LABORATÓRIO NACIONAL
DE ENGENHARIA CIVIL

CONSELHO DIRETIVO
Núcleo de Tecnologias da Informação
em Engenharia Civil

Proc. 1302/044

COMPUTAÇÃO PARALELA NO LNEC

Guia introdutório e estudo de caso

Lisboa • julho de 2013

I&D CONSELHO DIRETIVO

RELATÓRIO 237/**2013 – CD/NTIEC**

Resumo

Este trabalho introduz os princípios da computação paralela tendo em vista a sua aplicação como ferramenta de computação científica, em particular para acelerar a execução e alargar o escopo de aplicação de programas de cálculo científico. São introduzidos os principais conceitos relativos à computação paralela, as diferentes arquitecturas de máquinas paralelas e os diferentes modelos de paralelização que se podem adoptar, com destaque para o modelo de *message-passing* e a ferramenta MPI. São ainda descritos os métodos de operação com o *cluster* do LNEC, o Medusa, e um estudo de caso que ilustra a aplicação de MPI e de uma estratégia de paralelização à resolução de um problema real.

Abstract

In this work we introduce the fundamentals of parallel computing as a tool for scientific computing, particularly regarding performance improvement and application scope enhancement of scientific programs. We introduce the main relevant concepts and architectures in parallel computing, as well as the distinct parallelization models which may be adopted, highlighting the message-passing approach and the MPI tool. We also describe standard operational methods for the LNEC cluster, Medusa, along with a case-study demonstrating the usefulness of using both MPI and a structured parallelization strategy in solving a real problem.

Índice

Resumo . . . 3
Abstract . . . 3
Lista de Tabelas . . . 5
Lista de Figuras . . . 5

1 Introdução **7**

2 Computação Paralela **9**
2.1 Definição . . . 9
2.2 Conceitos . . . 9
2.2.1 Fundamentos . . . 9
2.2.2 Arquitetura e comunicação . . . 11
2.2.3 Limites da paralelização . . . 13
2.2.4 Modelos de paralelização . . . 15
2.3 Estado da arte . . . 17
2.4 MPI . . . 17
2.5 Estratégia de paralelização . . . 19
2.5.1 Abordagem inicial . . . 19
2.5.2 Paralelização . . . 20
2.5.3 Dicas . . . 22

3 Computação Paralela no LNEC **25**
3.1 Medusa . . . 25

4 Estudo de Caso: Interbath **27**
4.1 Definição do problema . . . 27
4.2 Ferramentas utilizadas . . . 28
4.2.1 ParMETIS . . . 28
4.2.2 K-D Tree . . . 29
4.2.3 *Timing* . . . 30
4.2.4 *Debugging* . . . 30
4.2.5 Análise . . . 30
4.3 Paralelização . . . 31
4.4 Conclusões . . . 37

5 Trabalho Futuro **39**

Bibliografia **41**

Lista de Tabelas

2.1 Concorrência entre processos. 10
2.2 Sincronização de processos com uma barreira. 10
2.3 Classificação de arquiteturas paralelas. 12
2.4 *Speedup* em função de P. 13
2.5 *Speedup* em função de N. 14
2.6 N em função de P para $S = 24$. 14
2.7 Preservação da reversibilidade durante a paralelização. 22

3.1 Principais comandos para utilização do Medusa. 26

4.1 Malhas de batimetria utilizadas. 28
4.2 Tempo de execução em função da malha de *background*. 28
4.3 Versões paralelas do *Interbath*. 32
4.4 Paralelismo na versão paralela v1. 32

Lista de Figuras

2.1 Memória partilhada. 12
2.2 Memória distribuída. 12

4.1 Tempos de execução (v1). 33
4.2 Tempos de execução para a malha BGround2 (v3). 35
4.3 Tempos de execução para as malhas Batim e BGround2 (v3.2). 36
4.4 Tempos de execução para as malhas Batim e BGround2 (v3.3 e v3.3.2). 37

Capítulo 1

Introdução

Não é nova a noção de que a evolução registada desde a década de 70 em matéria de *hardware* sequencial, que tem dado origem a uma duplicação da capacidade de processamento neste tipo de máquinas sensivelmente a cada 18 meses, está a atingir o seu limite. Esta constatação, juntamente com fatores práticos e económicos, tem estado na origem da tendência crescente de associação de máquinas sequenciais em aglomerados como forma de obter maior capacidade de processamento, e na utilização de computação paralela para explorar o potencial destas estruturas. O LNEC deu resposta a essa tendência com a aquisição de um *cluster* para computação paralela, o Medusa, um recurso que não é ainda aproveitado ao máximo pelos investigadores do laboratório.

Este relatório foi elaborado no contexto da execução de uma bolsa de iniciação à investigação no LNEC, na área da computação paralela, com dois objetivos: em primeiro lugar, pretende reunir os conhecimentos adquiridos durante a execução desta bolsa sob a forma de um guia de iniciação à programação paralela no ambiente do LNEC, de forma a permitir que futuros investigadores interessados em paralelizar os seus programas tenham acesso a um recurso introdutório relativamente simples e adaptado ao seu contexto. Este objetivo é particularmente importante por duas razões: por um lado, o *cluster* do LNEC para computação paralela, o Medusa, é relativamente recente e subaproveitado; por outro lado, a tendência de crescimento do paradigma paralelo torna expectável que o número de investigadores com interesse por esta matéria, particularmente em iniciar-se nela, venha a aumentar.

O segundo objetivo consiste em documentar o trabalho realizado até ao momento no âmbito desta bolsa; em particular, pretende-se descrever com algum detalhe a abordagem a um problema específico, o *Interbath*, tendo em vista a sua paralelização. Os dois propósitos são em larga medida coincidentes, sendo a paralelização do *Interbath* usada como estudo de caso no capítulo 4.

A estrutura do relatório divide-se essencialmente em três secções:

- Computação Paralela - esta secção pretende introduzir os fundamentos básicos do paralelismo, descrever sucintamente o estado da arte e apresentar uma estratégia geral para a paralelização, com base na experiência do trabalho desenvolvido durante a bolsa.
- Computação Paralela no LNEC - aqui pretende-se descrever os recursos de computação paralela no LNEC, nomeadamente o Medusa, e introduzir os fundamentos para a sua utilização.
- Estudo de Caso - *Interbath* - esta parte visa descrever em detalhe o trabalho realizado no âmbito da paralelização do programa *Interbath*, incluindo as ferramentas utilizadas, a abordagem adoptada e a análise dos resultados obtidos.

Capítulo 2

Computação Paralela

2.1 Definição

Define-se computação paralela como a utilização simultânea de mais do que um processador (*core*) para resolver um problema computacional, por oposição à tradicional computação em série em que apenas um processador é utilizado em cada momento. A sua aplicabilidade assenta no princípio de que é possível subdividir um problema de grandes dimensões em vários subproblemas de menor dimensão que podem ser resolvidos concorrentemente. A motivação para recorrer a este tipo de computação prende-se com duas necessidades fundamentais:

1. Minimizar o tempo de execução
2. Resolver problemas de maiores dimensões

Estes dois fatores são usualmente apontados como os objetivos principais do recurso à paralelização, como resposta a dois problemas frequentes da computação série: por um lado, a sua excessiva lentidão no tratamento de problemas de complexidade elevada; por outro, as restrições que a utilização de uma única máquina coloca em termos de memória disponível e que afetam a sua capacidade de resolver problemas de grandes dimensões. Consoante o setor de atividade, outras motivações podem surgir como relevantes, tais como a redução de despesas (tipicamente relacionadas com energia, e portanto associadas à redução do tempo de computação) ou a necessidade de utilizar recursos não locais[1].

2.2 Conceitos

Existem vários conceitos essenciais associados à computação paralela que é importante introduzir. Nesta secção são descritos em primeiro lugar os fundamentos da paralelização e a sua terminologia; posteriormente, são apresentadas as diferentes arquiteturas paralelas e os modos de comunicação entre elas. São ainda expostos os limites teóricos da paralelização e os diferentes modelos de paralelização usualmente adotados.

2.2.1 Fundamentos

É importante começar por definir alguns conceitos básicos relacionados com a paralelização, não só por serem utilizados repetidamente ao longo deste trabalho mas sobretudo por se tratarem de elementos importantes desta forma de computação.

Designamos por **processador** (ou CPU) a estrutura básica de computação, que pode incluir um ou vários *cores*; um conjunto de processadores, juntamente outras estruturas de base (memória e interface de rede, por exemplo), forma um **nó**. Em cada processador é executado um **processo**. Um supercomputador é tipicamente formado por um conjunto de nós, e pode ser um ***cluster***, se esse conjunto for aproximadamente homogéneo e estiver localizado no mesmo local, ou uma ***grid***, por exemplo se não houver homogeneidade entre os nós, caso estes estejam dispersos geograficamente, ou ainda se pertencerem a domínios administrativos distintos.

É frequente introduzir a distinção entre dois tipos de paralelismo, ***data parallelism*** e ***task parallelism***, consoante o método de paralelização usado ([1]). No essencial, para cada processador, a

[1] Por exemplo, no caso de uma *grid* de recursos geograficamente dispersos em que cada unidade dispõe de informação não acessível às restantes.

execução da mesma tarefa sobre diferentes conjuntos de dados constitui uma forma de *data parallelism*, ao passo que a execução de diferentes tarefas - quer seja sobre conjuntos de dados diferentes ou não - se classifica como *task parallelism*. Na prática esta distinção é quase exclusivamente formal, uma vez que o modelo de paralelização adotado reside frequentemente algures num contínuo entre estes dois extremos.

Em qualquer caso, a partir do momento em que mais do que um processador está envolvido na resolução de uma determinada tarefa, é praticamente certo que existirá necessidade de trocar informação entre processadores; para isso, é preciso criar a estrutura que permita essa troca de informação, ou **comunicação**, entre processadores. Os processos de comunicação tendem a aumentar o tempo de execução do programa, pelo que é recomendável reduzir o volume total de comunicações ao mínimo. Neste contexto, o conceito de **granularidade**, definido (qualitativamente) como rácio entre o tempo de CPU dispendido em atividades de computação *versus* de comunicação, $G = \frac{T_comp}{T_comm}$, surge como uma razoável medida do peso relativo da comunicação no programa, que em geral se classifica como de granularidade grosseira caso processe volumes elevados de computação entre eventos de comunicação ($G >> 1$), ou de granularidade fina caso esses volumes sejam relativamente pequenos ($G \approx 1$).

A conceção tradicional de um programa assenta numa lógica sequencial - isto é, as diversas instruções são executadas pela mesma ordem em que estão presentes no código-fonte, de forma previsível e determinada. Pelo contrário, num programa em paralelo, os vários processos são executados de forma concorrente, não havendo maneira de prever qual deles executará uma determinada instrução primeiro; a lógica sequencial é preservada apenas localmente (*i.e.*, para cada processo). Por exemplo, o seguinte fragmento de código produz, em três chamadas diferentes, outros tantos *outputs*:

```
if (rank == 0) str='isto'
if (rank == 1) str='é'
if (rank == 2) str='uma'
if (rank == 3) str='frase'

do i=0,3
  if (i == rank) print*, str
end do
```

(a) Input

Run 1	Run 2	Run 3
é	é	uma
isto	uma	frase
frase	isto	é
uma	frase	isto

(b) Output

Tabela 2.1: Concorrência entre processos.

Este tipo de comportamento torna-se problemático quando é necessário trocar informação entre processos, pois é impossível saber *a priori* se essa informação estará já disponível no processo emissor quando for solicitada pelo recetor. Para evitar esse problema é necessário recorrer à **sincronização** de processos, ou seja, à coordenação dos processos em tempo real; a solução mais frequente para isso é introduzir uma barreira, *i.e.*, uma instrução para que o processo espere até que outros processos alcancem esse ponto do programa.

```
if (rank == 0) str='isto'
if (rank == 1) str='é'
if (rank == 2) str='uma'
if (rank == 3) str='frase'

do i=0,3
  if (i == rank) print*, str
  call MPI_Barrier
end do
```

(a) Input

Run 1	Run 2	Run 3
isto	isto	isto
é	é	é
uma	uma	uma
frase	frase	frase

(b) Output

Tabela 2.2: Sincronização de processos com uma barreira.

A **escalabilidade** de um programa paralelo define-se como a sua capacidade de, mediante um aumento dos recursos disponíveis (por exemplo, mais processadores), exibir um aumento proporcional da eficiência em termos de tempo de execução. No caso ideal de um programa ser perfeitamente

escalável, é de esperar que um aumento do número de processadores para o dobro reduza o tempo de execução para metade. Vários fatores contribuem para a escalabilidade do programa, incluindo especificações de *hardware* e da rede de comunicação; de entre aqueles que são controláveis pelo programador, os mais importantes para garantir máxima escalabilidade são o volume de comunicações, que deve ser mantido no mínimo, e a distribuição de carga computacional pelos processadores, que deve ser o mais equilibrada possível.

Em geral, o tempo de execução de um programa série diz respeito apenas ao tempo despendido nas tarefas de computação. Ao introduzir paralelismo, será necessário gastar mais tempo na gestão das tarefas exclusivamente paralelas; a este tempo extra, que não está relacionado com a execução de trabalho útil mas apenas com a gestão do paralelismo, é dado o nome de ***overhead* paralelo**. Vários fatores podem contribuir para o aumento deste *overhead*:

- Comunicações entre processos - com tendência crescente para um maior número de processos;
- Tempo de espera por sincronizações;
- Inicialização e encerramento do ambiente paralelo.

O *overhead* paralelo pode ser visto como um custo mínimo da introdução de paralelização, que na prática é o que impede que a melhoria de *performace* mediante o aumento do número de processadores seja linear[2].

Tem sido referida repetidamente a necessidade de minimizar o tempo de execução, o que pressupõe, em primeiro lugar, a sua medição, o que não é trivial; de facto, existem duas medidas relevantes para a contabilização do tempo: tempo de relógio e tempo de CPU.

O **tempo de relógio**[3] corresponde ao tempo real que o programa demora a executar, a mesma medição que seria obtida usando um cronómetro. Esta é a métrica mais importante e a que deve ser usada para comparar os desempenhos de versões série e paralela do mesmo programa, uma vez que é a que incide sobre o objetivo da paralelização: minimizar o tempo *real* de execução. No entanto, esta medida é afetada por tudo o que possa estar a acontecer no sistema no momento da computação - sobrecarga do CPU com outros processos, por exemplo - pelo que não é a melhor opção para comparar diferentes corridas de uma mesma versão sujeita a diferentes condições, como por exemplo a *performance* dessa versão para diferente número de processadores.

