INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Avaliação dos mecanismos de concorrência na API do Java 8

DANIEL JORGE DE OLIVEIRA AGUAS
Outubro de 2015

POLITÉCNICO
DO PORTO

isep Instituto Superior de Engenharia do Porto

Instituto Superior de Engenharia do Porto ⊙ Rua Dr. António Bernardino de Almeida, 431 ⊙ 4249-015 Porto
Tel. +351 228 340 500 ⊙ Fax +351 228 321 159 ⊙ mail@isep.ipp.pt ⊙ www.isep.ipp.pt

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

isep

Avaliação dos mecanismos de concorrência na API do Java 8

DANIEL JORGE DE OLIVEIRA AGUAS
Outubro de 2015

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep




Avaliação dos mecanismos de concorrência na API do Java 8

DANIEL JORGE DE OLIVEIRA AGUAS
Outubro de 2015

POLITÉCNICO DO PORTO

Instituto Superior de Engenharia do Porto

Avaliação dos mecanismos de concorrência na API do Java 8

Daniel Jorge Oliveira Águas

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas, Sistemas e Redes

Orientador: Luís Miguel Pinho Nogueira

Júri:

Presidente:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, Outubro 2015

Aos que me desejam bem.

Resumo

As plataformas com múltiplos núcleos tornaram a programação paralela/concorrente num tópico de interesse geral. Diversos modelos de programação têm vindo a ser propostos, facilitando aos programadores a identificação de regiões de código potencialmente paralelizáveis, deixando ao sistema operativo a tarefa de as escalonar dinamicamente em tempo de execução, explorando o maior grau possível de paralelismo. O Java não foge a esta tendência, disponibilizando ao programador um número crescente de bibliotecas de mecanismos de sincronização e paralelização de código.

Neste contexto, esta tese apresenta e discute um conjunto de resultados obtidos através de testes intensivos à eficiência de algoritmos de ordenação implementados com recurso aos mecanismos de concorrência da API do Java 8 (*Threads*, *Threadpools*, *ExecutorService*, *CountdownLach*, *ExecutorCompletionService* e *ForkJoinPools*) em sistemas com um número de núcleos variável.

Para cada um dos mecanismos, são apresentadas conclusões sobre o seu funcionamento e discutidos os cenários em que o seu uso pode ser rentabilizado de modo a serem obtidos melhores tempos de execução.

Palavras-chave: núcleos, programação paralela, programação concorrente, sistema operativo, Java, sincronização, algoritmos, API

Abstract

Multicore platforms made parallel/concorrent programming a topic of general interest. Several programming models have been proposed, making it easier for Various programming models have been proposed, making it easier for developers to identify potentially parallelizable code regions, leaving to the Operative System the task of scheduling the run time, exploring the greatest possible degree of parallelism. Java is no exception to this trend, offering the developer a growing number of libraries with synchronization mechanisms and parallelization of code.

This thesis, within this context, presents and discusses a set of results obtained through extensive testing the efficiency of sorting algorithms implemented using the concorrency mechanisms of the Java API 8 (Threads, Threadpools, ExecutorService, CountdownLach, ExecutorCompletionService and ForkJoinPools) in systems with a variable number of cores.

For each of the mechanisms, findings are presented on their functioning and discussed the scenarios where their use may be more advantageous in order to secure better runtimes.

Keywords: Multicore, parallel/concorrent programming, Operative System, Java, synchronization, algorithms, API

Agradecimentos

Gostaria de agradecer a todos que contribuíram ao longo do desenvolvimento da dissertação por todo o apoio e auxílio que me ofereceram.

Agradeço em primeiro lugar ao Doutor Luís Nogueira que foi o principal impulsionador para a ideia da tese e também pela disponibilidade ao orientar-me no decorrer destes meses.

Um especial obrigado também dirigido aos professores do Departamento de Engenharia Informática do ISEP, pelos conselhos oferecidos, pela ajuda em momentos de necessidade e pela disponibilização de acesso à sala de desenvolvimento de projetos de estágio/teses.

Aos amigos com que partilhei a sala, gostava de agradecer pela ajuda fornecida e pelo bom ambiente que foi possível manter mas acima de todos aos meus amigos Edgar Moreira e Hugo Leal que me permitiram acesso aos seus computadores no decorrer dos testes das implementações.

Por fim, quero agradecer à minha família pelo esforço que fizeram ao apoiar-me e acreditarem em mim mas em especial à minha namorada que esteve sempre do meu lado e fez com que eu fosse capaz de ultrapassar todos obstáculos.

A todos, um sincero obrigado.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

RAM	*Random Access Memory*
CPU	*Central Processing Unit*
JDK	*Java Development Kit*
IDE	*Integrated Development Environment*
JVM	*Java Virtual Machine*
SO	Sistema Operativo
FIFO	*First-In-First-Out*
LIFO	*Last-In-Last-Out*
Deque	*Double Ended Queue*
CSV	*Comma-Separated Values*
HTML	*HyperText Markup Language*
GC	*Garbage Collector*
FJP	*ForkJoinPool*
WS	*WorkStealing*

Lista de Símbolos

σ	Desvio padrão
µs	Microssegundo
iC	Intervalo de confiança
$\mathbf{Z\alpha/2}$	Valor crítico
MB	Megabyte
GB	Gigabyte

1 Introdução

Um algoritmo é uma representação de um conjunto de eventos finitos a realizar de modo a obter uma solução para um problema ou tarefa de modo a atingir um objetivo. No ramo da computação um algoritmo, ou uma coleção destes, é um programa que irá executar passo a passo para concluir uma operação.

Estes programas são desenvolvidos na generalidade dos casos em uma linguagem de programação. Esta é importante pois adota uma sintaxe de alto nível mais facilmente entendida por humanos e realiza a tradução do código desenvolvido para código máquina, que será executado pela Unidade de Processamento Central (CPU), também conhecida como processador.

O processador funciona como o cérebro do computador, executando um conjunto de operações aritméticas e lógicas sobre dados e coordenando o fluxo de informação. Originalmente este foi desenvolvido como uma componente com apenas uma única unidade de processamento, conhecida como núcleo. Este apenas efetuava uma sequencia de instruções num determinado instante e à medida que a sua velocidade de processamento aumentava o consumo de energia acompanhava. Este aumento continuou com a evolução dos processadores, até esta subida que atingir a *Power Wall*. Este é o ponto em que as velocidades dos processadores não conseguiam amplificar sem os custos de energia tornarem-se excessivamente elevados para serem suportados. Este problema conduziu a uma abordagem de paralelização dos CPUs, uma componente com múltiplos núcleos de processamento, possibilitando a execução a um número de sequências de instruções equivalente ao número de núcleos presentes. Estes sistemas *multicore* são responsáveis por distribuir as tarefas entre si, concebendo um ambiente multitarefa e possibilitando melhorias nos tempos de resposta através do paralelismo.

Depois de anos de evolução neste momento no mercado é possível encontrar sistemas *multicore*, desde dois núcleos até oito para computadores pessoais, sendo que atualmente para servidores já existem versões que ultrapassam os sessenta núcleos [Intel Xeon Phi, 2015]. Apesar do contínuo aumento do número de núcleos presentes no sistema, outras habilidades foram incorporadas. A mais comum de se encontrar atualmente denomina-se de *Hyper-Threading*. Esta dá a capacidade ao Sistema Operativo (SO) de virtualizar o processamento de cada núcleo físico, em dois lógicos, partilhando a mesma *cache*. Esta partilha resulta na atribuição dos

mesmos recursos a duas *threads*, em que cada uma é um caminho independente de execução de código de um processo.

Com a introdução de todas estas melhorias a nível de performance, em relação aos sistemas de um único núcleo, continua a existir bastante dependência do algoritmo usado e da sua implementação. A principal preocupação a ter em conta e a procurar soluções, é como tirar o máximo de proveito de toda esta evolução de *hardware*. A solução para os sistemas com múltiplos núcleos conduziu a que os desenvolvedores substituíssem a programação sequencial pela paralela, que apesar das melhorias significativas de performance também sucedeu um aumento na dificuldade na programação. Agora era necessário pensar não só em distribuir o trabalho de maneira a existir concorrência, mas igualmente ponderar na sincronização de acesso a dados partilhados.

