração deste ponto: a ordenação de uma lista (ou vetor) de n inteiros. Uma alternativa possível é escolher um algoritmo que crie uma segunda lista vazia que vai preenchendo à medida que ordena a lista original. Este algoritmo ocupa assim a memória de $2n$ inteiros. Alternativamente pode-se usar um

algoritmo que a cada passo apenas troca dois elementos de posição, precisando apenas de $n + 1$ inteiros (para guardar um valor temporariamente), sem criar outra lista. Apesar de estes dois casos serem de igual comportamento assintótico $O(n)$, a constante 2 pode fazer a diferença, sobretudo se este for um dos maiores vetores do programa.

3.2.3 Estruturas de dados compactas

Para além de algoritmos eficientes, é crucial para a otimização de um programa a utilização de estruturas de dados adequadas.

Veja-se um exemplo em Fortran. Na Figura 3.6, estão duas estruturas de dados que pretendem representar uma data e hora.

```

type :: date
  integer :: second
  integer :: minute
  integer :: hour
  integer :: day
  integer :: month
  integer :: year
end type :: date

type :: date
  integer*4 :: seconds
end type :: date

```

Figura 3.6 – Código Fortran que declara duas estruturas de dados para representar uma data e hora

No primeiro caso, usa-se de maneira direta um inteiro para representar cada uma das variáveis (segundos, minutos, horas, dia, mês, ano), enquanto que no segundo caso se recorre a uma representação mais sofisticada: um único inteiro de 32 bits ($4*8$ bytes) que representa o número de segundos desde uma data de referência (por exemplo, desde 1 de janeiro de 1970). Apesar de implicar levar a cabo algumas operações para converter para o formato convencional, esta estrutura de dados permite reduzir os requisitos de memória por um fator de 6. No primeiro caso para cada data usamos $6*4*8\text{bits}=192\text{ bits}=24\text{ bytes}$, enquanto que no segundo caso gastamos apenas $32\text{ bits}=4\text{ bytes}$. Caso tenhamos um milhão de registos, por exemplo, no primeiro caso usamos 24 GB de memória e no segundo apenas 6 GB. Naturalmente deve existir um compromisso entre quão compacta é uma estrutura de dados e a sua precisão numérica ou gama de valores.

3.2.4 Entradas de dados adequadas

Para além de recorrer a algoritmos e estruturas de dados eficientes é importante considerar se a entrada do programa está otimizada e é adequada a estes algoritmos. Considere-se, por exemplo, um programa que resolve sistemas lineares esparsos e em que é utilizado um algoritmo otimizado para matrizes esparsas. Ao tentar resolver o sistema com uma matriz não esparsa, é possível estar

perante um algoritmo mais lento em várias ordens de grandeza. Deve-se assim ter o cuidado de verificar se a entrada de dados é a adequada ao algoritmo a ser usado.

Um exemplo de uma entrada não otimizada pode surgir no caso dos elementos finitos para o cálculo elástico linear tridimensional se se usar uma matriz global. Neste caso, estamos perante uma matriz de banda, que ocupa em memória um espaço de tamanho kn , com k sendo a largura de banda (em vez de n^2 para uma matriz densa). No entanto, a largura de banda k é extremamente dependente da forma como são numerados os nós da malha de elementos finitos. Embora esta diferença seja bastante específica a cada caso, é comum este fator ser de uma ordem de grandeza ou mais, isto é, a mesma malha com o mesmo algoritmo pode ocupar, por exemplo, 1 GB ou 10 GB de memória, apenas por uma diferente numeração dos nós da malha.

3.2.5 Opções de compilador

Alguns erros de memória podem ser detetados imediatamente ao nível da compilação. No entanto, a qualidade e utilidade desta deteção depende da linguagem, do compilador e das opções do compilador. A título de exemplo, recomenda-se a utilização da opção '-fbounds-check' em Fortran que deteta quando o programa tenta aceder a um índice de uma matriz maior do que o maior índice da mesma. Em C, as opções de otimização (por exemplo, a opção '-O2') detetam alguns dos erros de memória.

Recomenda-se usar várias opções dos compiladores. Em geral, a pergunta a fazer não deve ser 'será que devo usar esta opção?', mas sim 'que razão tenho para não usar esta opção?'.