Para isso, uma melhor opção é o **tempo de CPU**, que contabiliza apenas o tempo que a máquina despende a executar o código (incluindo chamadas ao I/O) e por isso é menos afetado por fatores externos à computação. Para um único processador, o tempo de CPU é necessariamente inferior ao tempo de relógio (no limite será igual). Em computação paralela é frequente calcular o tempo de CPU como a soma dos tempos individuais despendidos por cada processador (**tempo de CPU unitário**) que deve, em teoria, ser independente do número de processadores a menos do *overhead* de paralelização. Por vezes é também útil usar o tempo de CPU unitário, por exemplo comparando-o entre os vários processadores para confirmar se a carga computacional se encontra bem distribuída.

2.2.2 Arquitetura e comunicação

Existem várias classificações possíveis para as máquinas de computação paralela. Uma classificação básica, definida em função do nível em que o *hardware* suporta paralelismo, compreende duas categorias: sistemas *multi-core*, caso um único CPU disponha de mais do que um *core*; ou sistemas como *clusters* ou *grids*, em que várias máquinas diferentes são usadas para resolver uma mesma tarefa.

A distinção mais comum entre arquiteturas de máquinas paralelas, usualmente designada por *Taxonomia de Flynn*, é definida de acordo com as características da máquina segundo duas dimensões independentes, o **fluxo de instruções** e o **fluxo de dados** passados à máquina em cada ciclo de relógio; cada uma destas dimensões compreende apenas dois estados, singular e múltiplo. Desta forma fica definida a matriz de classificação de arquiteturas da tabela 2.3.

[2]Considerando o caso ideal de um programa 100% paralelizável.
[3]Alternativamente, *wall-clock time* ou *elapsed time*.

SISD	SIMD
fluxo de instruções singular fluxo de dados singular	fluxo de instruções singular fluxo de dados múltiplo
MISD fluxo de instruções múltiplo fluxo de dados singular	**MIMD** fluxo de instruções múltiplo fluxo de dados múltiplo

Tabela 2.3: Classificação de arquiteturas paralelas.

Nesta classificação a terminologia é bastante intuitiva. As máquinas **SISD**[4] correspondem aos computadores série tradicionais (um sistema não-paralelo); por cada ciclo de relógio, recebe apenas uma instrução[5] e um conjunto de dados, que são executados de forma determinística. As máquinas **SIMD** são já um tipo de computador paralelo, em que todos os processadores executam o mesmo conjunto de instruções mas sobre conjuntos de dados que podem ser distintos; são ideais para resolver problemas com elevada regularidade e separabilidade, como por exemplo o tratamento de imagens com *GPUs*, e a execução é determinística e feita de forma síncrona. Não são frequentes as máquinas de tipo **MISD**, em que em teoria cada *core* processa um conjunto diferente de instruções sobre o mesmo conjunto de dados; uma possível aplicação para este tipo de máquina seria, por exemplo, ter vários algoritmos criptográficos distintos a tentar quebrar a segurança de uma única mensagem. Por último, os computadores paralelos de tipo **MIMD** podem executar diferentes instruções sobre diferentes conjuntos de dados, de forma síncrona ou assíncrona (dependendo da utilização de *locksteps*); este é o tipo de computador paralelo mais comum, correspondendo por exemplo à maior parte dos supercomputadores atuais e aos sistemas *cluster* ou *grid*.

Um outro ponto fundamental acerca da arquitetura de computadores paralelos prende-se com a questão da memória. No essencial, são admitidas duas estruturas: memória partilhada e memória distribuída. Nas máquinas com **memória partilhada** (figura 2.1), todos os processadores podem aceder a todo o espaço de memória, pelo que as alterações efetuadas por um processador são visíveis para todos os outros; cada processador opera de forma independente dos outros, mas os recursos de memória são partilhados entre si. Esta é uma boa forma de partilhar informação entre processadores, que elimina a necessidade de uma rede de comunicação entre processos; a desvantagem é que, por um lado, é muito pouco escalável - a introdução de mais unidades de CPU vai congestionar o tráfego no sistema CPU-memória, atrasando a execução; e, por outro, é mais vulnerável a erros de alocação de memória, dado que sobrecarrega o programador com a responsabilidade de coordenar todos os processos de forma a garantir que não existam acessos concorrentes simultâneos.

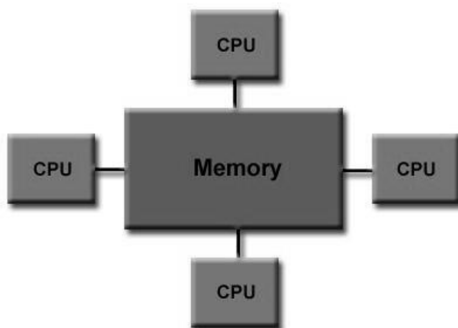


Figura 2.1: Memória partilhada.

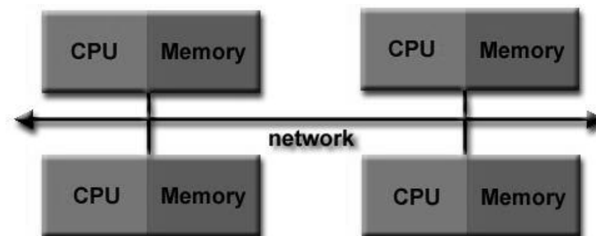


Figura 2.2: Memória distribuída.

Em sistemas de **memória distribuída**, cada processador tem a sua própria memória, pelo que opera de forma verdadeiramente independente - as alterações que faz não são reproduzidas nos outros processadores. Desta forma, é necessário manter uma rede de comunicação entre processadores; sempre que um processador precisa de informação contida noutro, é necessário usar a rede para que troquem uma mensagem entre si. É da competência do programador formular explicitamente esse pedido e garantir a sincronização entre os dois processos (isto é, garantir que a informação já está disponível num processador quando deve ser enviada para o outro). A principal vantagem deste modelo reside

[4]Do inglês *Single Instruction, Single Data*; as restantes siglas seguem uma terminação análoga.
[5]No sentido de *fluxo*, ou seja, recebe uma ou mais instruções de apenas uma fonte.

precisamente na questão da escalabilidade: ao introduzir mais CPUs, o espaço de memória aumenta proporcionalmente, pelo que é possível aumentar a capacidade de processamento sem comprometer a memória; adicionalmente, neste formato é também possível ter um CPU a trabalhar na sua própria memória, sem necessidade de sincronização permanente com os restantes, evitando assim *overheads* e outras interferências. Pela negativa, o programador tem a responsabilidade de coordenar toda a rede de comunicação, e o tempo de execução está sujeito a maiores flutuações, visto que o tempo de acesso através da rede a um determinado conjunto de dados depende da sua localização física.

Em geral, sistemas de memória distribuída adequam-se melhor a uma situação em que os processadores possam trabalhar independentemente e apenas necessitem de comunicar, com pouca frequência, os seus resultados entre si; se, por outro lado, houver necessidade de permanente sincronização entre todos os processos, então os sistemas de memória partilhada são uma melhor opção. Evidentemente, pode ser encontrado um bom equilíbrio usando ambas as arquiteturas simultaneamente; basta considerar um sistema de memória distribuída em que cada nó, em vez de ser um único processador, é uma máquina com vários processadores com memória partilhada entre si. Estes **sistemas híbridos** permitem aproveitar as vantagens dos dois modelos e ao mesmo tempo aumentar a escalabilidade do sistema, com o óbvio custo de uma programação mais complexa.

2.2.3 Limites da paralelização

Para ser possível avaliar o benefício da paralelização é necessário compreender em que aspetos ela pode ser útil, perceber quanto se pode ganhar em cada um desses aspetos e definir métricas apropriadas para medir esses ganhos.

Na secção 4.1 estabeleceu-se a minimização do tempo de execução de um programa como um dos objetivos fundamentais da paralelização. Tornam-se portanto pertinentes as questões sobre qual o grau expectável dessa minimização e como poderá ela ser quantificada. É conveniente introduzir o conceito de ***speedup*** (S_N), que se define como o fator pelo qual o tempo de execução é alterado mediante a introdução de paralelização ([3]):

$$S_N = \frac{t_1}{t_N} \tag{2.1}$$

em que t_1 é o tempo de execução para um processador [6] e t_N a quantidade análoga para N processadores.

A **lei de Amdahl** ([4]) estabelece que o *speedup* potencial de um programa é definido pela fração de código que pode ser paralelizada (P)[7]:

$$S_{max} = \frac{1}{1-P}$$

Como corolário desta definição, é evidente que se nenhuma parte do código for paralelizável então $S_{max} = 1$, ou seja, não há *speedup*; e que se o código for inteiramente paralelizável, então o *speedup* é, em teoria, infinito. Dela se conclui também que é fundamental que o código seja altamente paralelizável, uma vez que S_{max} cresce exponencialmente com P, tal como mostra a tabela 2.4.

P	.5	.8	.9	.95	.99	.999	.9999
S_{max}	2	5	10	20	100	1000	10000

Tabela 2.4: *Speedup* em função de P.

Tomando em consideração os N processadores utilizados e assumindo uma máquina de dois estados - isto é, que em cada momento ou apenas um processador está a ser usado, ou então todos os N processadores estão - resulta que

$$S_{max} = \frac{1}{\frac{P}{N} + (1-P)}$$

[6]A definição usual é que t_1 corresponde ao *melhor* tempo de execução possível para um processador; por conveniência, é comum utilizar-se o tempo de execução da aplicação paralela correndo num único processador, o que, não sendo em rigor correto, introduz apenas um erro residual.

[7]Por coerência, definimos o tempo total t como a soma do tempo despendido na porção P de tarefas paralelizáveis, p, com o tempo dispendido na porção $(1-P)$ de tarefas não-paralelizáveis, s; $t = s + p$.

É imediato observar que a primeira parte do denominador corresponde à porção de código que é paralelizável e a segunda àquela que é estritamente sequencial. Tal como seria de esperar, o *speedup* aumenta com o número de processadores utilizados, até ao limite $\frac{1}{(1-P)}$, como mostram as tabelas 2.5:

N	S_{max}
10	1.818
100	1.980
1000	1.998
10000	1.999

(a) P=.5

N	S_{max}
10	5.263
100	9.174
1000	9.911
10000	9.991

(b) P=.9

N	S_{max}
10	9.174
100	50.251
1000	90.992
10000	99.020

(c) P=.99

Tabela 2.5: *Speedup* em função de N.

Esta formulação ignora ainda o contributo do *overhead* de paralelização, que pode ser significativo e em geral cresce com N; introduzindo o termo $\sigma(N)$ para designar este *overhead*, vem finalmente a expressão geral para S_{max}

$$S_{max} = \frac{1}{\frac{P}{N} + (1-P) + \sigma(N)} \tag{2.2}$$

que, no caso $P = 1$ (paralelização total), se reduz a $S_{max} = \frac{N}{1+N\sigma(N)}$. Com $N \to \infty$, $S_{max} \to \frac{1}{\sigma(N)}$, que é uma função decrescente; isto significa que existe um N ótimo para o qual o *speedup* é maximizado[8].

É importante frisar que a principal limitação ao paralelismo vem da própria estrutura do programa, e não do número de processadores utilizados. Considere-se o caso de se ter um programa com tempo de execução de 1 dia, que se pretende reduzir para 1 hora; o *speedup* desejado é então $S = 24$. A tabela **??** mostra o número de processadores necessários para obter $S = 24$, em função do grau de paralelização do programa; a diferença entre paralelizar o programa a 96% ou 97%, aparentemente pouco significativa, é a diferença entre precisar de 576 processadores - impossível, por exemplo, no *medusa* - ou 84 - um número razoável na maior parte das infraestruturas.

P	.96	.965	.97	.975	.98	.985	.99
N	576	145	84	59	46	37	32

Tabela 2.6: N em função de P para $S = 24$.

Os resultados da lei de Amdahl baseiam-se na assumpção de que o tempo gasto por um processador em tarefas paralelizáveis, p, é independente de N. Na verdade, essa abordagem *fixed-size* raramente é aplicada fora do campo da investigação académica; na prática, o tamanho do problema escala com o número de processadores, ou seja, p depende de N (e, em geral, aumenta)[9], uma vez que as componentes série do programa, como inicializações de vetores e I/O, não variam com N. Esta observação limita a aplicabilidade da lei de Amdahl e está na origem da chamada lei de Gustafson [5]. Definindo p e s como os tempos paralelo e sequencial dispendidos no sistema paralelo, então, para um processador série, $t_1 = s + p \times N$, donde

$$\begin{aligned} S_s &= t_1/t_N \\ &= (s + p \times N)/(s + p) \\ &= s + p \times N \\ &= N + (1 - N) \times s \end{aligned}$$

Este *speedup escalado* S_s[10] tem um comportamento linear em N, que contrasta com o comportamento exponencial previsto pela lei de Amdahl; esta abordagem de tamanho escalável justifica que, na

[8]Este resultado mantém-se obviamente válido para $P < 1$.

[9]Frequentemente parâmetros como resolução da rede, número de e*timesteps*, etc, são ajustados para que o programa corra dentro de uma janela temporal aceitável; ao aumentar a capacidade de computação disponível, tende-se a reajustar esses parâmetros de forma a obter melhores resultados dentro da mesma janela temporal. É a isto que se refere a expressão “o tamanho do problema escala com o número de processadores”.

[10]Do original *scaled speedup*.

prática, conseguir um *speedup* elevado não seja tão complicado como a lei de Amdahl sugere, desde que se escale o tamanho do problema para acompanhar o número de processadores[11].

Um indicador complementar ao *speedup* é a **eficiência**, definida como

$$E_N = \frac{S_N}{N} = \frac{T_1}{N\,T_N}$$

A eficiência toma valores no intervalo $[0, 1]$ e é frequentemente utilizada para perceber se os processadores estão a ser bem utilizados, ou seja, se estão dedicados a tarefas de computação úteis ($E_N \approx 1$), ou se estão a ser *desperdiçados* em processos de comunicação e sincronização ($E_N \approx 0$). No caso ideal de perfeita escalabilidade (e portanto *speedup* linear), $E_N = 1$; em geral, a eficiência decresce com N, refletindo o aumento dos *overheads* de comunicação. Esta medida é uma forma fácil de perceber se a paralelização está a funcionar bem ou se, pelo contrário, os recursos estão a ser desperdiçados.