A programação concorrente é a designação atribuída à programação, às técnicas para expressar o potencial paralelismo e à resolução de problemas na construção de sincronismo e comunicação. Esta a nível de implementação, esta torna-se mais cansativa de ser implementada que a sequencial. Para se alcançar desfechos favoráveis de performance necessita-se de ultrapassar imensas dificuldades como *race conditions*, problemas de *software* devido à introdução de novas classes, permutação de informação e sincronismo entre as *tasks*.

A linguagem de programação Java começou a ser desenvolvida no inicio da década dos anos 90 pela Sun Microsystems. A execução de programas desenvolvidos nesta é baseada em *threads* e neste momento persiste com um estruturado conjunto de bibliotecas para apoio à programação concorrente. A programação é possível com um ambiente de desenvolvimento integrado (IDE) e com o uso de uma máquina virtual Java (JVM), que neste momento se encontra suportada para os SO mais comuns. Esta é uma aplicação de *software* que simula um computador, ocultando o Sistema Operativo e o *hardware* que interage com a JVM e realiza a tradução dos *bytecodes*, código fonte traduzido pelo compilador Java (javac), na respetiva linguagem para a máquina executar. Atualmente as principais implementações são combinadas em um produto alvo para desenvolvedores num Java Development Kit (JDK), lançado pela Oracle Corporation que no presente são os proprietários da Sun Microsystems. O JDK vem equipado com uma máquina virtual Java, uma grande coleção de ferramentas de apoio e uma API (Application Programming Interface) expressa como um conjunto de classes com uma lista de métodos associada a cada.

Os utilizadores de computadores nos dias de hoje esperam que os seus sistemas empreguem mais de uma operação em simultâneo, seja em aplicações separadas ou em uma única. A plataforma Java desde a versão 5.0 inclui uma API de auxílio à programação de alto nível concorrente. Atualmente esta encontra-se na versão 8.40 e as bibliotecas de concorrência oferecem um aglomerado número de ferramentas para ajustar a performance dos algoritmos.

O trabalho sobre o qual convenciona esta dissertação baseia-se assim numa investigação das bibliotecas de concorrência disponíveis e de algoritmos. Com a finalidade de demonstrar um conjunto de resultados depois de efetuada uma elaborada coleção de testes intensivos. A partir dos resultados, serão elaborados um conjunto de princípios, provando-os usando técnicas

matemáticas, de modo a melhorar a programação concorrente e determinando a razão da performance consoante as implementações.

1.1 Motivação

Esta dissertação baseia-se em analisar os consumos de tempo e de memória de diferentes implementações de algoritmos. Elaborando possíveis versões paralelas dos algoritmos base, explorando a possibilidade de aperfeiçoamentos na performance, utilizando as bibliotecas de concorrência na atual API do Java. Através da comparação dos diferentes consumos é esperado ser possível obter análises que possibilitem elaborar conclusões sobre as vantagens e desvantagens dos mecanismos de concorrência em computadores com arquiteturas de *hardware* diferentes.

O desenvolvimento desta dissertação e o código implementado são baseados na linguagem de programação Java. A escolha deve-se à simplicidade da linguagem, aos mecanismos disponíveis de sincronização e de interface que permitem uma implementação adequada ao esforço. Deve-se também ao dinamismo em que das classes da API só serão carregadas quando necessárias, reduzindo o consumo de memória e o tempo de arranque do algoritmo. A robustez é outra vantagem devido à confiança que nos dá a verificar os erros através dos compiladores antes da execução. Através da capacidade de ser multiplataforma é permitindo a execução das verificações de resultados em diferentes Sistemas Operativos. Por fim, o fato de ser *multithread* que consiste na capacidade de executar diversas tarefas em simultâneo num programa.

Através da elaboração de diversas versões dos mesmos algoritmos aproveitando as bibliotecas de concorrência, é possível adquirir a capacidade de comparar a eficiência das ferramentas, obtendo conhecimento sobre quais os cuidados que os programadores devem de ter durante o decorrer da criação dos seus projetos.

1.2 Organização do documento

Esta dissertação encontra-se organizada em seis capítulos. No primeiro capítulo, é apresentando brevemente o estudo realizado assim como a motivação que levou à realização do projeto. No capítulo número dois é apresentada no estado de arte a investigação dos mecanismos de concorrência da API e da linguagem de programação JAVA. O capítulo três foca-se na explicação da estrutura de funcionamento do projeto, apresentando os principais mecanismos da API e algoritmos que preencham os objetivos traçados, os procedimentos de verificação de dados e como os resultados são apresentados. No capítulo número quatro são apresentadas as implementações elaboradas assim como os fluxos de funcionamento e indicando os problemas e soluções encontradas. O capítulo cinco expõe os resultados obtidos recorrendo a justificações sobre cada ponto de interesse, indicando também as principais vantagens e desvantagens de cada implementação. No último capítulo, são apresentadas as conclusões finais obtidas com a realização deste estudo indicando os principais aspetos a realçar e sugestões de maneira a dar continuidade a este projeto no futuro.

2 Concorrência e Java

Os primeiros computadores não continham Sistema Operativo e apenas executavam um único programa em cada instante, desde o início até ao fim da sua execução, esse programa tinha o acesso total aos recursos da máquina. Esta metodologia tornava-se muito ineficiente e dispendiosa devido à escassez dos recursos dos computadores. Ao longo do tempo os SO evoluíram e tal levou à introdução de conceitos de execução de múltiplos programas em simultâneo e partilha de recursos.

Concorrência, como termo informático, é utilizada quando um programa tem a capacidade de executar mais do que uma tarefa no mesmo instante. Contudo esta descrição de execução tarefas em paralelo pode ser não exata. Esta inexatidão acontece pois apesar do sistema poder, de fato, conseguir realizar a separação das atividades pelos diferentes processadores, em que cada um trabalha em separado, outros casos são possíveis verificar. O sistema é capaz de simplesmente executa as tarefas num único núcleo, levando este a partilha o seu tempo de execução, realizando substituições entre as tarefas, a uma velocidade que causa a ilusão de execução em simultaneidade.

A programação concorrente no Java consiste na utilização de construtores que possibilitem mapear as atividades independentes, através da Máquina Virtual Java e do Sistema Operativo, para executarem em paralelo. Este mapeamento é possível se pretendido ou praticável através dos diversos núcleos disponíveis, ou na carência destes através da partilha de tempo de CPU. [Douglas Lea, 1999]

A partir da versão 5 do Java, também conhecido como Java 5, a API foi lançada com um *package java.util.concurrent* que introduziu um conjunto de classes e interfaces que permitiam um nível de programação concorrente muito mais simples e fácil de implementar. Esta versão era o contrário das versões anteriores que, apenas continham mecanismos de sincronização de *threads* muito limitados e difíceis de implementar corretamente. [Peter Andersen Busterud, 2012]

Quando introduzido o *package concurrent*, este incluiu um pequeno conjunto de *frameworks* para que a programação concorrente se tornasse padronizada e ainda incluiu duas novas *sub-*

packages, *Lock* e *Atomic*. Estas *packages* continham um conjunto de ferramentas que permitiam uma gestão mais correta da sincronização de recursos partilhados por múltiplas *threads*, através de operações condicionais. Com a introdução de novas ferramentas de apoio à concorrência na versão Java 6, Java 7 e na versão atual Java 8, a biblioteca de concorrência contém, atualmente, uma enorme variedade de ferramentas de apoio ao desenvolvimento de *software*. Assim, atualmente é possível produzir implementações com um número muito reduzido de linhas de código quando comparado com as necessárias prévias à versão Java 5.

Atualmente a versão Java 8 é suportada por apenas quatro máquinas virtuais a JVM Hotspot, a JamVM, a Zing e a Zulu. Todas estas versões foram analisadas para uma possível utilização no decorrer deste estudo. Na seguinte tabela [Tabela 1] é possível verificar as propriedades das máquinas virtuais Java, que foram consideradas de maior relevo para o estudo.

Tabela 1 – Comparação de Máquinas Virtuais

Nome	OS Windows	MAC OS X	OpenJDK	Processador x64	Disponibilidade
Hotspot	Sim	Sim	Sim	Sim	Grátis
JamVM	Não	Sim	Não	Sim	Grátis
Zing	Não	Não	Sim	Sim	Comercial
Zulu	Sim	Sim	Sim	Sim	Grátis

Estas foram as propriedades mais significantes escolhidas, de acordo com os recursos disponíveis para a realização do estudo. De acordo com os resultados, a máquina virtual Java selecionada para uso foi a JVM Hotspot pois, apesar da máquina Zulu apresentar resultados equivalentes, a JVM Hotspot já se encontrava presente no material adquirido para a execução dos testes. Uma outra vantagem para o uso da JVM Hotspot é a existência de distribuições de um pacote completo fornecido pela Oracle que inclui a distribuição JDK mais recente, a JVM e o IDE Netbeans.