3.2.6 Debuggers de memória

Os depuradores de memória (*memory debuggers*) são programas que detetam fugas de memória (*memory leaks*) e *buffer overflows*, erros que estão relacionados com a alocação e desalocação de memória dinâmica. Existem vários depuradores com características diferentes, dependendo da linguagem de programação, do sistema operativo e da análise feita¹⁰. Em sistemas *linux* (ou em *Windows* usando o *cygwin*), a opção standard é o *Valgrind*¹¹ por funcionar com qualquer linguagem compilada e fornecer uma análise detalhada, quer em termos de análise do perfil quer de erros.

3.2.7 Paralelizar

No caso de as estratégias anteriores não serem suficientes, pode ser necessário recorrer à paralelização do programa. A paralelização, apesar de ser normalmente associada a melhorias de velocidade e tempo de execução, serve na prática sobretudo para resolver problemas maiores, isto é, problemas com requisitos de memória maiores e/ou mais dados de entrada. As chamadas leis de Amdahl (Amdahl, 1967) e de Gustafson (Gustafson, 1988) encapsulam este comportamento.

¹⁰ Para uma lista de depuradores de memória, consultar o anexo A.

¹¹ Valgrind, <http://valgrind.org>.

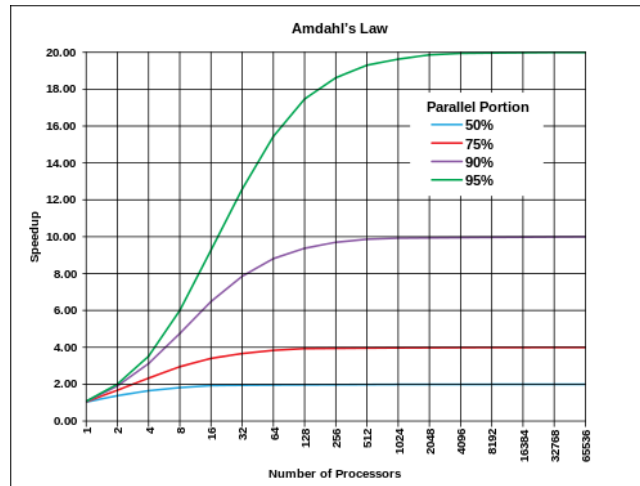


Figura 3.7 – Ilustração da lei de Amdahl. Fator de *speed-up* em função de número de processadores, para diferentes percentagens da parte paralelizável do programa original. Fonte: https://en.wikipedia.org/wiki/Amdahl%27s_law (acedido em 24-06-2016)

Ambas as leis relacionam o *speedup* S de toda a tarefa, s o *speedup* relativo à parte do programa que beneficia da melhoria do sistema e p a percentagem do programa que beneficia da melhoria do sistema antes da melhoria. Com esta notação, a lei de Amdahl é dada por:

$$S = \frac{1}{(1 - p) + \frac{p}{S}}$$

e encontra-se ilustrada na Figura 3.7 para diferentes valores de p , a parte "paralelizável" do programa. Note-se que mesmo para um programa em que a parte paralelizável corresponde a 95% do programa original, o *speedup* tem um limite teórico de 20, independentemente do número de processadores usados. Se à primeira vista este parece ser um resultado algo desanimador, o facto de a análise ter sido feita sobre um programa de tamanho fixo esconde a maior vantagem da paralelização - geralmente não se trata de resolver um dado problema em menos tempo, mas sim um problema significativamente maior num tempo semelhante.