Apesar de largamente utilizados, ambos estes indicadores, *speedup* e eficiência, apresentam algumas lacunas que justificam a introdução de uma nova métrica:

$$e = \frac{\frac{1}{S_N} - \frac{1}{N}}{1 - \frac{1}{N}} \tag{2.3}$$

Esta **métrica de Karp-Flatt**[12] ([6]) representa a fração série do programa, que em condições ideais se anula (para *speedup* linear, $S_N = N$). Como e deve manter-se constante para N crescente, é conveniente utilizá-la para detetar e compreender pequenas flutuações nos outros dois indicadores, nomeadamente relacionadas com:

- Distribuição da carga computacional - assume-se uma distribuição equilibrada para todos os nós, o que não é necessariamente verdade; uma distribuição menos equilibrada traduz-se num aumento do valor de e.
- *Overheads* - um aumento dos *overheads* resulta na redução do *speedup* , logo também num aumento de e com N; um crescimento regular de e é um possível indicador de que a granularidade da parelelização é demasiado fina.

Resumindo, existem vários indicadores para aferir o desempenho de um programa paralelo, cada qual com as suas vantagens e inconvenientes. O *speedup* introduzido na equação 2.1 é uma medida aceitável e largamente utilizada do grau de paralelização do programa, que deve ser calculado e comparado com S_{max} da equação 2.2 para compreender se a paralelização está a ser feita da melhor forma, caso estejamos a tratar um problema de tamanho fixo; caso contrário, devemos recorrer a S_s. A eficiência E_N é útil para perceber se o *hardware* está a ser utilizado convenientemente, ao passo que a métrica de *Karp-Flatt*, e, pode ser usada para um diagnóstico mais detalhado do programa, nomeadamente para identificar problemas de distribuição de carga computacional ou de excessivos *overheads*.

2.2.4 Modelos de paralelização

Vários modelos diferentes permitem concretizar a paralelização do programa, que no essencial diferem entre si no que diz respeito à utilização da memória (partilhada ou distribuída), ao modo de comunicação entre processos e ao tratamento do $input$ de dados. É conveniente introduzir alguns desses modelos[13] e especificar as suas características, ainda que de forma simplificada, de modo a facilitar a escolha do modelo mais adequado em cada situação. Note-se que o modelo de paralelização é independente da arquitetura da máquina - por exemplo, pode ser aplicado um modelo de memória partilhada a uma arquitetura de memória distribuída ([7]).

[11] O tamanho do problema não é necessariamente constante. Tipicamente esta grandeza é controlada por parâmetros ajustáveis pelo utilizador - a resolução da malha, por exemplo, ou o número de iterações, na resolução numérica de equações. Perante uma maior capacidade de processamento, estes parâmetros são em geral ajustados para obter melhores resultados, de tal forma que em geral o que se pretende constante é o tempo de execução - dentro de algum limite razoável - e não a dimensão do problema. Ver [5].

[12] Descrita aqui para problemas de tamanho fixo. No caso de um problema de tamanho escalável, nos moldes considerados na lei de Gustafson, pode ser generalizada introduzindo um fator de escala adicional

[13] Utiliza-se uma combinação livre do idioma original e do português nos nomes dos modelos de forma a minimizar traduções forçadas e perdas de significância.

- Memória partilhada (sem *Threads*)

Este é provavelmente o modelo mais simples de paralelização, assente no mesmo paradigma que a arquitetura de memória partilhada: os processadores partilham um mesmo espaço de memória, no/do qual escrevem/leem assincronamente. Tem a vantagem de não precisar de comunicações, logo permitir uma escrita simples do programa, mas implica a coordenação dos vários processos para impedir escritas concorrentes e conflitos de dados. Em geral é indesejável para programas com grande volume de computação, devido à necessidade de sincronização entre os processos.

- *Threads*

No modelo de *threads* um processo principal é subdividido em n sub-processos (as *threads*) concorrentes entre si. Estas *threads* operam independentemente umas das outras e partilham os recursos do processo original, pelo que comunicam entre si pela atualização da memória, sem recurso à troca de mensagens, tal como acontece no modelo de memória partilhada. Da mesma forma, exigem sincronização para evitar a escrita simultânea num mesmo endereço de memória. Este modelo pode ser visto como uma versão localizada do modelo de memória partilhada, confinado ao escopo do processo principal, sendo principalmente utilizado para evitar replicar a informação do processo original e para introduzir paralelismo localmente, sem necessidade de paralelizar todo o programa.

- *Message Passing*

Neste modelo cada processo executa as suas tarefas na sua própria memória local, não partilhando espaço de memória com outros processos (mesmo que os processos partilhem a mesma máquina física, cada um tem o seu espaço de memória). A comunicação é feita por troca de mensagens entre os processos, através de uma rede de comunicações, e geralmente envolve uma ação cooperativa, isto é, a uma mensagem de *send* tem de corresponder uma de *receive* do processo adequado. Esta troca de mensagens é feita usualmente recorrendo a uma biblioteca de sub-rotinas, que pode variar dependendo da implementação usada.

- *Data Parallel*

Este modelo pressupõe um tratamento global do espaço de memória. Cada processo executa um mesmo conjunto de operações sobre diferentes porções de uma mesma estrutura de dados - por exemplo, num *array* de 100 unidades, usando 4 processos, o processo 1 trabalha sobre as unidades 1 a 25, o processo 2 sobre as unidades 26 a 50, o processo 3 sobre as unidades 51 a 75 e o processo 4 sobre as restantes. A sua implementação pode depender da arquitetura: se for de memória partilhada todos os processos têm automaticamente acesso à sua porção de dados, mas se for de memória distribuída é necessário definir *a priori* a subdivisão e atribuição de cada bloco de dados ao processo respetivo.

- Modelos Híbridos

É rara a utilização estrita de apenas um destes modelos, sendo mais comuns soluções híbridas em que vários modelos se combinam. Um exemplo de modelo híbrido comum reside na combinação dos modelos de *message passing* e *threads*: a computação intensiva é desenvolvida pelas *threads* localmente, em cada nó, e a comunicação entre processos diferentes é feita através da rede de comunicações quando é preciso trocar dados entre processos. Esta forma de paralelismo adequa-se especialmente às arquiteturas de *cluster*.

- Modelos de alto nível

Em geral a programação paralela é feita recorrendo a um modelo de alto nível construído com base numa combinação dos modelos-base apresentados. Distingue-se habitualmente entre modelos *SPMD* e *MPMD*[14], sendo o modelo *SPMD* o mais comum. Este modelo consiste em ter vários processos a executar um mesmo programa, sobre conjuntos que podem ser distintos; este programa é composto por uma combinação de *threads*, *message passing*, *data parallel* ou híbridos, e em geral não exige a execução da totalidade do programa por todos os processadores - admitindo, por exemplo, distribuições ramificadas ou condicionais da carga computacional. O modelo *MPMD* apenas difere do *SPMD* na medida em que cada processo pode executar um programa distinto, o que, não sendo frequentemente necessário, pode por vezes constituir uma solução relevante.

[14] Do original *Single* e *Multiple Program Multiple Data.*

2.3 Estado da arte

Em 1967 já G. Amdahl afirmava que "Há mais de uma década que existem profetas a afirmar que a organização de um computador singular atingiu o limite e que apenas a ligação de uma multiplicidade de computadores de maneira tal que permita uma solução cooperativa pode dar origem a avanços significativos" ([4, p.1]).

Apesar de existir interesse em computação paralela desde meados da década de 1950, apenas nos anos 60 e 70 esse interesse se materializou sob a forma de supercomputadores, assentes num modelo de memória partilhada. Esse interesse fomentou uma evolução contínua, que se refletiu no aparecimento de *Massively Parallel Processors* (*MPPs*) em meados de 80, nos *clusters* de máquinas independentes desde o início dos anos 90 e, mais recentemente, na banalização de máquinas *multi-core* que incluem vários processadores.

Actualmente, a profecia a que Amdahl fazia referência parece estar a concretizar-se, alimentada pela progressiva constatação de que o aumento da capacidade computacional pela maximização do *clock speed* está a atingir o seu limite ([8]) e que se torna mais eficiente, em termos económicos e energéticos, promover esse aumento através da agregação de vários *cores* ([9]); paralelamente, a contínua verificação da lei de Moore tem significado um crescimento exponencial da quantidade de *hardware* disponível, permitindo construir plataformas *multi-core* a um custo cada vez mais reduzido. Estes fatores têm contribuído para acelerar a transição da computação sequencial para a paralela.

Neste momento a computação paralela é largamente usada em vários setores - o da indústria é o mais dominante, usando 50% dos recursos existentes, seguido da investigação científica - e num conjunto variado de aplicações tão distintas como finança, medicina ou gestão logística ([7]). Apesar deste crescimento, a paralelização é ainda um processo pouco automatizado, exigindo uma abordagem muito *manual* e específica para cada problema e consequentemente aumentando a responsabilidade do programador.

As ferramentas disponíveis para paralelização são sobretudo Interfaces de Programação de Aplicativos (*APIs*) ou bibliotecas que trabalham sobre linguagens de programação de baixo nível (tipicamente *C/C++* e *Fortran*). As linguagens de programação exclusivamente paralelas existentes são em geral bastante específicas e de baixo nível; a inexistência de uma linguagem paralela mais global e de alto nível é vista frequentemente como uma das causas para a lenta transição para o paralelismo, e constitui um importante problema em aberto neste campo ([10]). Existem múltiplas ferramentas para desenvolvimento de programas em paralelo, das quais é conveniente destacar, pela sua elevada utilização, o **MPI**, para o modelo de *message-passing*, e o **OpenMP**, usado com o modelo de *threads*; ambos são utilizados em conjunto com uma linguagem usual, como *C++* ou $Fortran$.

Ainda que tenha vindo a verificar uma utilização crescente, a computação paralela é ainda vista largamente como uma tendência e não ainda como a abordagem padrão. Num estudo recente encomendado pela Intel sobre hábitos de programadores e gestores de *software*, apenas 26% classificam a utilização de paralelismo no seu trabalho como crítica - 56% veem-na como importante mas não essencial, e os restantes como irrelevante ([11]). O mesmo estudo mostra que as ferramentas auxiliares de deteção de defeitos de *threading* ou de memória são ainda pouco utilizadas, ilustrando a imaturidade do desenvolvimento de *software* paralelo e a necessidade de melhores ferramentas de paralelização.

Em resumo, tem sido verificado um aumento crescente da utilização de computação paralela nos anos recentes, potenciado pela noção de que o *hardware* está a atingir o limite do seu desenvolvimento; este crescimento não se restringe aos tradicionais setores académico e de investigação, antes é transversal a todos os setores de atividade e com especial impacto na indústria. Sendo um desenvolvimento recente, é ainda uma área imatura e não considerada essencial, existindo várias ferramentas de desenvolvimento - em geral não-*standard* e pouco *user-friendly* - mas poucas complementares, como *debuggers*. O paradigma atualmente dominante é o de *message-passing*, sendo o MPI a plataforma mais utilizada.

2.4 MPI

O desenvolvimento da paralelização descrita no capítulo 4 foi feito com base no modelo de *message-passing*, usando a implementação *Open MPI* do MPI. Esta secção pretende apenas introduzir o MPI e os seus conceitos essenciais; para detalhes sobre a sua utilização, em particular sobre a sintaxe das rotinas, é conveniente consultar a documentação de *Open MPI*, disponível em [12][15].

[15]Para uma descrição mais completa ver [13].

O MPI[16], cuja primeira versão data de 1992, é uma especificação de um conjunto de rotinas (biblioteca) usadas para a comunicação entre processos paralelos[17]. Tem como objetivo estabelecer um padrão para a escrita de programas paralelos no modelo de *message-passing*, de forma a maximizar a sua portabilidade, praticabilidade, flexibilidade e eficiência. A versão mais recente de MPI é o MPI-3.

Na prática, as razões por optar por MPI prendem-se com o facto de ser a única especificação que pode ser considerada *standard*, de praticamente não ser necessário alterar o código aquando de uma mudança de plataforma e de estar facilmente acessível, incluindo implementações *open-source* como o *Open MPI*. Pode ser utilizado sobre as linguagens C, *C++* ou $Fortran$, com diferenças mínimas ao nível da escrita[18].

A estrutura geral de um programa MPI é a seguinte:

- incluir o *header* **mpif.h** e/ou outras bibliotecas
- declarar variáveis, protótipos, etc (tal como num programa série)
- inicializar o MPI (com a função MPI_Init)
- processar a computação
- finalizar o MPI (com a função MPI_Finalize)

Não é recomendável a inclusão de código antes da inicialização de MPI (exceto declarações) nem após a sua finalização, caso contrário o comportamento é imprevisível.

Em MPI, a estrutura de **grupo** inclui um ou mais processadores que se pretende que comuniquem entre si. Cada processador está associado a um ou mais grupos. Cada grupo tem associado a si um **comunicador**, mais especificamente um intra-comunicador, que permite a troca de mensagens entre os processos desse grupo. O grupo existente por defeito, a partir do momento em que o MPI é lançado, é universal e o seu comunicador **MPI_COMM_WORLD** engloba todos os processos. Cada comunicador tem um tamanho - o número de processos que abrange - que vulgarmente se designa por **nproc**. Cada processo tem um número inteiro único que o identifica no contexto do seu comunicador, o rank, contido no intervalo $[0, nproc - 1]$.

A comunicação em MPI é feita por troca de mensagens entre os processos, que pode ser feita ponto-a-ponto ou coletivamente. As comunicações **ponto-a-ponto** envolvem a troca de uma mensagem apenas entre dois processos específicos; tipicamente o processo A chama um ***send*** com a mensagem pretendida, que só é recebido pelo processo B se este chamar um ***receive*** correspondente.

Em termos de aplicação, existem várias variações de *sends* e *receives*. A distinção mais importante reside no facto de serem ***blocking*** ou ***non-blocking***: uma operação *blocking* deixa o processo em espera até receber a mensagem correspondente, enquanto que uma *non-blocking* prossegue a execução. A primeira opção é mais segura e fácil de prever, mas mais demorada e pode dar origem a ***deadlocks***, isto é, situações em que dois processos esperam um pelo outro e nenhum avança, bloqueando o programa. Um exemplo típico de *deadlock* é o caso em que os processos A e B chamam simultaneamente um *send* para o outro; como cada processo fica à espera do *receive* do outro, que não é chamado por nenhum porque o *send* ainda não foi concluído, ambos os processos ficam bloqueados. A segunda opção é mais rápida mas mais difícil de controlar, uma vez que pode originar problemas de sobreposição de mensagens ou de correspondência errada, devido à execução concorrente dos processos (secção 2.2.1). Este tipo de comunicação é útil quando é apenas necessário trocar informação com um único processador sem envolver os restantes, por exemplo num problema do tipo “linha de montagem” em que cada processo recebe uma informação, executa uma certa operação e envia o resultado para o processo seguinte.