2.1 Noções de Concorrência e Paralelismo

Para melhor entendimento desta dissertação, existe um agrupado de conceitos que são imprescindíveis de se compreender, esta secção vai concretizar uma simplificação dos mesmos para futuras observações.

O termo paralelismo é utilizado na programação quando um conjunto de *threads* estão a progredir em simultâneo, dependendo do número de núcleos desocupados, onde cada *core* é responsável por uma única *thread*. Por outro lado, na programação concorrente o mesmo núcleo pode ser partilhado para a progressão de múltiplas *threads*, onde os seus tempos de execução serão agendados. Assim é possível determinar que o paralelismo é uma parte da concorrência. Visto que a programação concorrente utiliza múltiplas *threads,* se não existirem múltiplos núcleos nunca será possível ocorrer execução paralela. [Clay Breshears, 2009]

Quando um programa é iniciado, o SO produz um processo com o código e os dados alusivos a esse e gere o processo no decorrer da sua execução. Os Sistemas Operativos com a capacidade de multiprocessamento possibilitam a execução de diversos programas em simultâneo, partilhando os recursos como a memória e núcleos físicos do processador. Uma *thread* é uma unidade do processo, que executa uma função do programa consumindo os recursos enquanto ativa, principalmente a memória. Quando o processo de um programa em execução é iniciado com uma *thread* única, esta é chamada de *main thread* e executa a função *main* do programa em questão. Esta *thread*, num programa que implemente múltiplas *threads*, é responsável por produzir as novas *threads* com as suas próprias funções e com a capacidade de produzir novas *threads*. Na Figura 1 é possível observar-se uma visão do sistema. A criação de todas as *threads* é realizada através de construtores fornecidos pela API. [Richard H. Carver and Kuo-Chung Tai, 2005] A execução de código que se realiza numa *thread* é considerada uma *task*. Esta pode ser uma pequena fração de um problema (ex. *divide-and-conquer*), podendo trabalhar em paralelo com outras *tasks* para produzir a solução final. [Peter Andersen Busterud, 2012]

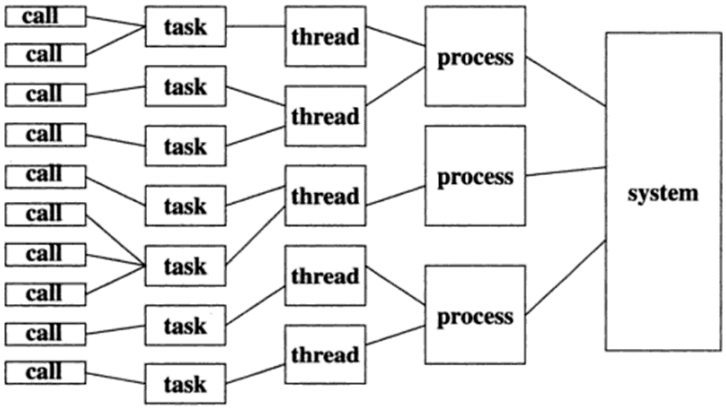

Figura 1 – Representação de programa *multithreaded* [1]

A criação de múltiplas *threads* pode conduzir a uma situação em que estas ultrapassem a quantidade de núcleos disponíveis, sempre que isto se concretiza o SO efetua o agendamento baseado em políticas. Estas políticas alteram consoante a máquina virtual Java e o Sistema Operativo sobre o qual ela opera. O caso geral mais praticado nos SO para o agendamento baseia-se em preemptividade. No caso das *threads* se encontrarem numa fase do ciclo de vida em que não são capazes de realizar trabalho, estas não são consideradas pelo escalonador. As *threads* que de fato se encontram preparadas, recebem um espaço de tempo para execução no processador. No caso de uma *thread*, num determinado momento, ter que aguardar por um evento ou tenha consumido o seu período de tempo atribuído, esta é interrompida. Assim é possível que outra *thread* entre em funcionamento. Esta mudança de tarefas em execução no processador é conhecida como *context switch* e precisa que o estado do processo seja armazenado, uma vez que o processo pode ser interrompido e lido o estado de um novo. As substituições entre *threads* em execução podem ocorrer várias vezes ao longo de um segundo o que pode causar um potencial *overhead* [Richard H. Carver and Kuo-Chung Tai, 2005]. Isto é, resumidamente um excesso de tempo de execução, consumo de memória ou outro recurso partilhado.

[1] Douglas Lea. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition, 2009

2.2 Benefícios e Riscos de *Multithreading*

A utilização de *multithreading* incorpora um conjunto de vantagens, como a diminuição dos tempos de execução devido à possibilidade de uma *thread* prosseguir com a sua execução, enquanto outra espera pela ocorrência de um evento de parte de terceiros. Também é possível transformar numa interface gráfica mais responsiva, uma vez que a *thread* que controla os eventos da interface tem a capacidade de criar novas *threads*, para realizar tarefas de longa duração em resposta a determinadas circunstâncias. [Richard H. Carver and Kuo-Chung Tai, 2005]

Os sistemas multiprocessadores cada vez têm mais núcleos. Este incremento de núcleos tem tendência a ocorrer cada vez mais vezes, uma vez que as velocidades de processamento são muito mais custosas de aumentar. Os programas com o objetivo de utilizar o número de processadores disponíveis eficientemente, garantirão que as consequências na performance serão cada vez mais positivas, através de uma cuidada implementação de código e de prática das ferramentas de concorrência.

Já como referido no capítulo 1, a mudança do tipo de implementação de código pode torna-se problemática para o programador, uma vez que este necessita de um conhecimento profundo dos mecanismos que tem ao dispor e da possível coleção de erros e como os corrigir. Outro sintoma referido no capítulo 2.1 é o *overhead* gerado pelo *context switching*, criando a necessidade de ter uma boa gestão das *threads*, para não criar excessos superiores ao necessário. As *threads* precisam de recursos para trabalharem para além do processador, estas necessitam também de memória para a própria *stack* local. Na *stack* local das *threads*, os dados de execução serão armazenados, no acontecimento de ultrapassar espaço atribuído ocorrerá um erro do nomeado de *stack overflow error*.

Uma vez que as *threads* compartilham endereços de memória, estas usufruem da habilidade de modificar valores de objetos que outras *threads* possam utilizar. Apesar de este fato ter bastante conveniência, ao possibilitar a partilha de informação de uma forma deveras simples, também pode conduzir negativamente a problemas com alguma complexidade. É preciso atenção ao permitir que múltiplas *threads* tenham acesso a objetos, e às permissões para editar de um modo imprevisível os seus dados. Assim ao autorizar-se tais fatos, gera-se a necessidade de muito cuidado e de uma coordenação correta para que as *threads* nunca interfiram entre si. [Brian Goetz et al., 2006]

2.3 Segurança ao utilizar *threads*

A produção de programas concorrentes em que o código seja *thread-safe*, baseia-se na gestão de acesso à informação, que possa ser modificada, dos objetos que estejam a ser partilhados. Tornar um objeto *thread-safe* é utilizar sincronização para coordenar a modificação e o acesso aos dados do mesmo. Se o objeto necessita ou não de ser *thread-safe* depende unicamente se vai ser acedido por múltiplas *threads*, e caso a implementação desta segurança falhe ou não exista pode levar à corrupção de valores e resultados inesperados ou indesejados. O principal mecanismo de sincronização no Java é o uso do mecanismo *synchronized*, este fornece um bloqueio exclusivo.

No caso de múltiplas *threads* acederem ao mesmo dado de um objeto sem a sincronização apropriada, a implementação do programa deve ser considerada errada e deve proceder à sua correção. É possível retificar os erros utilizando sincronização sempre que é acedido o estado da variável, tornar esse valor inalterável ou simplesmente não partilhar dados dos objetos entre *threads*.