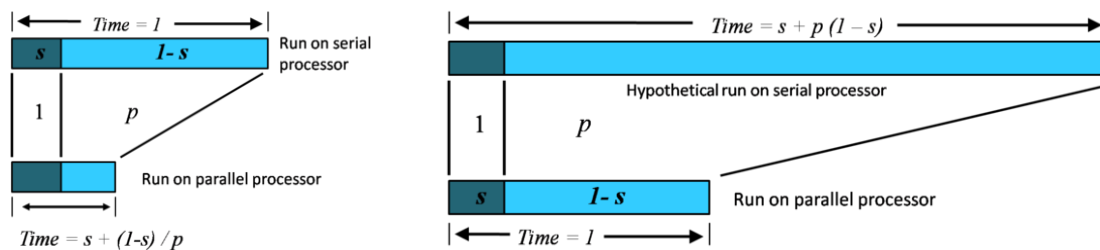


Figura 3.8 – Ilustração da lei de Gustafson. A secção azul-claro do diagrama corresponde à parte paralelizável do programa que é a que beneficia do aumento do número de processadores. Fonte: http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2012/4b_rs (acedido em 24-06-2016)

A Figura 3.8 ilustra este ponto visualmente. Muitas vezes é possível aumentar o tamanho do problema sem praticamente aumentar a parte sequencial do programa. Deste modo, se duplicarmos o número de processadores ao mesmo tempo que duplicamos o tamanho do problema (duplicando a

parte paralelizável), o programa tem aproximadamente o mesmo tempo de execução. A esta observação dá-se o nome de lei de Gustafson que pode ser formulada da seguinte maneira:

$$S = 1 - p + sp$$

O relatório “Computação paralela no LNEC - Guia introdutório e estudo de caso” (Coelho; Inês, 2013) pode ser consultado para mais sobre o tópico da paralelização em computação científica.

3.3 Uma *checklist* para otimizações de memória

As Tabela 3.1 e Tabela 3.2 reúnem as estratégias anteriores e podem ser usadas como uma *checklist* a percorrer, caso seja necessário otimizar um programa em termos de memória. O capítulo seguinte discute a aplicação desta *checklist* a um caso de estudo no LNEC.

Tabela 3.1 – Estratégias para uso eficiente de memória, sem alterar o código

Estratégia	Comentário/descrição
64 bits	Compilar para 64 bits em vez de 32, evitando o limite de 4GB de memória
Mais memória	Adquirir mais memória ou passar para um sistema com mais memória

Tabela 3.2 – Estratégias para uso eficiente de memória, alterando o código

Estratégia	Comentário/descrição
Memória dinâmica	Fazer uso eficiente da memória dinâmica, e libertar a já usada
Algoritmos eficientes	Usar algoritmos eficientes que usem menos memória
Estruturas de dados compactas	Escolher as estruturas de dados mais compactas e eficientes, poupando memória
Entradas de dados adequadas	Utilizar as entradas de dados adequadas aos algoritmos utilizados
Opções de compilador	Otimizar o código e detetar erros usando as opções de compilador
<i>Debuggers</i> de memória	Fazer a análise de perfil da memória e detetar erros
Paralelizar	Resolver problemas maiores do que o possível apenas com um processador

4 | Caso de estudo: cálculo estrutural viscoelástico de barragens de betão usando o método dos elementos finitos

Este capítulo descreve um caso de estudo de otimização de memória de um programa de computação científica. O programa, chamado PAVK, é escrito em Fortran e usa o método dos elementos finitos para realizar cálculos estruturais de barragens de betão (Leitão, 2015).

Na primeira secção, descreve-se brevemente o programa em termos do problema físico que é estudado, da sua modelação matemática e da sua implementação em Fortran. Discute-se na segunda secção o que foi feito em termos de análise de objetivos da otimização e restrições de memória, assim como quais as estratégias adotadas, os resultados da otimização e perspectivas futuras.

4.1 Descrição do programa

O programa PAVK (Programa de Análise Viscoelástica utilizando a cadeia de Kelvin) foi desenhado para permitir realizar cálculos estruturais de barragens de betão mediante a utilização de elementos sólidos com comportamento elástico ou viscoelástico e de elementos de interface para simular as juntas de contração. O processo construtivo é simulado por um algoritmo de ativação e desativação de elementos (*element birth and death*, na literatura inglesa), que correspondem aos diversos blocos de betão. A Figura 4.1 mostra a simulação da construção dos blocos de uma barragem tipo abóboda. Além do peso próprio e das cargas concentradas nos nós, foram implementadas a carga devida à pressão hidrostática (carga de superfície com variação triangular) e as deformações iniciais devidas às variações de temperatura.