Mais comuns são as comunicações **coletivas**, em que a troca de informação envolve todos os processos do comunicador. Estas comunicações também podem ser *blocking* ou *non-blocking*. Relativamente às comunicações ponto-a-ponto, estas são mais fáceis de escrever - por exemplo para enviar uma mensagem para N processadores basta fazer uma chamada a uma função coletiva, em vez de N *sends* - mas exigem uma maior atenção do programador para garantir que todos os processos são envolvidos na comunicação (mesmo os que não tenham necessidade disso). Distinguem-se neste campo três tipos de operações: sincronização, troca de dados e computação coletiva.

[16]Do original *Message Passing Interface*.
[17]Não é, em si, uma biblioteca - esta resulta da implementação, por exemplo *Open MPI*.
[18]Neste trabalho as referências dizem respeito à sintaxe de $Fortran$ a menos que o contrário seja especificado.

- Sincronização - frequentemente é necessário forçar que todos os processos esperem por um determinado evento, isto é, sincronizar os processos. Isso pode ser feito através de uma ***barreira*** coletiva (função *MPI_Barrier*) - cada processo espera até que todos tenham chamado a mesma barreira antes de prosseguir.

- Troca de dados - permite trocar informação entre mais do que dois processos. As principais operações são de ***broadcast*** - envio de uma mensagem de um processador para todos -, de ***scatter*** - envia diferentes mensagens com origem num único processador para todos os processadores ("dispersão") - e de ***gather*** - recebe informação de vários processadores num único, de forma inversa ao *scatter* ("recolha").

- Computação coletiva - permite executar uma operação sobre dados residentes em todos os processadores. Tipicamente corresponde a uma operação de ***reduce***, que junta a informação de todos os processadores, trata-a e devolve o resultado para um único processador. Além das operações mais comuns definidas pelo MPI (soma, multiplicação, máximo) é possível ao utilizador definir a sua própria função de *reduce*.

Estes são os conceitos fundamentais do funcionamento do MPI; não se pretende fazer uma descrição exaustiva das capacidades do MPI mas apenas permitir a compreensão do seu modo de aplicação genérico. Para uma exploração mais detalhada das funcionalidades do MPI recomenda-se a consulta das referências [14] e [15].

2.5 Estratégia de paralelização

É conveniente reunir os conceitos já abordados numa única estratégia de paralelização, tão genérica quanto possível, que possa ser aplicada a uma variedade de problemas. É esse o objetivo desta secção, cujo conteúdo tem como origem a experiência de paralelização de alguns problemas como o descrito no capítulo 4. Apesar de tão geral quanto possível, esta estratégia foi pensada no contexto do modelo de *message-passing* com utilização do MPI, pelo que podem existir noções que não se apliquem a todos os modelos ou plataformas.

Distinguimos três momentos do processo de paralelização: a abordagem inicial, a paralelização propriamente dita e a análise dos resultados; esta última não é contemplada nesta secção, sendo apenas abordada no contexto do estudo de caso do capítulo 4 para melhor compreensão. Acrescenta-se ainda uma secção de "dicas" destinadas a chamar a atenção para alguns detalhes que podem ser úteis na resolução de determinados problemas. É ignorada a hipótese de recorrer a paralelização automática.

2.5.1 Abordagem inicial

Normalmente pretende-se introduzir paralelização para melhorar um programa já existente. A tarefa prioritária é **compreender qual o problema que o programa trata e como faz para o resolver**, para facilitar a sua posterior modificação para incluir paralelismo. Simultaneamente, a motivação para a paralelização deve ficar bem definida - é importante que seja claro se o programa demora demasiado tempo a executar, e portanto o objetivo é reduzir esse tempo, ou se é incapaz de processar os dados que devia, e então o objetivo é repartir a utilização de memória. Em cada caso devem ser estabelecidos objetivos concretos: caso o problema seja o tempo, definir o limite superior ao tempo desejado; caso seja a memória, definir a dimensão do problema que deve ser possível processar.

O passo seguinte é perceber se o programa é paralelizável. Não é de forma alguma trivial que todos os programas o sejam - por exemplo, um programa que calcule os termos da sequência de Fibonacci pelo algoritmo usual $F(n) = F(n-1)+F(n-2)$ não é paralelizável, porque existe uma dependência entre os termos. Nesta fase o programador deve perceber se existem partes do programa que possam ser paralelizadas e identificar elementos **inibidores de paralelismo**, tais como a dependência entre dados do exemplo anterior. As porções de computação que não sejam identificadas como não-paralelizáveis constituem os potenciais candidatos a serem paralelizados.

Conhecendo já o programa a este nível, torna-se essencial maximizar o proveito da paralelização, respondendo à pergunta "Onde gasta o programa mais tempo?"; o programador deve identificar onde é feita a maior parte da computação, e marcar esses ***hotspots*** como os mais fortes candidatos a paralelizar para obter o máximo benefício. Apenas as porções de código onde seja gasto um tempo

relevante devem ser consideradas para paralelização, uma vez que paralelizar secções com pouco impacto no tempo total de execução terá impacto reduzido no resultado final (pode, inclusivamente, aumentá-lo, devido aos *overheads*).

Para a determinação dos *hotspots* é muito útil o recurso a ferramentas de ***profiling***, das quais a mais acessível é o *gprof*[19]. Para analisar um programa série com esta ferramenta, o programador deve:

1. Acrescentar a *flag* “-pg” à compilação de *nome_programa*
2. Executar o programa normalmente (é criado o ficheiro “gmon.out”)
3. Correr o comando “gprof *nome_programa*>out”
4. Analisar ficheiro “out” que contém a informação de *profiling*

Para um programa paralelo este tipo de análise é mais complicado, dado que o *gprof* não funciona de maneira regular. Existem ferramentas para *profiling* de aplicações paralelas, mas que não foram abordadas neste trabalho; algumas orientações sobre esta questão podem ser encontradas em [16]. Resumindo, a abordagem inicial deve responder a três questões:

1. Como funciona o programa?
2. É paralelizável?
3. Onde é mais rentável paralelizar?

    onde se gasta mais tempo?

    que estruturas ocupam mais memória?

2.5.2 Paralelização

A partir do momento em que se conclui pela possibilidade da paralelização e se identificam os *hotspots* relevantes, estão reunidas as condições para passar à paralelização propriamente dita, que corresponde à fase de computação da estrutura global introduzida em 2.4. Esta fase apresenta em geral uma subestrutura semelhante à seguinte:

1. aquisição do *input*;
2. distribuição da carga computacional;
3. computação independente;
4. comunicação:

    com outros processos, se necessário para prosseguir computação (nesse caso regressar ao ponto 3);

    dos resultados (operação de *reduce*), caso a computação esteja finalizada.

A generalidade dos programas inclui uma parte inicial de processamento de I/O, nomeadamente para *input* dos dados com que vai trabalhar. Na transição para o paralelismo é necessário decidir como tratar este processo. Distinguem-se duas opções mais comuns, ambas constituindo formas de I/O standard (ou seja, sequencial): leitura simultânea em todos os processos (**transversal**) ou leitura no *root* seguida de *broadcast* para os restantes processos.

O primeiro caso é bastante simples de escrever: basta incluir as instruções de leitura tal como num programa sequencial. Cada processador executa essa instrução e lê o ficheiro designado. Esta solução tem a vantagem de ser a mais simples de usar e não contribuir para o *overhead* de comunicação; no entanto, exige a alocação dos *arrays* globais em todos os processos - o que pode não ser viável em termos de memória disponível - e pode dar origem a congestionamentos no sistema, quando múltiplos processos tentam aceder ao mesmo ficheiro, introduzindo assim um *overhead* de I/O indesejado, cuja magnitude depende da quantidade de informação que é necessário ler ou escrever.

[19] O Gprof é um projecto GNU sob licença GNU GPL (*open-source*, disponível por defeito nos sistemas Unix.

No caso leitura em ***root*** + ***broadcast***, evita-se o congestionamento provocado por múltiplas leituras simultâneas e é fácil trocar o *broadcast* por um *scatter* e com isso evitar a necessidade de alocar os *arrays* globais, o que é uma vantagem em termos de memória. A desvantagem desta opção é que introduz um *overhead* de comunicação, que pode ser muito significativo, e que obriga todos os processadores *non-root* a esperar enquanto o *root* faz a leitura e até lhes ser enviada a sua parcela de dados a trabalhar. Ambas as soluções têm aplicação, cabendo ao programador decidir, em função do problema, qual a mais adeaquada.

Existe ainda uma terceira opção, que é usar I/O paralelo, já incluída nas implementações de MPI-2, mas que é ainda algo imatura e tem algumas particularidades - por exemplo, a leitura pode ser feita apenas a partir de ficheiros binários - que complicam a sua utilização.

Concluído o processo de leitura é preciso distribuir a carga computacional pelos processadores. O primeiro passo é distribuir os dados pelos processadores que os vão trabalhar; a maneira mais simples de conseguir isso é definir N blocos de dados e distribuir um por cada processador. Existem várias alternativas para efetuar esta distribuição, por exemplo usando um modelo cíclico ou uma distribuição funcional; o ponto essencial é que essa distribuição seja equilibrada e permita que a cada processador seja atribuída uma carga semelhante, de forma a minimizar o tempo em que alguns processos estão à espera que outros acabem o seu trabalho.

Para problemas com estruturas de dados uniformes este tipo de distribuição estática é geralmente a melhor opção. Pode no entanto acontecer que essas estruturas sejam mais heterogéneas (por exemplo, matrizes esparsas) e que uma distribuição regular origine uma sobrecarga de alguns processos; nesse caso pode ser proveitoso adotar uma estratégia de distribuição dinâmica: o *root* lista todas as tarefas pendentes e recebe de cada processo um pedido de atribuição de tarefa, que quando é concluída dá origem a novo pedido, e assim sucessivamente. Desta forma assegura-se em princípio que todos os processos estão sempre ocupados, até que não haja mais tarefas; o custo é ao nível da programação, sendo necessário definir a tarefa *unitária* e introduzir um algoritmo de gestão das tarefas no processo *root*.

A partir do momento em que todos os processos têm a sua carga de trabalho, começam a trabalhar nela de forma independente uns dos outros, até que seja necessário trocar informação entre eles ou forçar a sincronização de todos os processos. Cabe ao programa decidir o perfil de comunicações - ou seja, a granularidade - do programa. Uma granularidade fina - baixo volume de computação entre comunicações - facilita o balanceamento da carga computacional, mas implica um grande *overhead* de comunicação. Por oposição, uma granularidade grosseira permite minimizar os *overheads* comunicativos mas dificulta a distribuição da carga. O balanço ótimo entre estes extremos depende fortemente do problema, mas em geral é mais indicado optar por uma granularidade mais grosseira, uma vez que os *overheads* de comunicação representam o maior contributo para um tempo de execução excessivo.

No final da computação é necessário reunir toda a informação para gerar o resultado pretendido, o que normalmente corresponde a fazer uma operação de recolha de informação sobre todos os processadores. Esta operação pode ser feita para um *array* global ou apenas local, dependendo do objetivo e dos constrangimentos de memória. Em ambos os casos pode ser feita para um único processador, por exemplo com *reduce*, ou para todos, com *allreduce*. Novamente, a decisão depende fortemente do problema e dos objetivos em questão.

É conveniente ilustrar esta estrutura genérica com um exemplo. Considere-se o exemplo trivial de um programa que calcula o quadrado de cada elemento de um vetor v com n elementos. O código série será algo como:

```
do i=1,n
  v(i) = v(i)*v(i)
end do
```

Vejamos como podem estas orientações ser aplicadas à paralelização deste exemplo[20]. Inicialmente, a leitura de *input* (como exemplo usamos o modelo *root* + *broadcast*):

```
if(rank==root)
  allocate v(n)
```

[20]Para facilitar a compreensão e evitar detalhes desnecessários usamos pseudo-código baseado em $Fortran$.

```
  read v
end if
```

Note-se que apenas é preciso alocar o vetor global no *rank* onde vai ser lido. A distribuição pode então ser feita com um *scatter* em vez de *broadcast*, definindo a dimensão dos vetores locais n_pp e alocando-os:

```
n_pp = n/n_proc
allocate (v_pp(n_pp))
call scatter (n_pp elementos de v para v_pp)
```

Cada processo tem a sua carga e pode começar a computação independente:

```
do i=1,n_pp
  v_pp(i) = v_pp(i)*v_pp(i}
end do
```

Neste exemplo não é preciso combinar os resultados dos diferentes processos (fazendo um *reduce*), mas apenas agregá-los. Reunindo a informação de todos os processos, voltamos a obter o vetor global:

```
call gather (n_pp elementos de v_pp para v no rank root)
```

Apesar de simples, este exemplo da paralelização de um *loop* ilustra a estrutura fundamental de um programa paralelo: leitura de *input*, distribuição da carga computacional, execução da computação em cada processador e reunião final dos resultados.

2.5.3 Dicas

Em conjunto com esta estratégia global, é importante ter presente um conjunto de apontamentos que podem facilitar o processo de paralelização:

- preservar reversibilidade - é frequente cair na tentação de paralelizar uma parte demasiado extensa do código logo à partida, sem nenhum tipo de verificação periódica, e chegar a um ponto em que o código semi-paralelo/semi-série não corre ou não dá os resultados esperados. Nesse sentido, é boa prática preservar a reversibilidade do programa para a versão série durante a paralelização, e remover essa reversibilidade apenas após testar o funcionamento da versão paralela. Na prática, uma maneira de fazer isto é começar por incluir todo o programa num *if(rank==root)*, de forma a preservar a serialidade, e ir progressivamente paralelizando porções do código e retirando-as do *if*. A tabela 2.7 apresenta um exemplo genérico desta solução. Note-se que ela implica que a porção paralelizada (no exemplo, *work_mesh_pp*) devolva os mesmos parâmetros que o original não paralelizado (*work_mesh*, o que implica usar os *arrays* globais e portanto usar mais espaço de memória do que o necessário; porém, essa situação pode ser corrigida à posteriori removendo a operação de *reduce* do *work_mesh_pp*, pelo que esta é uma estratégia útil para a parte inicial da paralelização e que não compromete o resultado final.