Uma classe pode ser *thread-safe* se continuar a comportar corretamente quando acedida por diversas *threads*, independente do escalonado. Se os objetos forem implementados corretamente nenhuma sequência de operações, sequenciais ou concorrentes, conseguiriam criar dados errados de objetos. [Brian Goetz et al., 2006] Mesmo que os objetos se encontrem completamente seguros através de *locks* e *synchronized*, múltiplas *threads* podem complicar a execução entre elas ao adquirem o acesso às *locks* de objetos partilhados entre várias *threads*. O principal exemplo deste fenómeno é o *deadlock*. Este acontece quando uma *thread* adquire a *lock* a um objeto e uma segunda *thread* em paralelo adquire de um outro objeto. Durante a execução a segunda *thread* antes de libertar a *lock* do primeiro objeto, precisa de adquirir a *lock* do objeto da primeira *thread*. Esta, por sua vez, para libertar a sua *lock* necessita de adquirir a *lock* do objeto adquirido pela segunda *thread*, criando uma corrente de necessidades. Este exemplo é ilustrado na seguinte imagem [Figura 2].

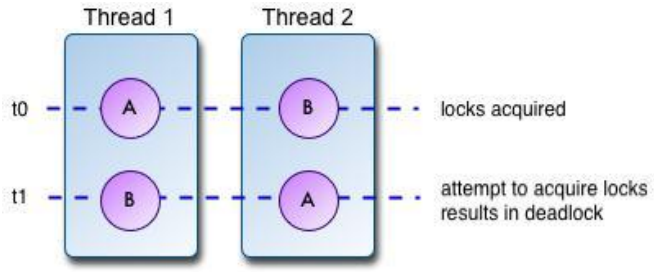

Figura 2 – Exemplo de *deadlock* [2]

2.4 Concorrência no Java

Nesta secção, irão ser abordados os principais mecanismos presentes na API da versão Java 8. Será elaborada uma breve descrição destes mecanismos analisando as classes, interfaces e *frameworks*. A versão mais recentemente disponibilizada da plataforma Java contém um a API composta por uma enorme quantidade de *packages* que por sua vez podem ou não contem *subpackages*. Sendo a concorrência o principal destaque, este estudo forcar-se-á essencialmente no package *java.util.concurrent*.

Devido à enorme dimensão atual da API da plataforma Java, foi necessário escolher um número de mecanismos a serem investigados implementando assim um conjunto de algoritmos para produzir conclusões que, permitissem uma boa aprendizagem de como elaborar uma correta implementação com concorrência.

[2] https://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/4/html/images/deadlock-definition.jpg

2.4.1 *Thread*

Em Java *threads* são objetos instanciados da classe *java.lang.Thread* ou de subclasses desta. A classe *Thread* e as suas subclasses são a base de toda a *framework* de concorrência existente no Java e permitem a execução de código utilizando *threads* de duas maneiras. A primeira baseia-se na construção de uma classe e à aplicação da extensão de *Thread*, tornando esta classe numa subclasse de *Thread* aplicando herança. Implementando o código que se deseja executar no método *run()*, este será iniciado assim que objeto chamar o método *start()*. A segunda alternativa é muito semelhante à primeira, mas com duas diferenças importantes. Na construção da classe está implemento *Runnable*, substituindo a extensão de *Thread*. De forma a criar a *thread* e a iniciá-la, é necessário criar o objeto *Runnable*, enviá-lo por parâmetro numa *thread* e invocar o método *start()*.

Durante a construção de uma *thread* é possível atribuir um nome à mesma para mais fácil identificação através dos construtores disponíveis. A identificação é possível ao enviar um dado do tipo *String* por parâmetro na criação do objeto. Caso a *thread* seja criada utilizando um *Runnable* o atributo para o nome deve ser enviado como segundo parâmetro. Cada *thread* tem uma prioridade, e as que possuam uma prioridade superior serão executadas com preferência sobre as que tenham uma prioridade menor. Quando uma *thread* produz uma nova thread, esta última fica com uma prioridade atribuída igual à *thread* que a produziu. As prioridades no Java podem variar desde o valor mínimo 1 até o máximo de 10, já existem constantes atribuídas na classe que podem ser utilizadas para a prioridade desejada assim como, *MIN_PRIORITY(1)*, *NORM_PRIORITY(5)*, e *MAX_PRIORITY(10)*. Por norma as *threads* quando criadas são iniciadas com uma prioridade com valor 5. Quando duas ou mais *threads* têm diferentes prioridades, o escalonador escolherá a que tiver atribuído o maior valor. Caso tenham a mesma prioridade o escalonador utilizará o algoritmo *round-robin*, em que este concede um bloco de tempo igual a cada e realizará a gestão de cada. [JavaSE 8, 2015]

Cada *thread* tem o seu próprio ciclo de vida ao longo da sua execução, atravessando diversas fases desde o momento que nasce até o da sua morte. Estas fases podem ser examinadas na Figura 3.

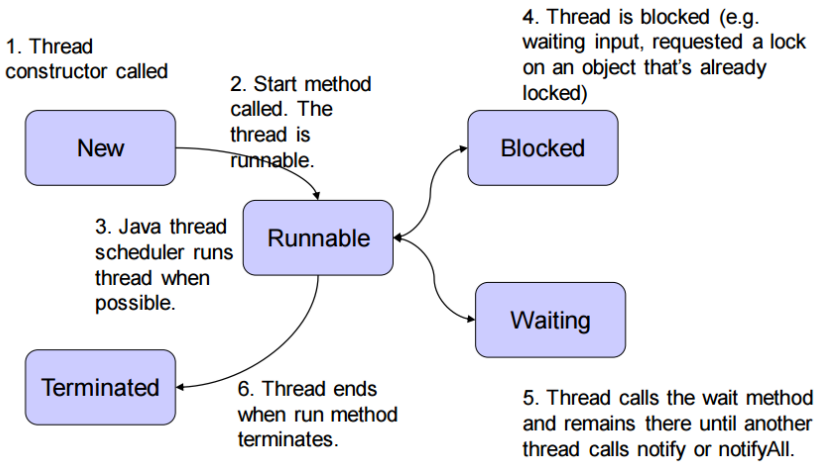

Figura 3 – Ciclo de vida de uma *thread* [3]

[3] Documentação de SOCOF do mestrado em Arquicteturas, Sistemas e Redes no ano lectivo 2014/2015

Entre os principais métodos disponíveis na classe *Thread* destacam-se os seguintes, com a explicação da sua utilidade.

- *Static boolean isInterrupted()*

  Este método retorna, se a *thread* em questão encontra-se interrompida ou não, devolvendo um valor do tipo booleano. Este valor vai retornar *true* se a *thread* estiver interrompida ou *false* se o oposto. Caso a *thread* se encontre não no estado *Runnable* quando lançada a interrupção da mesma, é retornado o valor false. Este método permite ao programador utilizar *threads* cujo trabalho é desnecessário, oferecendo uma nova utilidade ou se desejado terminar o funcionamento desta.

- *Static void yield()*

  Com o uso deste mecanismo é possível informar o escalonador que a *thread* em questão se encontra disponível para abdicar da utilização do processador, apesar de ainda existir código para executar. Esta informação pode ser ignorada pelo escalonador. Com o uso deste método é possível melhorar a performance da aplicação, afinando a progressão de *threads* que utilizariam em demasia o processador.

- *Static void sleep(int millis)*

  Utilizando este procedimento é possível especificar o número de milissegundos que se pretende que a *thread* suspenda temporariamente a sua execução, passando para o estado de *Blocked*. O escalonador é então informado que a *thread* não necessita do restante bloco de tempo e para não atribuir outro bloco enquanto se encontra a "dormir". É possível ser mais específico na quantidade de tempo que se deseja bloquear a *thread* ao enviar também quantidade em nanosegundos no método *public static void sleep(long millis, int nanos)*.

- *Void join()*

  A invocação deste método por um objeto do tipo *Thread*, que se encontre em execução, faz com que as *threads* restantes esperem que *thread* que invocou o *join()* termine. Caso seja necessário estabelecer um sistema de espera definindo o tempo que se esteja disposto a esperar, pode-se o fazer utilizando a mesma arquitetura de argumentos que o método *sleep()* anteriormente referido.

- *Void setPriority(int newPriority)*

  Utilizando este método é possível modificar a prioridade de uma *thread*. Esta definição está protegida pelos limites já mencionados e uma exceção de segurança que é acionada na ocorrência da tentativa de mudança da prioridade sem possuir permissões para tal.