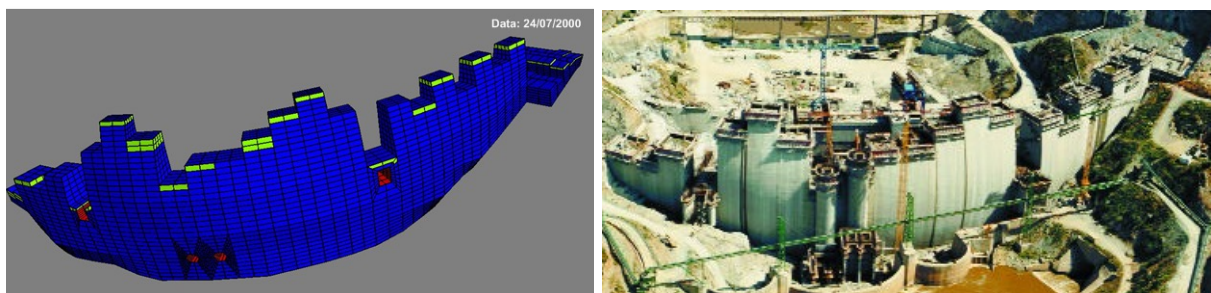


Figura 4.1 – Simulação da barragem de Alqueva. Fonte: (Castilho, 2013)

O programa foi baseado num programa detalhado em (Griffiths; Smith; Margetts, 2013), o qual foi posteriormente modificado por forma a implementar as juntas de contração, as cargas de superfície devido à pressão hidroestática e as deformações devidas às variações de temperatura. A modelação

das barragens tipo abóboda é efetuada mediante uma malha de elementos finitos hexadrais isoparamétricos de 20 nós, que se mostra em esquema na Figura 4.2.

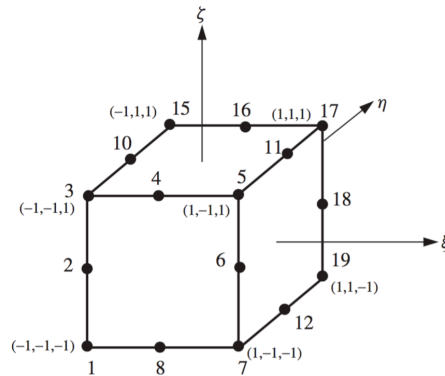


Figura 4.2 – Elemento hexahedral de 20 nós. Fonte: (Griffiths; Smith; Margetts, 2013)

Essencialmente o programa resolve o sistema linear

$$[K][u] = [F]$$

em que $[K]$ é a matriz global de rigidez, $[u]$ o vetor dos deslocamentos e $[F]$ o vetor das forças aplicadas (para mais detalhes, consultar (Griffiths; Smith; Margetts, 2013)). A matriz $[K]$ é uma matriz em banda, simétrica e esparsa (ver Figura 4.3), e o algoritmo usado faz uso destas propriedades ao armazenar a matriz em *skyline*, isto é, usando um vetor que guarda apenas entradas da matriz que pertencem a uma das bandas e à diagonal.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 & \cdots & 0 \\ & A_{22} & A_{23} & A_{24} & \ddots & \vdots \\ & & A_{33} & A_{34} & A_{35} & 0 \\ & & & A_{44} & A_{45} & A_{46} \\ & sym & & & A_{55} & A_{56} \\ & & & & & A_{66} \end{bmatrix}$$

Figura 4.3 – Matriz esparsa, simétrica e em banda

4.2 Aplicação da *checklist*

Antes de optar por qualquer das estratégias referidas no capítulo 3 foi feita uma análise de quais os objetivos da otimização e foram estimados alguns dos requisitos do sistema. O problema principal consistia no facto de o programa não executar para malhas de barragens muito grandes, por falta de memória. Especificamente, queria-se que fosse possível executar o programa com malhas com elementos de 2 metros de altura, que correspondem aproximadamente à altura das camadas de betão durante a construção da barragem. Para grandes barragens (na ordem dos 100 metros de altura) tal corresponde a uma malha de cerca de 100000 nós. Uma das malhas usadas foi a da barragem do Alqueva, ilustrada na Figura 4.4, e para esta malha o programa passou a ocupar mais de 4 GB de memória. O sistema em que estava tinha 8 GB de memória, mas o programa estava

compilado em 32 bits. O tempo de execução não foi considerado uma prioridade, já que as estimativas efetuadas previam tempos razoáveis de execução, mesmo para malhas grandes.