(a) Série	(b) Paralelo I	(c) Paralelo II
call create_mesh	if(rank==root)	if(rank==root)
call work_mesh	call create_mesh	call create_mesh
call print_mesh	call work_mesh	end if
	call print_mesh	**call work_mesh_pp**
	end if	if(rank==root)
		call print_mesh
		end if

Tabela 2.7: Preservação da reversibilidade durante a paralelização.

- minimizar comunicação - independentemente do modelo de programação, é sempre conveniente reduzir ao mínimo a comunicação entre processos de forma a evitar *overheads* excessivos que podem comprometer a viabilidade da paralelização.

- usar comunicações *non-blocking* - caso a topologia do programa exija um elevado volume de comunicações, deve ser considerada a hipótese de usar versões *non-blocking* das rotinas de comunicação, de forma a que os processos possam prosseguir em tarefas de computação em vez de ficar à espera da conclusão da comunicação. Esta solução acarreta um esforço acrescido de sincronização dos processos para evitar o *overwrite* de dados.

- minimizar I/O - tipicamente processos de I/O são uma fonte de demora no programa, podendo criar *bottlenecks* caso muitos processos tentem aceder ao mesmo conjunto de dados ao mesmo tempo, pelo que, tal como as comunicações, devem ser minimizados de forma a evitar *overheads*. Adicionalmente, sempre que possível devem ser escritos largos volumes de dados poucas vezes, em vez de pequenos volumes muitas vezes, para maximizar a eficiência.

- considerar outros algoritmos - é natural a tendência para manter o algoritmo série original e introduzir apenas as alterações necessárias para a execução em paralelo, mas esta pode não ser a melhor solução em termos de tempo. Se possível, devem ser considerados algoritmos alternativos que se adequem melhor a um ambiente paralelo (idealmente essa investigação deve ser feita antes de se começar a paralelização, para evitar esforço desnecessário).

- usar bibliotecas - em geral existem várias bibliotecas com rotinas que se adequam ao problema específico a ser tratado. Este *software* deve ser pesquisado e utilizado por forma a acelerar o programa e minimizar trabalho desnecessário. Dois exemplos são as bibliotecas ParMETIS, para partição de malhas não-estruturadas em paralelo, e o ScaLAPACK, uma biblioteca de álgebra linear.

- distribuições de *arrays* não uniformes - é frequente que, ao dividir um *array* global por vários processadores, os *subarrays* fiquem com dimensões diferentes (basta a dimensão do *array* não ser múltipla do número de processadores). São também frequentes operações de *gather* ou *scatter* entre esses *arrays*, e um erro comum é tentar utilizar as funções *MPI_Gather* ou *MPI_Scatter* para isso, o que resulta num erro porque essas funções apenas podem ser usadas para *arrays* de dimensões iguais. As funções adequadas a este caso são *MPI_GatherV* e *MPI_ScatterV*.

- manter nomenclatura inteligente - é relativamente fácil, particularmente numa fase inicial, que a paralelização dê origem a tantos novos vetores e operações que se torne confuso para o programador saber o que é o quê. Para minimizar confusões é conveniente adotar uma nomenclatura inteligente que permita ao programador perceber imediatamente se determinado vetor é local ou global, se tem algum equivalente local/global, etc. Uma sugestão para isso é acrescentar o sufixo $_pp$ aos elementos paralelizados, e manter a nomenclatura original para os série. Uma outra forma é apresentada na secção 4.3 no âmbito do caso de estudo; várias formas são possíveis, desde que cumpram o objetivo prioritário de facilitar a leitura e compreensão do código.

Capítulo 3

Computação Paralela no LNEC

No contexto do LNEC o panorama da computação paralela concentra-se na utilização do *cluster* Medusa. Este capítulo pretende introduzir o Medusa e descrever em traços gerais o seu modo de utilização, de maneira a que o investigador interessado possa encontrar aqui todas as instruções necessárias para executar o seu código - já paralelizado - neste ambiente.

3.1 Medusa

O Medusa é um *cluster* de 67 servidores, cada um com 4 *cores* , num total de **268 *cores*** , com 1GB de RAM por *core* ; dispõe ainda um conjunto adicional de máquinas de *login*. No sentido da classificação introduzida no capítulo 2 é um computador paralelo de arquitectura MIMD, uma vez que cada processador pode estar a executar instruções diferentes em dados diferentes (dependendo do código), com memória distribuída. Dispõe de dois *filesystems*, **NFS** nas *homes* dos utilizadores e o sistema paralelo **Lustre** em $/data/lnec$. Em termos operacionais, os servidores têm instalado *Scientific Linux 5*[1] e a gestão local de recursos é feita usando o sistema *Son of Grid Engine*. O sistema está sob operação técnica do LIP no âmbito da iniciativa INGRID, que disponibiliza a *wiki* indicada em [17].

Como utilizar

O primeiro passo para começar a utilizar o Medusa é solicitar o registo de um novo utilizador LNEC à equipa INGRID, o que pode ser feito enviando um *e-mail* para o endereço de *ingrid.helpdesk@lip.pt*.

O acesso ao Medusa é feito remotamente por *ssh*[2], entrando numa das máquinas de *login* e tendo acesso à sua pasta pessoal (uma subpasta de */home/lnec*). A partir daqui o utilizador pode submeter *jobs* para execução no *cluster*. Toda a computação deve ser efetuada pela submissão de *jobs* - deve evitar-se usar a máquina de *login* para qualquer tipo de computação, incluindo compilações. Os *jobs* submetidos, que vão ser executados nos nós de computação, devem estar sob a forma de *shell script*, como mostra o exemplo seguinte[3]:

```
#--- job script example ---
#!/bin/bash
# Opção para herdar o ambiente do utilizador na execução
#$ -V
# Opção para começar os jobs na directoria de onde o utizador submete.
#$ -cwd
# Invocar o ambiente paralelo denominado mpi, e executar a tarefa com N instâncias
#$ -pe mpi N
# Carregar os modules de interesse
source /etc/profile.d/modules.sh
```

[1] Actualmente, versão x86_64.

[2] No caso de se pretender aceder a partir de um sistema operativo *Windows*, deve ser usado o utilitário equivalente.

[3] Nota: as linhas iniciadas com $\#\$$ são directivas ao sistema de gestão de *jobs*.

```
module load gcc47/gcc-4.7.1

module load gcc47/openmpi-1.6.3

# Executar a aplicação

echo "=== Running ==="

mpiexec -np $NSLOTS interbath_KDT_pp Grid1_BGround1.gr3 grid2.gr3 1

#--- end of job script example ---
```

As três primeiras instruções são auto-explicativas. No lugar de N deve ser inserido o número de processos em que se pretende executar o *job* . É necessário carregar os módulos de interesse, com a instrução ***module load***, que tenham as dependências corretas para o código a submeter. Este sistema de módulos permite a utilização concorrente de várias versões de um mesmo programa sem causar conflitos. Uma lista de todos os módulos disponíveis pode ser obtida correndo o comando *module avail | more*.

Em geral devem ser criados dois *jobs* , um para compilação e outro para execução, para facilitar a identificação de eventuais problemas em alguma dessas fases; estes podem no entanto ser reunidos num único se for mais conveniente. A submissão do *job* tem de ser feita para uma *queue* específica, que pode ser "medusa", exclusiva para utilizadores LNEC e cujos recursos estão ligados através de uma rede Gigabit Ethernet a 1Gbit/s, ou "hpcgrid", disponível para todos os utilizadores e que dispõe, além de uma rede Ethernet, de uma rede de recursos ligados por *infiniband* (8Gbit/s).

Após a submissão e execução do *job* , são gerados quatro ficheiros de *output*, dos quais os mais importantes são os de *standard error* (*.e) e *standard output* (*.o). Este último devolve o que numa execução local seria escrito para o ecrã. Recomenda-se a escrita dos resultados para um ficheiro em separado, por forma a facilitar o seu pós-processamento.

A tabela 3.1 resume os principais comandos relevantes para a operação com o Medusa a partir de sistemas Linux/Unix.

Função	**Comando**	**Exemplo**
login	ssh username@lnec.ncg.ingrid.pt	ssh jpcoelho@lnec.ncg.ingrid.pt
copiar ficheiros	scp *files *dest	scp file_to_copy jpcoelho@lnec.ncg.ingrid.pt:folder
submeter *job*	qsub -q queue filename.sh	qsub -q medusa run.sh
verificar estado do *job*	qstat -q queue	qstat -q medusa
apagar *job*	qdel -q job	qdel -q run.sh

Tabela 3.1: Principais comandos para utilização do Medusa.

Finalmente, convém referir algumas notas adicionais que podem ser relevantes:

- Podem ser aglomeradas no mesmo *job* várias tarefas, como por exemplo compilação, execução e algum pós-processamento. Esta opção é especialmente útil caso se pretendam fazer múltiplas corridas do mesmo *job* .
- Não devem ser copiados para o medusa ficheiros já compilados (executáveis), mas apenas ficheiros de código-fonte. A compilação deve ser sempre feita no Medusa, a partir da *home*.
- Cada *job* pode ser submetido para até 64 processadores. O número ideal de processos a lançar depende do programa em questão, mas geralmente 16 a 20 processos são suficientes.
- A *queue* escolhida para submissão deve ser *medusa*, por ser mais restrita, a menos que se pretenda usar a rede *infiniband*.
- O comando *scp* deve ser sempre corrido a partir da máquina local (PC), e não na remota (Medusa). Caso se pretenda copiar um ficheiro do Medusa para a máquina local, basta inverter a ordem dos argumentos.

Capítulo 4

Estudo de Caso: Interbath

4.1 Definição do problema

A consolidação efetiva dos conhecimentos, técnicas e procedimentos apresentados nos capítulos anteriores depende criticamente da sua aplicação a problemas reais; a resolução de um caso prático surge assim naturalmente como o passo seguinte deste processo de aprendizagem.

Em colaboração com o Dep. Hidráulica, por intermédio do investigador Alberto Azevedo, foi identificado o programa *Interbath* como um bom candidato para paralelização, sobretudo devido à sua simplicidade conceptual e frequente utilização.

A função deste programa é obter uma malha de batimetria regular a partir de uma outra não estruturada, por interpolação, sobre a qual são depois efetuados os cálculos específicos relativos a cada modelo; como tal, esta é uma operação básica fundamental ao estudo de qualquer modelo nesta área, o que justifica a sua vasta utilização e consequente benefício em ser paralelizada. Estas malhas são de elementos finitos, e em ambos os casos os elementos são triangulares. A partir da malha inicial, não estruturada e normalmente resultante de levantamentos no campo, habitualmente designada por malha de *background* (ou malha 1), pretende-se obter por interpolação - que pode ser linear ou não-linear - os valores batimétricos respeitantes aos pontos da segunda malha, estruturada, a que nos referimos como malha de interpolação (ou malha 2).

Em traços gerais, a estrutura do programa é a seguinte:

- Leitura dos argumentos da linha de comando (nome das malhas e tipo de interpolação)
- Leitura das malhas
- Cálculo das coordenadas e da área de cada elemento
- **Criação dos *arrays* de correspondência nó-elemento(*ine*) e e elemento-nó (*nne*)**
- **Criação da árvore para pesquisa**
- Interpolação linear

  Pesquisa na malha 1 pelo elemento em que está contido cada nó na malha 2

  Computação da interpolação com base nos nós do elemento da malha 1
- Interpolação não-linear (apenas se escolhida)

  Novo cálculo das coordenadas e batimetria dos centros de cada elemento da malha 2

  Pesquisa na malha 1 pelo elemento em que está contido o centro de cada elemento da malha 2

  Computação da interpolação com base nos nós do elemento da malha 1

  Distribuição das diferenças pelos nós da malha 2
- Escrita da malha 2 interpolada

Foram disponibilizadas duas versões deste programa: uma mais antiga, interbath.f, escrita em Fortran 77, e outra mais recente, interbath_KDT.f90, escrita em Fortran 90 e incorporando o método de KDTree para acelerar a pesquisa de elementos na malha 1; no esquema anterior, os pontos destacados dizem respeito apenas a esta última versão. Adicionalmente, estão também disponíveis quatro

malhas de *background* e uma de interpolação, utilizadas para testar o programa e analisar o seu comportamento mediante variações na malha de *background*.

Nome	Tipo	Elementos	Nós
Grid1_BGround1.gr3	*Background*	13 316	6 790
Batim.gr3	*Background*	121 351	61 136
Grid1_BGround2.gr3	*Background*	3 408 896	1 706 545
MAlt_BGround.gr3	*Background*	25 814 243	13 667 987
grid2.gr3	*Intpol*	490 641	247 978

Tabela 4.1: Malhas de batimetria utilizadas.

A paralelização deste programa tem dois objetivos principais:

1. Minimizar o tempo de execução do programa
2. Permitir a leitura de malhas de *background* de grandes dimensões

Tal como na generalidade dos casos, o primeiro objetivo da paralelização é distribuir a carga computacional por vários processadores de forma a minimizar o tempo total de execução; de resto, a comparação dos tempos de execução das versões série e paralela será um aspeto central na análise dos resultados. A tabela 4.2 apresenta os tempos de execução das versões série, para as quatro malhas de *background* disponíveis [1].

	Grid1_BGround1.gr3	Batim.gr3	Grid1_BGround2.gr3	MAlt_BGround.gr3
interbath.f	$21.2\,s$	$175.0\,s$	$3973.2\,s$	$19345.2\,s$
interbath_KDT.f90	$12.2\,s$	$15.3\,s$	$32.0\,s$	–

Tabela 4.2: Tempo de execução (*wall-clock*) dos dois programas em função da malha de *backgound* (interpolação linear).

Como mostra a tabela 4.2, a introdução do método de *K-D Tree* tem um impacto significativo sobre o tempo de execução[2]; na verdade, o tempo de execução é relativamente reduzido e dificilmente se pode classificar como um problema real. Ainda assim, numa perspetiva mais *académica*, será mantido o objetivo de o diminuir, tendo porém à partida esta indicação de que será complicado consegui-lo.

O objetivo de alargar o espetro de malhas legíveis pelo programa, por outro lado, é bastante mais pertinente no contexto deste problema. Tal como mostra a tabela 4.2, ambas as versões série são incapazes de processar a malha de $25\,M$ de elementos em tempo útil (ou de todo, no caso da versão KDT, por ser impossível alocar memória suficiente); torna-se por isso necessário introduzir paralelismo para subdividir a malha, e restantes *arrays* de grande dimensão, pelos vários processadores.

4.2 Ferramentas utilizadas

A ferramenta básica a utilizar para a paralelização do *Interbath* é o MPI; existem, no entanto, várias ferramentas adicionais relevantes não só para este caso em particular, como provavelmente também para futuros problemas de paralelização. Nesta secção descrevemos brevemente essas ferramentas, com o objetivo de explicar a sua função, como funcionam e qual a sua relevância para este estudo concreto.