- *Final void wait(), Final void notify()* e *wait()*

  Estes métodos permitem o bloqueio de uma *thread* quando esta os invoca. Este bloqueio é útil para libertar a *lock* associada ao objeto até que uma outra *thread* invoque o método *notify()* ou o *notifyall()*. Uma *thread* que se encontre em espera também pode ser desbloqueada. Se uma outra *thread* a interromper, é lancada uma *InterruptedException*. Também é possível limitar a duração de um bloqueio de uma *thread* ao utilizar os argumentos de tempo em simultâneo com o método *sleep()*. [JavaSE 8, 2015]

2.4.2 *Executor Framework*

Os mecanismos de funcionamento dos processos são as *threads* e nestas pode-se ter uma coleção de *tasks* a funcionar assincronamente. Ao executar as *tasks* utilizando as *threads* é possível dividir a quantia de *tasks* pela totalidade de *threads* ou executar cada *tasks* numa *thread* separada. Ambas abordagens têm as suas limitações de *performance* e de gestão de recursos. Para ultrapassar estas limitações utiliza-se a *framework Executor* da API do Java. Esta *framework* funciona bem em aplicações de grande escala pois, esta executa a gestão e criação de *threads* libertando esforço de parte dos programadores sem interferir com resto da aplicação.

O *package* de concorrência do Java é incorporado com três interfaces de Executor. A interface *Executor*, que fornece o apoio necessário na criação de novas *Runnable tasks*. A subinterface *ExecutorService* que introduz mecanismos de gestão para as *tasks* e para o executor em si, e a subinterface *ScheduledExecutorService* que faz o suporte à execução de *tasks* que, se repitam ou não periodicamente. O *ExecutorService* é um mecanismo que permite que uma *thread* encarregue *tasks* a este serviço para serem executadas em paralelo, como possível visualizar na seguinte ilustração [Figura 4].

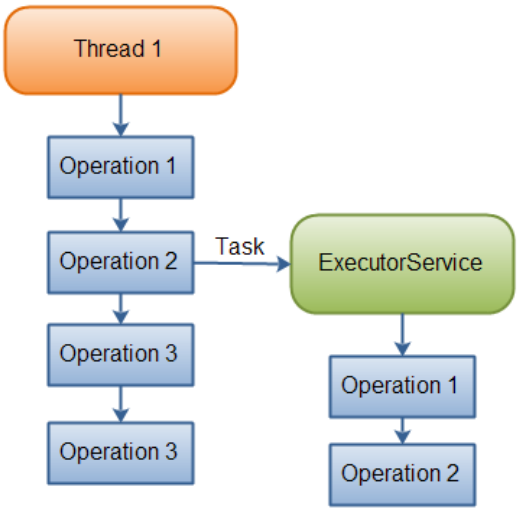

Figura 4 – Exemplo de funcionamento de ExecutorService [4]

[4] http://tutorials.jenkov.com/images/java-concurrency-utils/executor-service.png

A interface *Executor* fornece um único método, *execute()*, este permite substituir a criação de *threads* tradicional. Por exemplo a declaração *newThread(r).start()* passa a ser *e.execute(r)*, em que *r* é um objeto do tipo *Runnable*. Na implementação do Executor é possível atribuir a este uma *ThreadPool*. Esta pode ser atribuída em quatro formas:

- *newCachedThreadPool*

  Esta é uma *threadpool* dinâmica que se expande quando necessário, adicionando uma nova *thread* à *pool* sempre que não existir nenhuma disponível.

- *newSingleThreadExecutor*

  Uma *threadpool* com uma única *thread* disponível para operações, garantindo que as *tasks* são executadas sequencialmente e apenas uma única *task* estará ativa num determinado instante de tempo.

- *newFixedThreadPool*

  Esta alternativa permite submeter um número fixo de *threads* para reutilização, através da passagem de parâmetros na criação desta. Caso nenhum parâmetro seja enviado a *threadpool* é produzida com um número de *threads* igual ao número de *cores* presentes na máquina. Caso uma *task* seja submetida sem disponibilidade de parte da *threadpoll*, a *task* será enviada para um *queue* de espera até ocorrer a libertação de uma *thread*.

- *newWorkStealingPool*

  A última alternativa baseia-se em cria uma *threadpool* que implemente a estratégia *work-stealing*. Nesta implementação o escalonador redistribui o trabalho pelas múltiplas *threads* caso a distribuição não se encontre balanceada. Este tipo de *threadpool* permite definir o nível de paralelismo, quantia de *threads* presentes na *threadpool*, através da passagem de um valor do tipo inteiro por parâmetro. Caso nenhum valor seja enviado, o valor de paralelismo será definido como o número de processadores disponíveis.

A interface *ExecutorService* contém os seguintes métodos para executar *tasks*:

- *execute(Runnable)*

  Este método recebe um objeto do tipo *java.lang.Runnable*, o mesmo utilizado para *threads*, e executa-o assincronamente. Para obter retorno de resultados não é possível utilizando objetos *Runnables* e por isso é necessário utilizar *Callables*. Quando criado objecto do tipo *Callable*, este é alocado na *queue* de execução do *ExecutorService* e iniciado quando possível.

- *submit()*

  Existem duas versões para a implementação deste método. A primeira recebe como parâmetro um objeto do tipo *Runnable* e a outra que admite a passagem de um do tipo

Callable. Utilizando o método *submit()* com o *Runnable* é retornado um objeto do tipo *Future*. Este *Future* permite verificar se o *Runnable* já terminou a sua execução. Caso tenha terminado corretamente é retornado *null* ao invocar o método *get()*. A alternativa funciona de maneira semelhantemente mas permitindo obter resultados. O método *run()* na implementação *Runnable* é do tipo *void*, não permitindo retorno, já a interface *Callable* permite o retorno de resultado através do método *cal()*. Invocando método *get()* numa implementação com *Callables* é efetuado um bloqueio até que a *task* termine a sua execução, igual à implementação *Runnable*, mas também retorna o resultado do método *cal()*.

- *invokeAny(Collection<? extends Callable<T>> tasks)*

  Com a utilização deste método é possível a passagem de uma coleção de objetos do tipo *Callable* ou subinterfaces deste, retornando o valor do *Callabel* que termine primeiro com sucesso. Por outro lado, o *invoqueAll Collection<? extends Callable<T>> tasks)* executa a coleção de *tasks* atribuídas e retorna uma lista de *Futures* com os seus estados e resultados assim que todas terminem. É possível em ambas as implementações atribuir uma *deadline* através da incorporação de dois argumentos depois da coleção de *Callables*, um valor do tipo *long* e um valor do tipo *TimeUnit* para especificar a unidade de tempo para o *timeout*.

Com a implementação de um *ExecutorService*, o uso do método *shutdown()* da *threadpoll* para esta terminar, torna-se necessário. Se não for executado o *shutdown()* a *threadpool* continuará ativa e a consumir recursos. Um *ExecutorService* pode ser agendado para execução periódica ou a partir de um específico período de tempo de espera, utilizando a interface *ScheduledExecutorService*. O agendamento baseia-se na criação de *tasks* em diferentes instantes de tempo e no retorno de um objeto *task* que pode ser utilizado para cancelar ou verificar o estado da execução. [JavaSE 8, 2015]

2.4.3 *ThreadPoolExecutor*

No Java uma *threadpool* é uma coleção de *threads* com uma *BlockingQueue* partilhada onde as *tasks* são armazenadas. Estas são retiradas pelas *threads* e de seguida são executadas. As *threads* funcionam realizando pedidos de *tasks* para execução.

- O valor *corePoolSize*, referente à quantidade mínima de *threads* a manter na *threadpool*.
- O *maximumPoolSize*, é o valor máximo de *threads* que a *threadpool* é capaz de aceitar.
- O parâmetro *keepAliveTime*, é relativo ao tempo máximo que as threads inativas acima do valor *corePoolSize* continuaram na *pool* antes de serem removidas.
- A W*ork Queue* ou *Task Queue* é o local onde as *tasks* são armazenadas para futura execução.

Todo este sistema é possível ser analisado na seguinte ilustração [Figura 5].