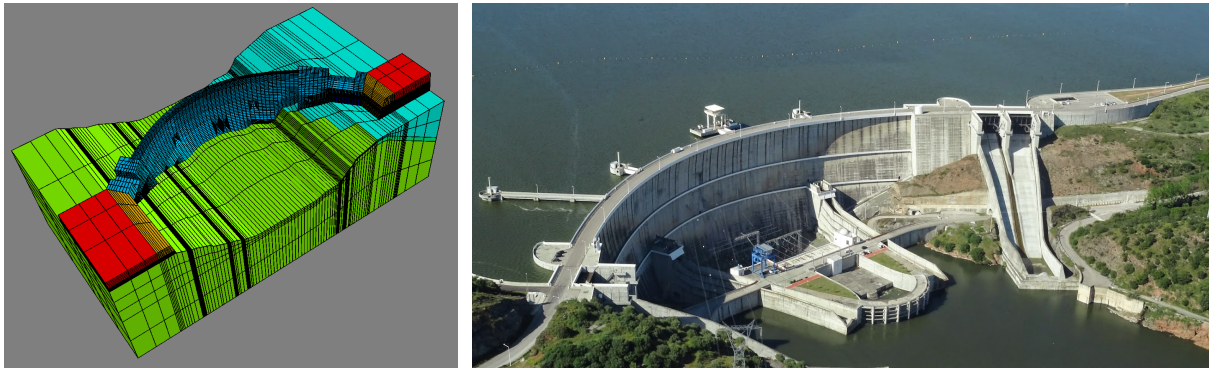


Figura 4.4 – Painel esquerdo: malha de elementos finitos da barragem do Alqueva. Painel direito: fotografia aérea de barragem do Alqueva. Fonte: (Castilho, 2013)

Por se tratar de um programa que já estava escrito (e não um que se iria escrever de raiz), optou-se por começar pelas estratégias mais simples que não envolvem alterar o código. Nomeadamente, as duas primeiras estratégias da *checklist* na Tabela 3.1: compilar o programa em 64 bits e correr o programa num sistema com mais memória.

Por o programa ter sido compilado em Windows usando uma versão do Compaq Visual Fortran que não suporta 64 bits, optou-se por usar outro compilador. Escolheu-se o gfortran¹², por ser um dos melhores compiladores de Fortran ao mesmo tempo que é *software* de utilização livre. O gfortran faz parte do gcc¹³, e por não ter uma versão para Windows foi necessário instalar o cygwin¹⁴, que é um programa de utilização livre que permite executar programas compilados para *linux* num ambiente *Windows*.

A nova versão de 64 bits do programa funcionou corretamente, produzindo os mesmos resultados que a versão de 32 bits, mas não foi possível correr uma malha de uma barragem grande. Ao correr o exemplo da barragem do Alqueva (ver Figura 4.4), que tem cerca de 90000 nós, este bloqueou e não terminou a execução, por falta de memória, que ficou um pouco acima do limite máximo do sistema. Deste modo, tendo já uma versão de 64 bits, adotou-se a segunda estratégia: usar um sistema com mais memória. O sistema utilizado foi o *cerberus*: um equipamento de elevado desempenho do LNEC gerida pelo NTIEC com 64 cores e 256 GB de memória. No *cerberus* o programa foi executado sem qualquer problema com a malha que anteriormente não era possível correr no sistema com 8 GB de memória. A análise feita sugere que o problema inicial foi resolvido e que não será necessária mais nenhuma intervenção, a menos de novos requisitos do programa ou dos dados de entrada.

Apesar de o problema inicial ter sido resolvido com apenas as duas estratégias da Tabela 3.1, aproveitou-se a análise feita e adotou-se uma terceira estratégia por ser de fácil execução e fornecer

¹² gfortran, <https://gcc.gnu.org/fortran/>.

¹³ gcc - GNU Compiler Collection, <https://gcc.gnu.org/>.