4.2.1 ParMETIS

O ParMETIS ([18]) é uma biblioteca paralela, construída sobre MPI, para particionamento e ordenamento de malhas não-estruturadas; é o equivalente paralelo da biblioteca METIS. Inclui várias rotinas para trabalhar as malhas não-estruturadas, das quais apenas uma será usada neste estudo, a de particionamento de grafos, *ParMETIS_v3_PartGeomKway* [3]. A utilização desta rotina é algo complexa e

[1] Estas medições dizem respeito a corridas no Medusa.

[2] O método de *K-D Tree* encontra-se descrito em detalhe na secção 4.2.2

[3] Existe também uma função para partição directa de malhas, *ParMETIS_v3_PartMeshKway*, que poderia ser usada como alternativa e provavelmente evitaria algum do trabalho de pré-processamento da malha; no entanto, como o funcionamento da rotina de partição de grafos é melhor conhecido, otou-se por manter essa opção.

envolve a criação de vários *arrays* auxiliares, pelo que seria excessivamente complicado implementá-la de raiz; antes, foi usado como referência o código fonte do modelo hidrodinâmico SELFE ([19]), que utiliza o ParMETIS para fazer a partição das malhas que analisa[4].

A relevância desta biblioteca para o trabalho advém precisamente do facto de permitir subdividir, de forma equilibrada, uma malha de grande dimensão por outras de menor dimensão. Este elemento é essencial tanto para conseguir subdividir a carga computacional pelos processadores, no sentido de acelerar a execução, como também permite a repartição de *arrays* demasiado grandes (tipicamente, a malha completa) pelos processadores sem necessidade de alocar previamente o *array* completo.

A utilização do ParMETIS resume-se à chamada da função *ParMETIS_v3_PartGeomKway*. No entanto, a construção dos argumentos recebidos por essa função envolve alguma complexidade, pelo que esse processo foi incluído no módulo de pré-processamento do programa global (*prep_grid.f90*). Essa construção é feita a partir da adaptação das rotinas utilizadas no modelo SELFE que, além de usar o ParMETIS com o mesmo fim, exibe uma correspondência muito aproximada com o *Interbath* em termos das variáveis relevantes (p.ex. número de nós e número de elementos).

É despropositada a descrição detalhada desse processo de construção[5], mas convém compreender em traços gerais o que faz o ParMETIS. Antes da chamada do ParMETIS são criadas e definidas (em *prep_grid.f90*) uma partição inicial dos elementos pelos processadores e as diferentes tabelas de correspondência local-global para os elementos[6]. A partir dessas componentes são definidos os vetores e parâmetros que a rotina *ParMETIS_v3_PartGeomKway* utiliza para criar a partição equilibrada da malha. A chamada a essa rotina devolve um vetor *part* com a informação da partição, ou seja, em que processador está localizado cada elemento na nova partição. Compilando os *parts* de todos os processadores através de uma operação de *gather*, obtém-se finalmente o vetor global de correspondência elemento-a-processador-residente que serve de orientação para o resto do programa[7].

4.2.2 K-D Tree

O método de *k-d tree* foi implementado originalmente na versão *interbath.f90* para acelerar a execução do programa, com os bons resultados que a tabela 4.2 mostra. Uma *k-d tree*[8] é uma estrutura de partição de espaço especialmente útil para pesquisas de elementos vizinhos, como a que é feita na rotina de interpolação.

No essencial, aquilo que o método faz é gerar, para cada nó da malha, uma subdivisão do plano desse ponto ao longo de uma determinada dimensão (eixos XX e YY, no caso bidimensional) de modo a criar dois subplanos; para cada ponto em cada subplano o processo é repetido, alternando a dimensão ao longo da qual é feita a subdivisão; e assim sucessivamente, até cada ponto ficar identificado numa subdivisão única. O resultado é uma reorganização dos nós em árvore que permite percorrer a malha de forma mais orientada, por isso acelerando, neste caso em particular, as pesquisas pelos elementos vizinhos.

O *trade-off* de aplicar este método é o custo de computação adicional proveniente da criação de uma árvore para cada nó; apesar de não ser possível quantificar com precisão esse custo, é expetável que aumente com a dimensão da malha. Por essa razão, em termos de tempo de execução, é de esperar algum *overhead* extra com a introdução do *k-d tree* que deve ser tanto menos relevante para o tempo total quanto maior for a malha a que é aplicado.

É evidente que o bom desempenho do *k-d tree* para a versão série torna desejável a sua transposição para a versão paralela. A sua implementação é feita com base na versão *interbath_KDT.f90*, que usa um módulo já construído para esse efeito; por essa razão não são abordadas questões intrínsecas do método de *k-d tree*, como por exemplo quais os algoritmos ótimos de construção da árvore ou de pesquisa dos elementos vizinhos[9]. Para simplificar a paralelização, numa primeira fase, e para compreender mais em detalhe o seu impacto na *performance* do programa, serão também criadas algumas versões em que este método não é utilizado (ver 4.3).

[4] A documentação sobre este modelo foi disponibilizada pelo investigador Alberto Azevedo, que nele colabora.
[5] Para os detalhes, consultar a documentação do ParMETIS ([20]).
[6] Mais concretamente, a correspondência entre os índices local e global de cada elemento. Na secção 4.3 este processo é explicado com algum detalhe.
[7] Mais concretamente, as tabelas de correspondência são novamente criadas a partir desta "boa" partição, e são essas que são depois utilizadas ao longo do programa propriamente dito.
[8] Abreviatura para *k-dimensional* tree.
[9] Detalhes sobre o método podem ser encontrados em [21]

4.2.3 *Timing*

A medição do tempo de execução do programa é, como foi descrito no capítulo 2, uma componente fundamental do processo de paralelização e o principal critério para analisar o seu sucesso. Foi atrás descrito que existem várias medidas possíveis do tempo de execução, cuja adequabilidade varia consoante aquilo que se pretende estudar; nesse sentido, com o objetivo de criar um método de medição dos tempos rápido e de fácil implementação, foi criado o módulo *timing.f90* com a subrotina *timer*, que calcula:

- tempo de *wall-clock*, usando a função de MPI MPI_Wtime
- tempo total de CPU, usando a função intrínseca de Fortran cpu_time
- tempo máximo de CPU por processador
- tempo médio de CPU por processador
- desvio padrão do tempo de CPU por processador

O cálculo de *wall-clock* e de CPU_time máximo é feito através da redução (chamando MPI_Reduce com a operação MPI_MAX) dos valores em cada processador dessas variáveis. O desvio padrão da amostra de tempos de CPU é calculado como $\sigma = \sqrt{\frac{1}{N}\sum(t_i-\mu)^2}$, com N número de processadores, t_i os tempos de CPU individuais e μ o tempo de CPU médio. A utilização desta subrotina é feita simplesmente através de três chamadas:

- iniciar o contador - com o argumento *start*
- parar o contador - com o argumento *stop*
- imprimir resultados - com o argumento *print*

Foram ainda incluídos dois argumentos adicionais, *date* e *time*, ambos opcionais, que caso estejam presentes armazenam a informação sobre a data e hora de execução do programa. Esta subrotina produz o ficheiro *tracking_filename.txt* com informação sobre todas as variáveis temporais que calcula, de modo a permitir ao utilizador escolher qual a mais apropriada para a sua análise. Note-se que é expetável que este módulo seja aplicável a outros problemas com um mínimo de alterações, uma vez que as variáveis globais de que precisa são apenas as de comunicação usuais, tais como *myrank*, *nproc*, etc.

4.2.4 *Debugging*

É expetável a ocorrência de problemas na compilação ou execução do código; em particular, devido ao paralelismo, é provável a existência de *memory leaks* que inviabilizem a execução do programa. Para fazer o *debugging* destas situações é utilizada, em primeiro lugar, a *flag* de compilação ***-fbounds-check*** (nativa do *gcc*), que identifica e reporta problemas com a utilização de espaço de memória não alocado. Complementar e ocasionalmente, foi utilizado o *debugger* **TotalView**[10], que entre outras funções avançadas permite acompanhar o desenvolvimento do programa passo-a-passo, facilitando a deteção precisa de erros.

4.2.5 Análise

Evidentemente, é necessário analisar os resultados da versão paralelizada e compará-los com a versão série para garantir que a funcionalidade do programa é preservada. Tendo em conta que o resultado deste programa é sempre uma nova malha de grandes dimensões[11], torna-se essencial automatizar esta comparação. Para isso é utilizado o comando *diff*, integrado no ficheiro de compilação e execução do programa, que compara os resultados obtidos com os provenientes da versão série; desta forma, em todas as corridas é feita a comparação de resultados e exibida uma notificação caso estes não coincidam.

[10] Foi usada apenas uma versão de *trial*.

[11] Até agora apenas foi usada uma malha de interpolação, *grid2*, de cerca de 400.000 elementos.

Não sendo essencial, é frequentemente bastante útil visualizar a topologia das malhas (não só as de resultados, como também as de *input*). Para isso são usadas duas ferramentas: o programa *xmgredit5* e o *script python viewGridBat.py*. A sua aplicação difere ligeiramente: o *script*[12] é mais simples e conveniente para gerar a imagem da malha (usualmente, exportando-a como .png), ao passo que o *xmgredit5* visa uma análise mais detalhada da malha, permitindo, por exemplo, a sobreposição de malhas e diferentes modos de visualização. De um modo geral o *script* é usado para obter uma imagem definitiva, enquanto que o *xmgredit* é mais útil para procurar e analisar problemas nas malhas. Ambos são difíceis de executar para malhas de grandes dimensões (da ordem de milhões de elementos); nesses casos, o recurso ao comando $diff$ é a única forma de comparar os resultados.

Uma situação em que a visualização das malhas é especialmente útil é quando se pretende ter uma confirmação de que, após a aplicação do ParMETIS, a malha se encontra bem dividida, isto é, que a união de todas as submalhas coincide com a malha original. De facto não existe nenhuma maneira simples de conseguir isso, e as ferramentas de visualização disponíveis permitem mostrar apenas duas malhas, no caso do *xmgredit*, ou mesmo só uma, no caso do *viewGridBat.py*. Para resolver esse problema foi criado um novo *script* em *python* a partir do *viewGridBat.py*, *viewGridPart.py*, que permite abrir um número arbitrário de malhas (dado como *input*) para visualização simultânea.

4.3 Paralelização

Estão assim reunidas as condições necessárias para descrever o processo de paralelização aplicado ao *Interbath*, que seguiu em larga medida as instruções definidas na secção 2.5. A primeira tarefa foi compreender o funcionamento do programa série, que se descreve em 4.1 para ambas as versões disponíveis, e perceber se poderia ser alvo de paralelização. Como o cerne do programa consiste em repetidas interpolações da malha original, independentes entre si, é imediato verificar que elas podem ser realizadas simultaneamente pelo que é possível introduzir paralelismo.

Em segundo lugar foi essencial definir em concreto os objetivos da paralelização, que neste caso são os dois possíveis: reduzir o tempo de execução e permitir a leitura de malhas de grande dimensão[13]. Concretamente, o pretendido era que a versão paralela fosse capaz de reduzir o tempo de execução para valores abaixo dos indicados na tabela 4.2 e, simultaneamente, processar a malha *MAlt_BGround.gr3*. Ao longo do processo foi-se tornando cada vez mais evidente que seria complicado combinar estes dois objetivos, uma vez que o $KDTree$ já permite uma melhoria tão significativa em termos de tempo de execução que a simples introdução do ParMETIS introduz um *overhead* superior a esse tempo. Por esta razão o objetivo de reduzir o tempo de execução é, como foi dito em 4.1, sobretudo académico, no sentido de não ser previsível que se consiga melhorar significativamente.

Independentemente desta constatação, o processo foi seguido tendo sempre ambos estes objetivos em mente. Para o passo seguinte, de identificação dos *hotspots*, foi utilizada a ferramenta *gprof* que permitiu distinguir a rotina *intpol* como aquela em que o programa despendia mais tempo (cerca de 98%). Uma análise mais detalhada revelou que dentro dessa rotina, a subrotina que mais tempo gastava era a *search_AA*, onde é feita a pesquisa do elemento da malha 2 na malha original de *background*. Através deste processo foi imediato concluir que o foco da paralelização devia ser esta subrotina de pesquisa e, em segundo plano, a própria rotina de interpolação.

Deste modo, finda a abordagem inicial, ficou estabelecido que o programa era paralelizável, que o objetivo da paralelização devia ser tanto de diminuir o tempo de execução como de conseguir ler malhas de grande dimensão, e que o processo de paralelização devia incidir sobre as rotinas de interpolação e pesquisa para que se conseguisse dele tirar o máximo partido.

As várias tentativas de paralelização resultaram em diferentes versões do programa paralelo, que por conveniência se resumem na tabela 4.3 e são adiante explicadas em detalhe.

[12] Da autoria do inv. Alberto Azevedo.
[13] Da ordem de 25 milhões de elementos.

Versão	ParMETIS	KDTree	Obs.
v1	-	grid1	Paralelização simples do ciclo de interpolação.
v2	grid1	-	Tempo de execução excessivo.
v3	grid2	grid1	Funcional mas não lê MAlt_BGround.gr3.
v3.2	grid2	-	Medir overhead de ParMETIS.
v3.3	grid2	-	Medir overhead de ParMETIS (apenas ParMETIS e reads).
v3.3.2	grid2	-	Medir overhead de ParMETIS (apenas ParMETIS).
v4	grid2	grid1	Melhor versão; não lê MAlt_BGround.gr3.

Tabela 4.3: Versões paralelas do programa.

v1

Por simplicidade, a primeira tentativa de paralelização consistiu em tomar a versão série com KDT e apenas paralelizar o ciclo principal da rotina *intpol*. Na fase de *input* adotou-se o método de leitura *root + broadcast*, sem grande preocupação sobre o *overhead* que esse processo introduziria (que em princípio seria não-negligenciável mas se resolveu ignorar dado que esse processo representava menos de 2% do tempo de execução). Foi feita uma distribuição em bloco dos elementos pelos processadores e adaptada a rotina de interpolação para tratar apenas os elementos residentes em cada processador e no final fazer um *reduce* dos resultados. A tabela 4.4 ilustra (em pseudo-código) as diferenças entre as duas versões ao nível da rotina de interpolação.