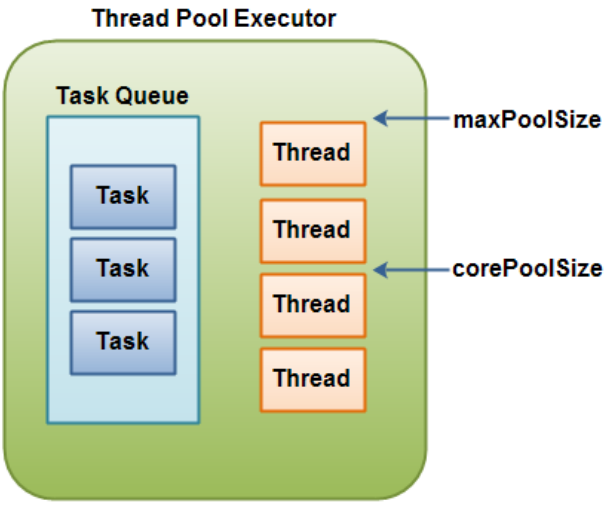

Figura 5 – Exemplo de um *ThreadPool Executor* [5]

A implementação de uma ThreadPoolExecutor é bastante útil em programas que utilizem paralelismo para tirar proveito de sistemas com processadores multicore. Ao utilizar o *multithreading* sem penalidades de performance, causadas por overhead na criação e partilha de informação nas threads.

Uma *ThreadPoolExecutor* contém um determinado número de *threads*. Este valor é estabelecido automaticamente de acordo com o número de núcleos disponíveis. Caso existam menos *threads* do que o *corePoolSize* então sempre que uma *task* é alocada na *threadpool* uma nova *thread* é criada. Se a *queue* de *tasks* estiver preenchida, e existam mais *threads* que o *corePoolSize* mas menos em execução que o maximumPoolSize, uma nova thread será produzida. Ao definir um *corePoolSize* igual ao *maximumPoolSize*, a *threadpool* criada será uma do tipo *FixedThreadPool*.

De maneira a diminuir o consumo de recursos, caso a *threadpool* contenha mais *threads* do que o valor *corePoolSize*, as *threads* em excesso são terminadas se estiverem inativas mais que do valor de *keepAliveTime*. Mais tarde, caso a *threadpool* se torne mais ativa, novas serão produzidas. É possível desligar esta política de poupança de recursos se desejado, utilizando o método *setKeepAliveTime* e enviando os parâmetros *Long.MAX_VALUE, TimeUnit.NANOSECONDS*. Assim é impedido que as *threads* sejam removidas antes do *shutdown* da *threadpoll*. [JavaSE 8, 2015] O tamanho ideal para uma *threadpool* depende do tipo de *tasks* que irão ser submetidas e das características do sistema, mesmo com este conhecimento, estabelecer a capacidade não é uma ciência exata.

[5] http://tutorials.jenkov.com/images/java-concurrency-utils/thread-pool-executor.png

A *Worker Queue* pode ser uma *BlockingQueue* de qualquer género, a seleção do tipo de *queuing* a implementar pode ser fundamentada nas seguintes estratégias:

- *Direct Handoffs*

  Esta estratégia é baseada na utilização de uma *Worker Queue* do tipo *SynchronousQueue*. Esta não tem qualquer capacidade de armazenamento de *tasks*, ou seja, cada operação de inserção na *queue* tem de ser imediatamente seguida por uma operação de remoção de parte de uma *thread*. Em vez do armazenamento de elementos de trabalho, esta mantêm uma lista de *threads* à espera de receber ou libertar um elemento. [Brian Goetz et al., 2006]

- *Unbounded queues*

  Nesta estratégia é implementada uma *queue* sem capacidade predefinida como exemplo uma *LinkedBlockingQueue*, garantindo que novas *tasks* são encaixadas na *queue* mas esta nunca conseguirá ser preenchida por completo. Esta estratégia faz com que não sejam criadas mais *threads* que o *corePoolSize* e que o parâmetro *maximumPoolSize* não tenha qualquer importância.

- Bounded *queues*

  Utilizar este tipo de *queues* como *ArrayBlockingQueue*, permite uma boa gestão de recursos e controlo. Isto é possível ao bloquear pedidos de inserção se a *queue* se encontrar completamente preenchida ou se for realizado um pedido de *tasks* e esta se encontrar vazia. Ao gerir cuidadosamente o tamanho da *threadpool* e da *Worker Queue* é possível obter melhorias na gestão de recursos. Se for implementada uma *Worker Queue* de uma grande dimensão mas uma pequena *threadpool* é possível minimizar o uso do processador e *overhead* gerado por *context-switching*, mas pode criar problemas de velocidade de resposta a pedidos. O uso de *Worker Queues* de pequenas dimensões em norma, leva a um uso de uma *threadpool* com uma dimensão considerável, o que mantém os núcleos bastante ocupados. [JavaSE 8, 2015]

2.4.4 *Futures*

Um *Future* representa o resultado de uma computação assíncrona. Esta é representada no Java através da classe *FutureTask*. A construção de uma *FutureTask* pode ser elaborada de duas maneiras, a mais utilizada é através do construtor *FutureTask(Callable<V> callable)*, em que ao ser iniciado irá executar a *task callable*. A segunda forma é utilizando *FutureTask(Runnable runnable, V result)* em que o segundo parâmetro será o retorno após a *task* ser completada com sucesso. Em circunstâncias que não seja necessário um resultado, pode-se contruir a *task* com o parâmetro *result* declarado a *null*. Uma vez que uma *FutureTask* pode ser produzida usando objetos do género não só *Callable* mas também *Runnable,* esta pode ser submetida para um *Executor*.

Ao utilizar Futures surge um problema, para tratar os resultados das tarefas é necessário invocar o método *get()* e este bloqueia a *thread* que o invoca até o resultado estar disponível. Tal fato é um inconveniente, uma vez que pode levar a atrasos do sistema sobretudo quando existe tratamento dos resultados. Para auxiliar nesta dificuldade o Java 8 foi incorporado com a classe *CompletableFuture<T>* que permite a substituição da invocação do método *get()* pela operação *thenApply()*. Este método permite passar por parâmetro uma operação que será executada assim que obtido o resultado do *Future* que invocou o *thenApply()*, retornando sem bloquear a *thread*. Os *CompletableFutures* permitem especificar quais operações assíncronas a realizar, e a ordem pelas quais são realizadas, de forma semelhante a uma linha de produção. [Cay S. Horstmann, 2014] Ao retorno do primeiro future é chamado *CompletionStage*, ou seja uma fase de uma possível computação assíncrona de *CompletableFutures* que, executa operações sobre o valor de retorno de uma outra CompletionStage. As operações a serem executadas numa fase podem ser funções expressas por expressões lambda, uma nova vantagem introduzida na API do Java 8.

Uma outra abordagem para operações de resultados não *null* de *Futures* é a implementação de um *CompletionService* com um *Executor*. Tal é possível criando um *CompletionService* com a interface *ExecutorCompletionService,* enviando um *Executor* por parâmetro para este, e submetendo os *Work Items* ao objeto *CompletionService*. Ao longo do tempo quando as *task* vão concluindo, estas são adicionadas a uma *queue* que apenas armazenará as *task* que já terminaram a sua função. A partir da invocação do método *take()* é possível conseguir obter o objeto *Future* relativo a uma *task* alocada na *queue*. Com análise deste funcionamento é possível afirmar que o *CompletionService* combina as funcionalidades do *Executor* e da *BlockingQueue* trazendo grandes benefícios ao simplificar a gestão de execução e conclusão das *tasks*. Estas vantagens podem ser verificadas segundo uma logica de programação considerando o seguinte caso. Se se submeter *n tasks* para o *ExecutorService* e se for necessário realizar o *shutdown* da *threadpool* para terminar o consumo de recursos, simplesmente é preciso percorrer um repositório onde as *task* produzidas tenham sido armazenadas e também invocar em cada uma destas o método *take()*. De maneira a aperfeiçoar o desempenho de forma ainda mais interessante, uma vez que o mecanismo aguarda pela terminação da *task*, pode-se reduzir o tempo de espera ao executar o trabalho sobre os resultados obtidos das *task*, que de fato já terminaram. Desta maneira possibilita-se que a duração de espera da próxima *task*, quando invocado o *take()*, seja muito inferior ao de invocação contínua.