¹⁴ Cygwin, <https://www.cygwin.com/>.

uma otimização substancial. Decidiu-se aplicar um algoritmo à malha da barragem que permitisse reordenar os nós e reduzir a largura de banda da matriz. Sendo a matriz armazenada em 'skyline', a redução da largura de banda traduz-se numa redução da memória utilizada (e também numa redução do tempo de execução). O algoritmo utilizado foi o Reverse Cuthill-McKee¹⁵ que foi aplicado à malha antes de o sistema linear ser resolvido. Esta alteração pode ser considerada um misto de 3 das estratégias da Tabela 3.2, nomeadamente o uso de algoritmos mais eficientes, de estruturas de dados compactas e de dados de entrada adequados. Desta otimização resultou uma ordenação mais eficiente dos nós da malha. Uma das malhas testada foi a da barragem de Foz Tua de 29296 nós. Para a banda correspondente ao cálculo térmico (1 grau de liberdade por nó), a largura de banda passou de 7179 para 2211, o que reduziu os requisitos de memória em cerca de 70%. Quanto ao cálculo efetuado com PAVK (3 graus de liberdade por nó), esta malha de Foz Tua não tem problemas de memória, mas demorava aproximadamente 28 minutos a resolver o sistema de equações. O sistema formado por 79017 equações (alguns nós tendo os deslocamentos restringidos, não entram no sistema a resolver), passou de "skyline storage" de 200359281 valores com a ordenação inicial para 103699479 com a nova numeração (isto é, cerca de metade do espaço) e o tempo de resolução do sistema passou a, aproximadamente, 8 minutos.

Caso venha a ser necessário, outras estratégias se podem adotar. Tendo em conta que o programa é baseado num código (o programa 5.4 de (Griffiths, Smith, Margetts; 2013)) já bastante pensado e sem nenhuma melhoria óbvia (como o uso incorreto da memória dinâmica, por exemplo), há duas abordagens futuras que merecem ser destacadas. A primeira consiste em usar um algoritmo mais eficiente como, por exemplo, um método iterativo que não necessite de tanta memória como o método com uma matriz de rigidez global. O programa 5.6 de (Griffiths, Smith, Margetts; 2013) é um exemplo, que usa a técnica *elemento-a-elemento* em vez de assemblar uma matriz global de rigidez, reduzindo os requisitos de memória. A segunda hipótese consiste na paralelização que é essencialmente a única hipótese para problemas suficientemente grandes e cuja eficácia é explicada na secção 3.2.7.

¹⁵ Foi usado o código Fortran que implementa o algoritmo Reverse Cuthill-McKee (George; Liu, 1981), disponibilizado em http://people.sc.fsu.edu/~jburkardt/f_src/rcm/rcm.html (acedido a 24-06-2016).

5 | Conclusões

Neste trabalho discutiu-se o uso eficiente da memória em computação científica. Descreveu-se um modelo simples de um computador que permite compreender melhor o comportamento e fatores que influenciam a performance de um programa e foi discutida uma lista de diferentes estratégias para uso eficiente da memória.

Estas estratégias são importantes porque permitem correr programas que ocupam menos memória, têm um menor tempo de execução, processam uma maior quantidade de dados de entrada e têm menos erros, enquanto poupam tempo de programador. No entanto, conforme a situação específica uma determinada estratégia pode ser eficaz ou irrelevante. É assim necessário medir, estimar e avaliar as características do programa e do sistema, assim como os objetivos e restrições do problema que se pretende resolver, por forma a obter uma ideia clara do valor de cada uma das estratégias.

Foi discutido um caso de estudo em que uma análise das diferentes estratégias permitiu resolver os problemas existentes de limitação de memória sem intervenção a nível do código-fonte. Ao recompilar o programa numa versão de 64 bits (em vez de 32) e mudar para um sistema com mais memória, evitou-se o tempo de programação e a complexidade envolvida noutra tipo de estratégias.

Forneceu-se uma *checklist* das estratégias a abordar como forma de resumir, clarificar e ajudar a adoção das mesmas, reproduzida na Tabela 5.1 e Tabela 5.2.

De entre os vários itens da *checklist*, destaca-se o uso da versão de 64 bits de um programa (em vez da de 32). É uma estratégia fácil de implementar que elimina o limite de 4 GB de memória das versões de 32 bits.