Versão série

```
do i=1,nodes2
  xx = x2(i)
  yy = y2(i)
  call search_AA(xx,yy, ...)
  if (search is successful) then
    (do computation)
  end if
  h2(i) = (...)
end do
```

Versão paralela

```
allocate ( h2_pp(nodes2_pp) )
do i=1,nodes2_pp
  iproc = nodes2_start + i -1
  xx = x2(iproc)
  yy = y2(iproc)
  call search_AA(xx,yy, ...)
  if (search is successful) then
    (do computation)
  end if
  h2_pp(i) = (...)
end do
call MPI_allgatherv
( nodes2_pp elements of h2_pp in h2 )
```

Tabela 4.4: Paralelismo na v1.

Para analisar os resultados é conveniente analisar, em primeiro lugar, o *Speed-up*, conforme descrito na secção 2.2.3. É também importante considerar as variações observadas em função do número de processadores e da malha de *background* utilizada, pelo que a principal ferramenta de análise, além do *speed-up*, será o gráfico dos tempos de execução em função do número de processadores, para diversas malhas (as mais relevantes em cada caso). A figura 4.1 apresenta os tempos de execução para esta versão.

Apresentam-se apenas os resultados para $np \in [1, 16]$ porque a partir de 16 processadores o gráfico oscila sem nenhuma tendência bem definida, para ambas as malhas[14]. Estes resultados são melhores do que o esperado, tendo em conta a simplicidade da abordagem escolhida, mesmo tendo em conta que esta versão beneficia do contributo do *KD-Tree* para acelerar a pesquisa e não tem ainda de

[14] Todas as malhas foram corridas para np a variar de 1 até 64.

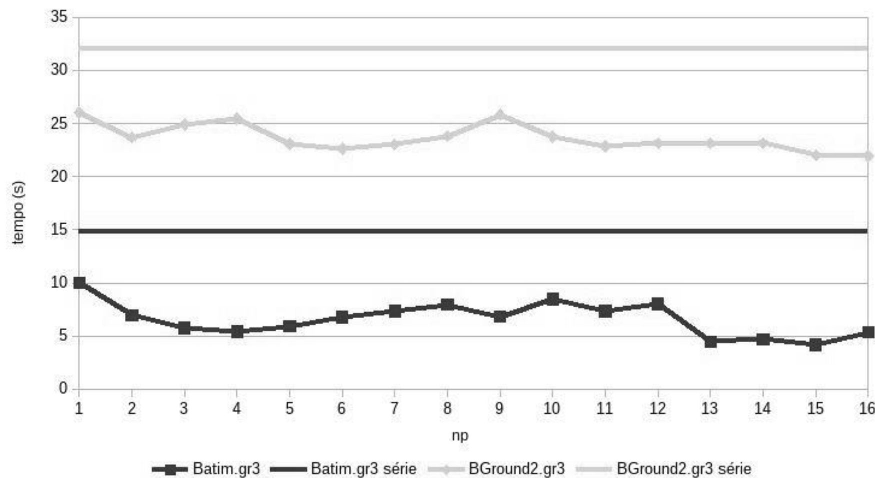


Figura 4.1: Tempos de execução (v1).

suportar o *overhead* do ParMETIS. Consegue-se, simplesmente com a paralelização do ciclo principal da interpolação - isto é, sem introduzir ParMETIS para otimizar a partição da malha - obter *speedups* de 4.13 para a malha Batim (correndo em 43 processadores) e 1.50 para a malha BGround2 (para 23 processadores). No entanto esta versão não consegue correr a malha MAlt_BGround2, pelo que apenas cumpre um dos objetivos - o da diminuição do tempo de execução; além disso, não parece ser uma boa solução para malhas de maiores dimensões do que BGround2, já que a evolução dos *speedups* sugere uma proporcionalidade inversa ao tamanho da malha. Resumindo, esta é uma abordagem conveniente para contextualizar o problema, mas provavelmente ineficiente para o tipo de malhas que se pretendem tratar na realidade, uma vez que não usa ParMETIS para conseguir uma partição equilibrada da malha e exibe uma tendência de diminuição do *speedup* para malhas de maior dimensão, o que limita a sua escalabilidade.

v2

A aparente relação de proporcionalidade inversa entre tamanho da malha de *background* e *speedup* fortalece a hipótese de que a subdivisão dessa malha e consequente pesquisa local para a interpolação acelere a execução. Nesse sentido foi construída uma nova versão em que foi introduzido ParMETIS para tratar da partição da malha de *background*. De forma a poder analisar o efeito desta alteração isoladamente, foi retirada a componente $KDTree$ - na prática, a paralelização foi feita a partir da versão *interbath.f*. Esta versão $v2$ usa então ParMETIS para subdividir a malha 1 e deixa de fora a componente de $KDTree$; o ciclo de interpolação é corrido para todos os nós da malha 2, mas agora a verificação é feita apenas sobre a subdivisão da malha 1 localizada nesse processador.

A transição para a partição efetiva da malha - por oposição à simples repartição da computação feita na v1 - implica a inclusão das rotinas de pré-processamento do ParMETIS no módulo *prep_grid.f90*, tal como descrito em 4.2.1. Nesse módulo estão também definidas as rotinas de criação das tabelas de correspondência local-global entre elementos e nós da malha, cujos resultados são utilizados pelo ParMETIS. Estas rotinas desempenham um papel importante na paralelização (mesmo que o ParMETIS não seja usado, isto é, se se otar por definir uma partição “manual”), inclusive para as versões seguintes, pelo que convém explicar o seu funcionamento com algum detalhe.

Este módulo inclui duas rotinas fundamentais, *aquire_hgrid* e *partition_hgrid*. A primeira trata de adquirir a informação sobre a malha a ser particionada e criar as correspondências local-global, a partir de uma partição inicial. A segunda define inicialmente uma partição grosseira da malha, em seguida chama *aquire_hgrid* para criar as correspondências para essa partição, e depois aplica ParMETIS para criar uma nova partição. No programa, após a chamada a *partition_hgrid* - que termina com a criação da partição equilibrada - é novamente chamada *aquire_hgrid* para repetir o processo de criação das correspondências, desta vez para esta nova partição.

A estrutura de *partition_hgrid* é simples e resume-se à descrição acima; é o funcionamento de *aquire_hgrid* que vale a pena detalhar. Esta rotina começa por adquirir a informação global contida no

ficheiro de *input*: número de elementos, número de nós e **tabela global elemento-nó**[15]. A partir de $nmgb$ constrói a **tabela global nó-elemento**, $inegb$, que tem a estrutura inversa e devolve, para cada nó, todos os elementos que lhe estão associados[16]. De seguida são alocadas e construídas as **tabelas de correspondência global-para-local** de elementos e nós, à custa da informação da partição inicial sobre a localização de cada elemento/nó.

O passo seguinte é construir as listas de elementos *ghost*. Estes elementos são aqueles que, numa divisão rígida dos elementos pelos processadores, ficariam na região de fronteira; nessa situação é provável que elementos contíguos sejam enviados para processadores diferentes, implicando custos de comunicação caso tenham necessidade de interagir entre si. Por essa razão é mais razoável criar um **domínio aumentado**, incluindo elementos comuns a outros processadores nestas *ghost regions*, de forma a minimizar a comunicação no caso de existir necessidade de interação entre elementos[17]. São então criadas as listas de *ghosts* e definidas as dimensões dos domínios aumentados de elementos e nós[18].

A partir daqui as tabelas global-para-local já construídas são reajustadas para passar a incluir os elementos *ghost*. São depois construídas as **tabelas local-para-global** de elementos e nós, para os domínios aumentados. Finalmente, são construídas as tabelas exclusivamente locais, no domínio aumentado, equivalentes às tabelas globais construídas em primeiro lugar, em particular as tabelas elemento-nó (nm, por comparação com $nmgb$) e nó-elemento (ine, por comparação com $inegb$). O último processo da rotina é a leitura das coordenadas do ficheiro de *input* para vetores já alocados localmente, no domínio aumentado.

O conjunto de processos incluídos na rotina *aquire_hgrid* permite criar uma *interface* constituída por tabelas globais, tabelas de correspondência global-local e local-global, e tabelas locais, o que é suficiente para localizar a cada momento qualquer elemento/nó tanto na sua partição local como na malha original completa.

Esta estrutura foi então aplicada ao original *interbath.f* para introduzir ParMETIS na malha de *background* e tentar melhorar a primeira tentativa de paralelização. Os resultados obtidos ficaram aquém do esperado, tendo o tempo de execução (*wall-clock*) sido 10 a 1000 vezes superior ao registado para a versão série, para as malhas *BGround1* e *BGround2*, respetivamente, o que desde logo compromete a escalabilidade do programa. Na verdade este aumento não era totalmente imprevisível, uma vez que era de esperar que a remoção do $KDTree$ tivesse um impacto significativo a este nível; apenas se desconhecia em que dimensão.

Concluindo, não sendo possível desenvolver uma versão funcional a partir desta abordagem, a $v2$ permitiu, por um lado, compreender a importância da inclusão do $KDTree$ no programa para obter um tempo de execução razoável; e, por outro, permitiu ainda aplicar o ParMETIS e testar o funcionamento de todas as rotinas associadas, o que se revelou um contributo valioso para o desenvolvimento das versões seguintes.

v3

Na versão $v2$ foi aplicado o ParMETIS à malha 1, com o objetivo de repartir a pesquisa pelos diferentes processadores. Na verdade, com a aplicação do $KDTree$, essa pesquisa na malha 1 não é problemática, uma vez que não é feita sobre toda a a malha mas apenas sobre um subconjunto relevante; o verdadeiro problema é que essa pesquisa tem de ser feita para todos os elementos da malha de interpolação (malha 2). Nesse sentido, parece mais razoável aplicar ParMETIS não à malha 1, mas à malha 2, para que o ciclo principal de interpolação seja repartido pelos processadores.

Essa premissa foi a base do desenvolvimento da $v3$, partindo da versão com $KDTree$, com o objetivo de usar ambas as ferramentas - $KDTree$ e ParMETIS - para melhorar os resultados da $v1$. O custo desta alteração é a necessidade de alocar os *arrays* relativos à malha 1 em todos os processadores[19], o que significa maior utilização de memória e pode comprometer a escalabilidade do programa.

[15]Respetivamente, ne_global, np_global e $nmgb$; esta última é uma tabela de dimensão $ne_global \times 3$ que para cada elemento devolve os três nós associados.

[16]Na versão completa são ainda criadas as tabelas globais elemento-aresta-elemento; no entanto, como as arestas não são relevantes para este problema (em qualquer versão), e para manter a explicação tão simples quanto possível, omite-se o seu papel (que, de modo geral, acompanha sempre o binómio elemento-nó).

[17]No caso particular do *Interbath*, como faz apenas uma interpolação simples, não é preciso criar *ghosts* porque a computação é independente, mas otou-se por preservar a estrutura mais geral para eventual aplicação futura.

[18]Dimensão aumentada $=$ dimensão local $+$ dimensão da *ghost*.

[19]Foi adotado o processo de leitura de *input* transversal, o que implica alocar os *arrays* em todos os processadores.

A estrutura da paralelização é análoga à da $v2$, com as rotinas *aquire_hgrid* e *partition_hgrid* a serem utilizadas da mesma forma; as únicas alterações são a inclusão de $KDTree$ e a aplicação de ParMETIS à malha 2. A figura 4.2 mostra os resultados obtidos para a malha $BGround2$[20].

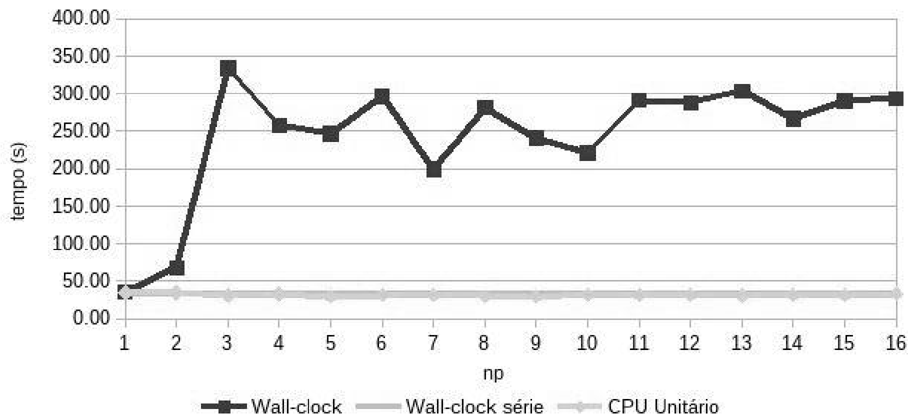


Figura 4.2: Tempos de execução para a malha BGround2 (v3).

Estes resultados são bastante surpreendentes, em particular o aumento irregular do tempo de relógio - salto para $np = 3$ seguido de aparente estabilização - e a sua magnitude, entre 8 a 10 vezes superior ao tempo série; por outro lado, o tempo de CPU unitário mantém-se aproximadamente constante e, sobretudo, consistente com o tempo de relógio da versão série, quando aquilo que se esperaria observar era um decréscimo progressivo do tempo de execução (mediante o aumento do número de processadores, aproximadamente como uma exponencial negativa). A origem deste comportamento está provavelmente relacionada com dois fatores:

1. O processo de *read* transversal induz um efeito de *bottleneck* que atrasa a execução do programa (*wall-clock*) mas não se reflete no tempo de CPU;

2. O $KDTree$ é tão eficiente que, perante uma estrutura do programa como esta[21], a introdução de ParMETIS não traz nenhum benefício e, pelo contrário, contribui para o aumento do tempo de execução através do seu *overhead*.

Um terceiro fator poderia ser relevante, a distribuição da malha, caso não estivesse a ser feita de forma regular e equilibrada. Para verificar isso é também medido na rotina de *timing* o desvio padrão da amostra, conforme descrito em 4.2.3, que dá valores na ordem de 0 a 2%, o que permite refutar a hipótese de má repartição da carga computacional.

Estas hipóteses sugerem que a aplicação desta estratégia a uma malha maior, ou a um programa com uma carga computacional mais substancial, surtiria melhores resultados. Seria interessante observar os resultados para a malha de maiores dimensões, $MAlt_BGround$, mas esta versão permanece incapaz de processar essa malha (o que era relativamente previsível, tendo em conta o excesso de memória que se torna necessária fazendo o *read* transversal).

v3.2

No sentido de suportar a hipótese de que o efeito do ParMETIS é ofuscado pela presença do $KDTree$, e também para tentar quantificar o *overhead* introduzido pelo ParMETIS, foi construída a versão v3.2 em que é mantido o ParMETIS tal como na versão v3 (aplicado à malha 2) mas é retirado o $KDTree$, ou seja, o ciclo de interpolação é corrido para uma submalha da malha 2, mas pesquisa correspondências para todos os elementos da malha 1. Os resultados observados encontram-se na figura 4.3.