2.4.5 Outros mecanismos de controlo de *tasks*

O estudo das vantagens do mecanismo *CompletionService* levou à tentativa de procura de opções semelhantes de como controlar o estado de conclusão das tarefas de maneira a ser possível tirar o mesmo proveito. Ao investigar implementações existentes na *internet* foi possível concluir que é necessário, na maioria dos casos, a introdução de uma variável de estado. Uma vez que se pretende determinar se uma *task* terminou o seu funcionamento, a implementação de uma variável do tipo booleano torna-se muito evidente e por isso foi estudado esse caso nos testes implementados. Outra possível implementação, que foi também investigada, consiste na utilização de objetos da biblioteca de concorrência. Ao estudar com maior rigor a API, foram analisados os objetos *CountDownLatch*. Estes têm o objetivo de auxiliar

a sincronização de *threads*, ao produzir uma estrutura de funcionamento de espera, em que a operação de uma *thread* aguarda pela conclusão de outra. Um *CountDownLatch* é inicializado com um valor a representar o contador, que obriga as *threads* a aguardar até este atingir o valor de zero. O contador é acedido através dos métodos *countDown()* e *getCount()*. O primeiro método decrementa a contagem e o segundo retorna o valor no contador. Combinando assim os dois métodos é possível implementar um sistema de análise de *tasks* atribuindo o valor ao contador de total de *tasks* criadas. Para ser possível decrementar o contador é necessário que o construtor das *tasks* permita a passagem do objeto CountDownLatch. Ao enviar o mesmo objeto para todas as *tasks* é possível que estas invoquem o método *countDown(),* assim que terminem a sua função, permitiundo que a *main thread* verifique com precisão o estado do contador.

2.4.6 *BlockingQueue* e subinterfaces

A *BlockingQueue* fornece os métodos *put*(), *take()* e os seus correspondentes *offer()* e *poll()*, que permitem definir uma *deadline* para finalizar a inserção ou remoção de elementos. Estes tipos de *queue* suportam o padrão *producer-consumer*, separando a inserção do trabalho realizada por produtores, da realização das tarefas pelos consumidores. Os produtores não necessitam de conhecer informações sobre os consumidores ou outros produtores, apenas que o seu trabalho é inserir elementos na *queue*. Os consumidores também não necessitam de conhecimento sobre os produtores e sobre a *queue* apenas necessitam de saber que é o local de onde removem elementos.

Uma *BlockingQueue* é uma *queue* que verifica as operações de inserção e remoção de elementos da mesma. Esta averiguação é realizada ao confirmar se esta se encontra cheia ou vazia bloqueando a thread, referente à operação, na confirmação da condição. [Brian Goetz et al., 2006] O funcionamento de uma *BlockingQueue* pode ser analisado na Figura 6.

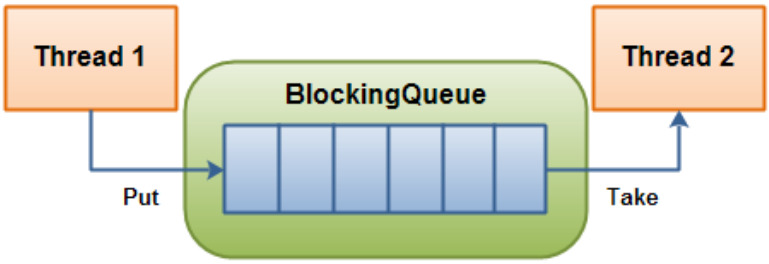

Figura 6 – Exemplo de *BloquingQueue* com duas *threads* a inserir e a remover elementos [6]

Uma *BlockingDeque* é uma *queue* do tipo *thread-safe* ao bloquear pedidos do tipo *put* ou *take* nas mesmas circunstâncias da *BlockingQueue* e não aceita elementos cujo valor seja *null*. Esta *queue* implementa uma estrutura *Double Ended Queue* ou *deque*, de uma forma simplificada, em que é possível executar as operações de inserção ou remoção de elementos da *queue* em ambos os seus extremos. A seguinte ilustração [Figura 7] representa o possível funcionamento de uma *BlockingDeque*.

[6] http://tutorials.jenkov.com/images/java-concurrency-utils/blocking-queue.png

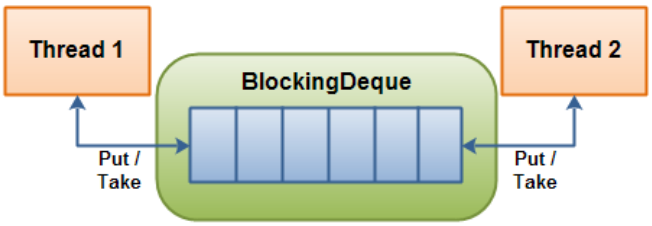

Figura 7 – Exemplo de funcionamento de uma *BlockingDeque* [7]

Uma vez que a interface *BlockingDeque* é uma extensão das classes *BlockingQueue* e *Deque*, os métodos de ambas as classes podem ser invocados por esta. Este fato é uma grande vantagem pois permite um conjunto de operações que possibilitam mover os elementos. As seguintes tabelas [Tabela 2 e Tabela 3] apresentam um resumo dos métodos fornecidos na *BlockingQueue* e os métodos adicionais na *BlockingDeque* para entender o melhor as diferenças no funcionamento de cada.

Tabela 2 – Resumo dos métodos das interfaces *BlockingQueue*

Métodos	**Levanta Exceção**	**Retorna valor**	**Bloqueia**	**Temporizador**
Inserção	add(e)	offer(e)	put(e)	offer(e,time,unit)
Remoção	remove( )	poll( )	take( )	poll(e,time,unit)
Examinar	element( )	peek( )	Inexistentes	Inexistentes

Tabela 3 – Resumo dos métodos das interfaces *BlockingDeque*

Métodos	**Levanta Exceção**	**Retorna valor**	**Bloqueia**	**Temporizador**
Inserção	addFirst(e), addLast(e)	offerFirst(e), offerLast(e)	putFirst(e), putLast(e)	offerFirst(e,time,unit), offerLast(e,time,unit)
Remoção	removeFirst( ), removeLast( )	pollFirst( ), pollLast( )	takeFirst( ), takeLast( )	pollFirst(e,time,unit), pollLast(e,time,unit)
Examinar	getFirst( ), getLast( )	peekFirst( ), peekLast( )	Inexistentes	Inexistentes

A *TransferQueue* é a segunda subinterface que faz extensão à *BlockingQueue*. Esta tem um conjunto de mecanismos do tipo *transfer*, para além dos métodos herdados da *BlockingQueue*. Os métodos *transfer* permitem enviar as tarefas diretamente dos produtores para os consumidores.

[7] http://tutorials.jenkov.com/images/java-concurrency-utils/blocking-deque.png

As bibliotecas das classes *BlockingQueue* e subinterfaces são implementadas por uma grande coleção de classes.

- *ArrayBlockingQueue*

  É uma *BlockingQueue* do tipo *bounded*, *array* de tamanho fixo, que ordena os seus elementos utilizando a logica FIFO (*first-in-first-out*). O elemento inserido há mais tempo está na cabeça da *queue* e o mais recente na *queue* localiza-se na cauda. As operações de inserção na *queue* colocam os elementos na cauda e operações de remoção vão recolher os elementos da cabeça. Quando criada, o seu tamanho não pode ser alterado e a tentativa de inserção de elementos, quando cheia, irá resultar num bloqueio. O bloqueio pode também acontecer na invocação da operação *take*, se a *queue* estiver vazia no momento de execução do método. A ordem dos pedidos *producer/consumer* a este tipo de *queue* por norma não existe. No entanto, a atribuição de *true* ao atributo *fair*, na construção da *queue*, garante que os pedidos serão efetuados utilizando a lógica FIFO. Para além de criar um *array* com um tamanho fixo e com uma política de ordenação justa a pedidos, também é possível iniciar a *BlockingQueue* com uma coleção de tarefas utilizando o construtor, *ArrayBlockingQueue(int capacity, boolean fair, Collection <? extends E> c)*.

- *DelayQueue*

  É uma *BlockingQueue* que armazena elementos do tipo *Delayed*. Quando inserido um elemento, este só poderá ser retirado da *queue* depois do seu *delay* expirar. A cabeça da *queue* possui o elemento cujo *delay* expirou há mais tempo. No caso de nenhum dos elementos terem alcançado o seu tempo de expectação, a cabeça da *queue* estará vazia e pedidos do tipo *poll* e *peek* irão retornar *null*. No caso dos pedidos do tipo take, será realizada uma espera até que um dos elementos consiga ultrapassar o seu *delay* para ser possível existir retorno. Este género de *BlockingQueue* permite a sua construção com uma listagem de tarefas, através do envio de uma coleção de instâncias do tipo *Delayed* como parâmetro.