Tabela 5.1 – Estratégias para uso eficiente de memória, sem alterar o código

Estratégia	Comentário/descrição
64 bits	Compilar para 64 bits em vez de 32, evitando o limite de 4GB de memória
Mais memória	Adquirir mais memória ou passar para um sistema com mais memória

Tabela 5.2 – Estratégias para uso eficiente de memória, alterando o código

Estratégia	Comentário/descrição
Memória dinâmica	Fazer uso eficiente da memória dinâmica, e libertar a já usada
Algoritmos eficientes	Usar algoritmos eficientes que usem menos memória
Estruturas de dados compactas	Escolher as estruturas de dados mais compactas e eficientes, poupando memória
Entradas de dados adequadas	Utilizar as entradas de dados adequadas aos algoritmos utilizados
Opções de compilador	Otimizar o código e detetar erros usando as opções de compilador
<i>Debuggers</i> de memória	Fazer a análise de perfil da memória e detetar erros
Paralelizar	Resolver problemas maiores do que o possível apenas com um processador

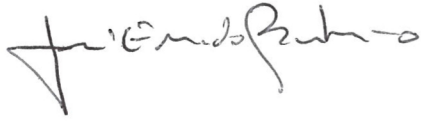
Agradecimentos

Os autores agradecem a colaboração da Investigadora Principal Noemi Schclar Leitão e da Bolsista de Iniciação à Investigação Eloísa Castilho dos Santos, do Núcleo de Observação do Departamento de Barragens de Betão, pela disponibilização do programa original, pela colaboração no caso de estudo e pelas contribuições para o desenvolvimento deste relatório.

Lisboa, LNEC, junho de 2016

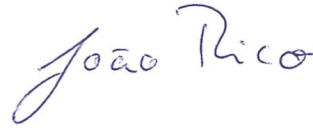
VISTO

O Chefe do Núcleo de Tecnologias da
Informação em Engenharia Civil



José Barateiro

AUTORIA



João Rico

Bolseiro de Iniciação à Investigação Científica



António Inês Silva
Investigador Principal

Referências Bibliográficas

- AMDAHL, Gene M., 1967 – **Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities**. AFIPS Conference Proceedings (30): 483–485.
- BRYANT, Randal; O'HALLARON, David, 2010 – **Computer Systems: A Programmer's Perspective**. 2ª edição, Pearson. ISBN: 0136108040.
- CASTILHO, Eloísa, 2013 – **Análise térmica de barragens de betão durante a construção. Aplicação à barragem de Alqueva**. Dissertação de mestrado, Instituto Superior Técnico, Universidade de Lisboa.
- COELHO, João; INÊS, António, 2013 – **Computação paralela no LNEC – Guia introdutório e estudo de caso**. LNEC - Relatório 237/2013 – CD/NTIEC.
- GEORGE, Alan; LIU, Joseph, 1981 – **Computer Solution of Large Sparse Positive Definite Matrices**. Pearson, ISBN: 0131652745
- GUSTAFSON, John L., 1988 – **Reevaluating Amdahl's Law**. Communications of the ACM 31(5), 1988. pp. 532-533.
- LEITÃO, Noemi, 2015 – **PAVK – Programa de Análise Viscoelástica utilizando a cadeia de Kelvin** [software]. LNEC – DBB/NO.
- MOORE, Gordon E., 1965 – **Cramming more components onto integrated circuits**. Electronics Magazine. p. 4.
- SIAM WORKING GROUP ON CSE EDUCATION, 2001 – **Graduate Education in Computational Science and Engineering**. (p. 168) SIAM Rev., 43(1), 163–177
- SMITH, I. M.; GRIFFITHS, D. V.; MARGETTS, L., 2013 – **Programming the Finite Element Method**. 5ª edição, Wiley. ISBN: 111997334

ANEXO
Lista de *debuggers* de memória

Lista de *debuggers* de memória. Fonte: https://en.wikipedia.org/wiki/Memory_debugger (consultado em 24-06-2016).