[20] São semelhantes para $BGround1$ e *Batim*.

[21] Contemplam-se neste termo dois fatores: a dimensão da malha, que é pequena (apesar de os resultados exibidos dizerem respeito à maior entre as disponíveis), e a computação feita pelo programa, que neste caso é uma simples interpolação.

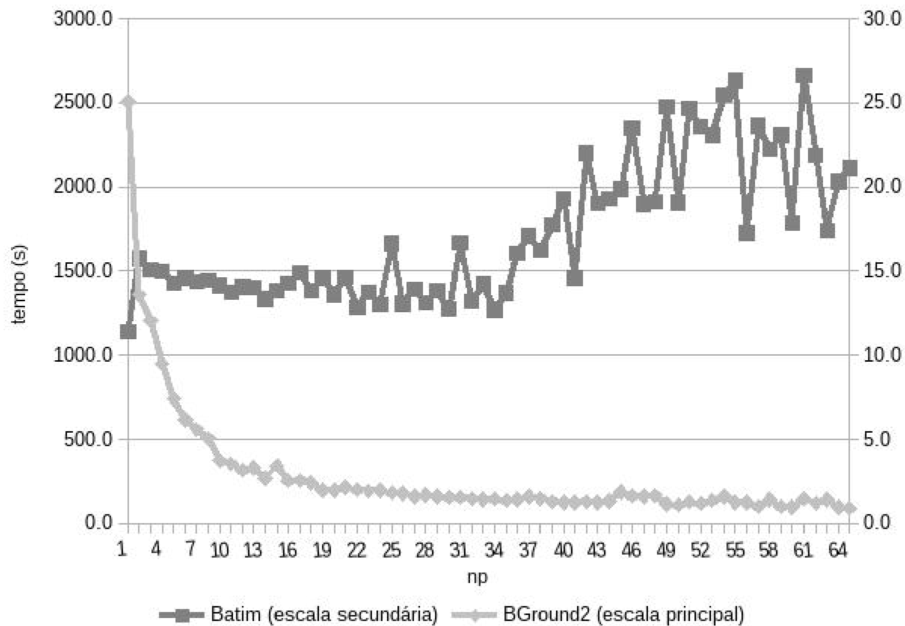


Figura 4.3: Tempos de execução para as malhas Batim e BGround2 (v3.2).

A primeira observação relevante é o aumento no tempo de execução que a remoção do $KDTree$ provocou. Este aumento é ligeiro para a malha $Batim$, que mantém tempos da ordem dos registados para a versão série, mas muito acentuado no caso da malha maior, $BGround2$ (uma a duas ordens de grandeza acima do verificado para v3). Este comportamento compromete obviamente qualquer ideia de usar esta versão na prática, pela sua evidente falta de escalabilidade, mas é útil para verificar o impacto do $KDTree$, em particular que a sua presença é tanto mais relevante quanto maior a malha de *background*.

Em segundo lugar, é de assinalar a forma da variação com np do tempo de execução, para a malha $BGround2$ - sem o $KDTree$, assemelha-se efetivamente à exponencial negativa que era esperada. Esta evidência suporta a hipótese de que o efeito do ParMETIS é encoberto pelo KDTree na versão v3.

Por último, no caso da malha Batim, é visível que o tempo de execução decresce até um certo ponto e a partir daí começa a aumentar; este comportamento é coerente com a assunção de que, a partir de certa altura, aquilo que se ganha em usar o ParMETIS já não e suficiente para compensar o seu overhead.

Globalmente, os resultados registados para esta versão permitem afirmar com alguma segurança que, para problemas desta dimensão, o efeito do $KDTree$ sobrepõe-se ao do ParMETIS.

v3.3 e v3.3.2

Subsiste a questão de tentar determinar qual o *overhead* introduzido pelo ParMETIS, de modo a poder definir um limite a partir do qual a sua utilização seja justificável. Para isso foi criada a versão $v3.3$, da qual foram removidas todas as funcionalidades do programa à exceção da leitura da malha 1 e da subdivisão (com ParMETIS) da malha 2. Visto que, retirando a componente principal de computação, o contributo do *overhead* da leitura do $input$ pode ser não-desprezável, foi ainda criada uma outra versão, $v3.3.2$, partindo de $v3.3$ e retirando ainda a leitura da malha 1, com o objetivo de verificar se, nessa situação, o *overhead* de ParMETIS era o mesmo independentemente da malha 1 utilizada. Os resultados obtidos encontram-se na figura 4.4

Nesse gráfico a linha que se destaca diz respeito à $v3.3$ para a malha $BGround2$; as restantes, que se confundem, correspondem a $v3.3.2$ para $BGround2$ e ambas as versões para $Batim$. Esta figura deixa claro que a parte de leitura da malha 1 pode ter um contributo significativo para o tempo de execução, dependendo da sua dimensão; esta observação é relevante porque, em situação real, a malha 1 a utilizar pode ser muito densa (25M de elementos, pelo menos), pelo que uma solução

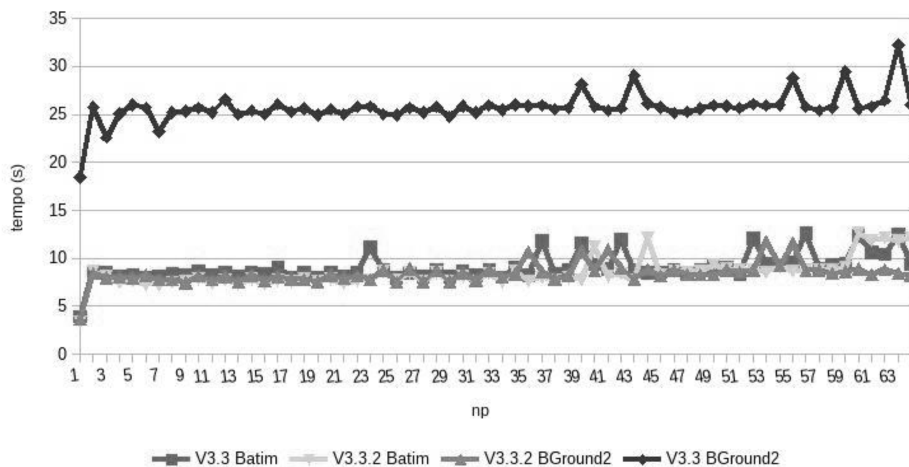


Figura 4.4: Tempos de execução para as malhas Batim e BGround2 (v3.3 e v3.3.2).

como a aplicada na v3 pode não ser viável por falta de escalabilidade. Impõe-se então, para malhas dessa dimensão, encontrar uma forma de particionar também a malha 1. A solução mais óbvia será aplicar-lhe ParMETIS, paralelamente à malha 2, o que implica coordenar ambas as partições para que cada elemento esteja no mesmo processador em ambas as malhas.

Comparando os tempos relativos à versão $v3.3.2$ (coincidentes para ambas as malhas), é evidente que a utilização do ParMETIS só por si - sem computação associada - não depende da malha 1. Desta forma, é possível estimar o *overhead* de ParMETIS como sendo da ordem de $8 - 10\,s$; este valor depende evidentemente da malha a que o ParMETIS é aplicado, e refere-se, neste caso, a uma malha de cerca de 400 mil elementos (a malha de interpolação utilizada). A diferença de *performance* entre as versões $v1$ e $v3$ era da ordem de $10s$ (considerando o tempo de CPU), que pode agora ser claramente atribuída ao *overhead* introduzido pelo ParMETIS que, como foi visto, para um problema desta dimensão não traz nenhuma vantagem observável quando usado em conjunto com o $KDTree$.

4.4 Conclusões

A $v3.3.2$ foi a última versão criada no âmbito desta abordagem. Enquanto que as primeiras versões visaram atingir os objetivos da paralelização definidos em 4.1, a partir da versão v3.2 o foco principal foi compreender o papel dos diversos fatores em jogo, numa perspetiva mais *académica*. Esta análise permitiu chegar a algumas conclusões:

1. A simples introdução do ParMETIS cria, para uma malha de 400.000 elementos, um *overhead* da ordem dos 10 segundos.

2. O impacto do ParMETIS não é percetível em problemas desta dimensão, quando usado em conjunto com $KDTree$, mas é previsível que se torne tanto mais relevante quanto maior a malha e a complexidade da computação.

3. A fase de leitura do *input* pode vir a ter um peso considerável no caso de malhas muito grandes, pelo que deve ser considerada a hipótese de particionar ambas as malhas em simultâneo ou a utilização de soluções de I/O paralelo.

Após a sucessão de versões *experimentais* (v3.2 a v3.3.2), foi criada uma versão estável, $v4$, com base em $v3$ mas retirando alguns elementos dispensáveis[22].

É importante frisar que a tentativa de executar a versão $v4$ para a malha $MAlt_BGround$ não foi bem-sucedida, por impossibilidade de alocar memória suficiente, o que fortalece a necessidade de

[22]Estes elementos existiam sobretudo como resquício do processo de adaptação das rotinas do modelo SELFE, não tendo qualquer utilidade.

subdividir a malha 1. Na verdade, esta malha não pôde ser processada em nenhuma versão além da original $interbath.f$, e nesse caso o tempo de execução foi de mais de 5 horas, o que não é obviamente satisfatório.

Resumindo, a tentativa de paralelização permitiu cumprir apenas um dos objetivos propostos - o da redução do tempo de execução - e através da abordagem que à partida mereceria menos consideração. É provável que a razão por detrás desta incapacidade de melhorar o tempo de execução usando ParMETIS se prenda com o *overhead* que esse método introduz e que, para problemas desta dimensão e com este volume de computação, ultrapassa os ganhos potenciais. Seria interessante poder aplicar este processo a um programa com maior carga computacional, onde fosse possível observar de forma mais clara a vantagem da integração do ParMETIS para o tempo de execução.

No que diz respeito ao $Interbath$, o problema permanece em aberto. O desenvolvimento de uma solução que permita cumprir simultaneamente os objetivos de menor tempo de execução e capacidade de processamento de maiores malhas de *background* passará necessariamente pela partição da malha de *background*, juntamente com a de interpolação. Uma hipótese será aplicar $KDTree$ localmente nessa partição da malha de *background*.

Capítulo 5

Trabalho Futuro

O trabalho desenvolvido no âmbito da bolsa de iniciação à investigação sobre computação paralela, descrito parcialmente neste relatório, permitiu tomar contacto com vários conceitos, métodos e ferramentas de paralelismo. Este contacto teve como resultado o desenvolvimento da capacidade de usar paralelismo para melhorar programas existentes e aproveitar o Medusa para daí extrair melhores e mais rápidos resultados. É expectável que este documento reúna recursos suficientes para permitir ao utilizador interessado dar os primeiros passos nesse sentido.

De futuro, será dada continuidade ao trabalho desenvolvido até aqui. Uma tarefa prioritária será a melhoria da última versão paralela do *Interbath*, ainda que numa perspetiva quase académica de compreensão do impacto das diversas componentes do programa no *speedup* final, uma vez que, como foi visto, a margem para melhoria efetiva (conseguir um tempo de execução inferior ao tempo série) é algo reduzida.

Uma outra perspetiva interessante é a de incluir *OpenMP* e explorar a hipótese de recorrer a estruturas híbridas, em particular para perceber se existe vantagem em usar o modelo de *threads* para aumentar o grau de paralelização do programa.

Será também relevante investigar ferramentas de paralelização avançada até agora não utilizadas, por exemplo para *profiling* de aplicações paralelas, bem como outros instrumentos não diretamente ligados à paralelização mas frequentemente usados de forma complementar, como é o caso da linguagem *python* usada no exemplo do *Interbath*.

Bibliografia

[1] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Exploiting task and data parallelism on a multicomputer," *SIGPLAN Not.*, vol. 28, no. 7, pp. 13–22, 1993.

[2] A. J. Van der Steen and J. J. Dongarra, *Overview of recent supercomputers*. Citeseer, 1995.

[3] B. L. Buzbee, "The efficiency of parallel processing," *Frontiers of Supercomputing, Los Alamos Science*, 1983.

[4] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, 1967.

[5] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, 1988.

[6] A. H. Karp and H. P. Flatt, "Measuring parallel processor performance," *Communications of the ACM*, vol. 33, no. 5, 1990.

[7] B. Barney, "Introduction to parallel computing." *https://computing.llnl.gov/tutorials/parallel_comp*. Consultado em 20-6-2013.

[8] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from berkeley," tech. rep., University of California at Berkeley, 2006.

[9] Microsoft, "The manycore shift," 2007.

[10] K. Yelick, "Parallel languages: Past, present and future," in *HOPL Conference*, 2007.

[11] "The state of parallel programming: The parallel programming landscape." *http://software.intel.com/sites/billboard/article/dr-dobbs-state-parallel-programming-research*, 2012.

[12] "Open mpi v1.6.4 documentation." *http://www.open-mpi.org/doc/v1.6/*. Consultado em 24-6-2013.

[13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.1*, 2008.

[14] N. MacDonals *et al.*, "Writing message-passing parallel programs with mpi," in *Course Notes*, University of Edinburgh, 1996.

[15] Y. Aoyama and J. Nakano, "Practical mpi programming," tech. rep., IBM, 1999.

[16] http://www.open-mpi.org/faq/?category=perftools. Consultado em 24-6-2013.

[17] http://wiki.ncg.ingrid.pt/index.php/Medusa. Consultado em 25-6-2013.

[18] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview. Consultado em 19-6-2013.

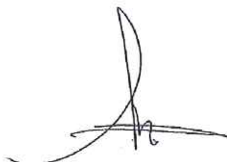
[19] http://www.stccmop.org/knowledge_transfer/software/selfe. Consultado em 12-6-2013.

[20] G. Karypis, K. Schloegel, and V. Kumar, *ParMETIS Parallel Graph Partitioning and Sparse Matrix Ordering*. University of Minnesota, 2003.

[21] A. W. Moore, *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, 1991. Excerto disponível em *http://www.ri.cmu.edu/pub_files/pub1/moore_andrew_1991_1/moore_andrew_1991_1.pdf*. Consultado em 27-6-2013.

VISTOS

A Chefe do NTIEC



Ana Lucas
Investigador Principal

AUTORIA

João Coelho
Bolseiro de Iniciação à Investigação Científica

António Inês
Investigador Principal