Name	OS	License	Languages	Technique
AddressSanitizer	Linux, Mac OS	Free/open source (LLVM)	C, C++.	Compile-time instrumentation (available in Clang and GCC) and specialized library
Allinea DDT	Linux, Blue Gene	Proprietary commercial	C, C++ and F90. Also for parallel programs on supercomputers	Runtime - through dynamic linking
Alloclave	Windows	Proprietary commercial	C++	Compile-time
AQtime	Windows (Visual Studio, Embarcadero IDEs)	Proprietary commercial	.NET, C++, Java, Silverlight, JScript, VBScript	Runtime
Bcheck	Solaris			
BoundsChecker	Windows (Visual Studio)	Proprietary commercial	C++	Runtime intercepts or compile-time
checker (deprecated in favor of valgrind)	Linux, Solaris	Free/open source (GPL)	C	Compile-time override
D Profiler	Windows	Free/open source	C, C++	Runtime. Support allocation profiling, leak detection. Also support CPU, I/O and lock profiling.
Daikon	Unix, Windows, Mac OS X	Free/open source	Java, C/C++, Perl, and Eiffel	Runtime dynamic invariant detection
DDDebug	Windows (Borland Delphi)	Commercial	Delphi (5 to XE7)	Runtime
Debug_new	(general technique)	(general technique)	C++	Compile-time override
Deleaker	Windows (Visual Studio)	Proprietary commercial	C / C++	Runtime intercepts
dmalloc	Any	Free/open source (public domain)	C	Compile-time override
dmalloc	Any	Free/open source	C	Compile-time override
Dr Memory	Windows, Linux	Free/open source (LGPL)	C, C++	Runtime
Duma (fork of Electric Fence)	Unix, Windows	Free/open source (GPL)	C, C++	Compile-time override
Electric Fence	Unix	Free/open source (GPL)	C, C++	Compile-time override
grapprof	GNU/Linux	Free/open source (GPL)	C, (C++)	Link-time override
HP Wildebeest (WDB)				Based on gdb
IBM Rational Purify	Unix, Windows	Proprietary commercial	C++, Java, .NET	Runtime
Insure++	Windows (Visual Studio plugin), Unix	Proprietary commercial	C, C++	
Intel Parallel	Windows	Proprietary	C, C++	

Inspector	(Visual Studio)	commercial		
libcwd	Linux (gcc)	Free/open source	C, C++	Compile-time override
libumem	Solaris	Bundled with Solaris		Link-time override
MemCheck (Hal Duston)	Unix, Mac OS X	Free/open source (GPL)	C, C++	Compile-time override
Memcheck (Soci�t� � G�n�rale)	Windows (Borland Delphi)	Free/open source	C, C++	Compile-time override
MemPro	Windows (Visual Studio)	Free beta	C++	Compile-time override, uses TCP connection to track memory events
Memwatch	Any (programming library)	Free/open source	C	Compile-time override
mpatrol ^[11]	Unix, Windows	Free/open source (LGPL)	C, C++	
mtrace	Various	Free/open source (LGPL)	GNU C library	Built-in, outputs accesses
MTuner	Various	Proprietary commercial	C, C++	Runtime intercepts, Link-time override (MSVC, Clang and GCC), Leak detection
ocp-memprof	Linux 64	Proprietary/open-source	OCaml	Compile-time
OpenPAT	Any	Free (requires registration)	Any compilable language including C, C++, Java and Fortran	Compile-time instrumentation, runtime intercepts
Oracle Solaris Studio (formerly Sun Studio Runtime Checking)	Linux, Solaris	Proprietary freeware	C, C++, Fortran	
softwareverify	Windows	Proprietary commercial	.NET, C, C++, Java, JavaScript, Lua, Python, Ruby, VMs	Runtime
Splint	Any	Free/open source (GPL)	C	Static program analysis
TotalView	Unix, Mac OS X	Proprietary commercial	C, C++, Fortran	Runtime
Valgrind	Linux, Mac OS, Android	Free/open source (GPL)	Any	Runtime intercepts
Visual Leak Detector	Windows (Visual Studio)	Free/open source (LGPL)	C, C++	Compile-time overrides, leak detection
WinDbg	Windows	Proprietary freeware	C, C++, .NET, Python	Runtime