- *LinkedBlockingDeque*

  É uma *BlockingDeque* baseada em *linked nodes* com uma capacidade fixa para prevenir aumentos desnecessários do tamanho da *queue*. A capacidade pode ser especificada por parâmetro na criação desta *queue*. No caso de nenhum parâmetro de capacidade ser passado no construtor, a capacidade será igual ao valor *Integer.Max_VALUE*.

- *LinkedBlockingQueue*

  Estas são *queues* muito semelhantes às *LinkedBlockingDeque* mas implementam a filosofia FIFO das *queues*.

- *LinkedTransfereQueue*

  É um *queue unbound* fundamentada em *linked nodes* que ordena os seus elementos através da logica FIFO. Esta implementa os mecanismos das *TransferQueue*.

- *PriorityBlockingQueue*

  É um género de *BlockingQueue* de dimensão não estabelecida que contêm regras de acordo com as estabelecidas nas *PriorityQueue*. Os elementos a inserir requerem a implementação da interface *Comparable*. Esta necessidade existe pois no momento da inserção de um elemento, deve ser aplicado o método *compareTo()* de maneira a determinar a sua colocação na *queue*.

- *SynchronousQueue*

  É um tipo de *BlockingQueue* que não armazena elementos pois não contêm qualquer capacidade interna. O modo de funcionamento deste tipo de *queues* é baseado em esperas de ações onde uma operação de inserção deve esperar por uma operação de remoção e vice-versa. Uma vez que esta não tem capacidade de armazenamento, a utilização do método *peek()*, operações de remoção ou tentativas de obter o tamanho utilizado ou capacidade total da *queue*, irão retornar sempre os mesmos valores de *false*, *null* ou zero consoante o método invocado. [JavaSE 8, 2015]

2.4.7 *Fork/Join*

A *framework Fork/Join* é um *ExecutorService* que permite executar *ForkJoinTasks* e implementa a estratégia *work-stealing*. Esta estratégia indica que, núcleos do processador com uma *WorkerQueue* vazia têm a iniciativa de tentar roubar trabalho a outros núcleos. Desta maneira é possível usufruir o máximo do tempo de execução. Esta *framework* foi desenvolvida para resolver problemas dividindo o trabalho criando *ForkJoinTasks*. O problema a resolver numa *ForkJoinTask* pode ser dividido de forma a ser realizado por ainda mais *tasks* (*divide-and-conquer*). O princípio de divisão em *tasks* é possível através da implementação recursiva do método *fork()*, podendo este ser executado concorrentemente em que cada *task*. Este procedimento de separação produz *overhead* que pode ser superior aos benefícios da execução das *tasks* paralelamente. Quando uma *task* se divide, esta necessita de aguardar pela conclusão das novas *tasks* geradas por si. Esta espera é possível através da utilização do método *join()*. [Javier Fernández González, 2012]

A Figura 8 apresenta a ideia implementada no *Fork/Join*, ilustrando o sistema *divide-and-conquer* e os mecanismos de sincronização.

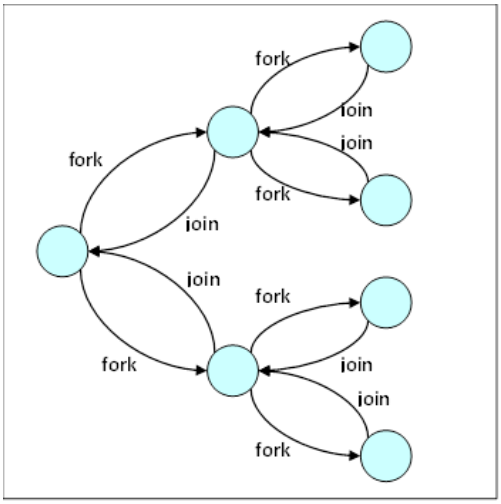

Figura 8 – Exemplo do *Fork/Join* [8]

A implementação do *Fork/Join* implica uma operação de paragem para a criação de *tasks* para que o problema pare de ser subdivido e comece a ser resolvido e o resultado da *task* seja retornado para o responsável da sua criação.

As *ForkJoinPools* permitem a execução de *tasks* do tipo *Runnable*, *Callable* e *ForkJoinTask*. Este último tipo de *task* é conseguido através da produção de *tasks* que realizem a extensão a uma das subclasses existentes da classe abstrata *ForkJoinTask*. A classe *ForkJoinTask* tem três tipos de subclasses disponíveis:

- *CountedCompleter*

  É uma *ForkJoinTask* com uma ação a realizar com os resultados de uma ou várias tarefas em que, é possível estabelecer um contador na criação das *tasks* que, através da invocação do método *tryComplete()* irá decrementar esse contador até atingir zero. Assim que atingir o valor zero é efetuada a próxima etapa, cada etapa pode ser considera um vértice na estrutura *Fork/Join* que aguarda que as *tasks* que criou terminem.

- *RecursiveAction* e *RecursiveTask<R>*

  São ambas *tasks* do tipo de *ForkJoinTask* mas, a RecursiveAction impossibilita que seja possível obter um resultado de retorno. A *ForkJoinTask*<R> garante o retorno de um resultado no final da execução, em que *R* é o tipo de resultado produzido.

[8] http://www.developer.com/imagesvr_ce/3378/join-fork-image001.png

A natureza do *Fork/Join* no Java é uma variante adaptada do escalonador de *work-stealing* do Cilk de acordo com o criador Doug Lea. O *work-stealing* introduz uma grande vantagem uma vez que permite reduzir o conflito entre *tasks*, devido às *threads* terem permissão para processarem as suas tarefas e roubar *task* delegadas a outras *threads*. As regras introduzidas no *work-stealing* são:

- Cada CPU tem uma *thread* com a sua própria *deque*;
- Deques permitem operações em ambos extremos desta, LIFO e FIFO em simultâneo.
- As tarefas criadas por outras *tasks* são colocadas na *Worker Deque* da sua *thread* correspondente;
- As *threads* processam as tarefas das suas *deques* pela ordem LIFO;
- Quando uma *thread* não tem tarefas para executar, esta rouba tarefas da *Worker Deque* de outras *threads*;
- Quando não existe trabalho a ser executado a *thread* fica inativa até novas tarefas serem introduzidas. [Cláudio Maia, et al., 2011]

3 Ambiente de testes

O capítulo atual interpela os procedimentos e o ambiente gerado para análise das ferramentas de concorrência e as explicações necessárias. Este aborda também a estratégia elaborada para a comparação de performance e os requisitos de memória. São também apresentadas as explicações relativamente aos ajustes realizados nas máquinas onde o código é executado, a utilização de *software* para o *profiling* dos mecanismos e como os dados são recolhidos assim como que operações são executadas para o seu estudo.

Este capítulo é a base de apoio de informações para melhor entender as implementações que foram testadas de maneira a existir um melhor entendimento das futuras conclusões. Com este estudo inicial espera-se que, a partir dos testes realizados, seja possível a realização de um estudo de comparação entre os diferentes mecanismos de concorrência. Para a elaboração do estudo são empregando os mecanismos em diversos algoritmos com a mesma finalidade e por fim executá-los em diferentes máquinas. Este processo serve para garantir a obtenção de conclusões sobre os mecanismos tendo conhecimento do impacto do *hardware*.

3.1 Estrutura de testes

Elaboraram-se os testes para produzir um conjunto de conclusões sobre os mecanismos de concorrência e a sua implementação, e não sobre qual o melhor algoritmo a ser implementado. Com este pensamento em mente foram desenvolvidos um conjunto de testes intensivos de forma a ser possível demonstrar os tempos de execução e consumos de memória de diversos algoritmos com implementações sequenciais e concorrentes.

Para completar o objetivo inicial de obter resultados de diferentes algoritmos que facultassem um fim comum, optou-se pela implementação de algoritmos que permitissem ordenação ascendente. Do estudo realizado, vários algoritmos foram pensados para fundamentar esta dissertação mas devido à semelhança ou incapacidade de preencher por completo os objetivos desejados apenas foram escolhidos os seguintes:

- Quicksort

  O algoritmo *Quicksort* foi escolhido para análise pois é das implementações mais comuns e conhecidas. Este opta por fazer uma divisão da estrutura de dados baseando-se em pivôs. [Figura 9] O cálculo dos pivôs irá representar o processo de divisão de trabalho a ser atribuído às *threads* e *tasks*. Ao utilizar os pivôs nunca se garante que a divisão seja equilibrada. Como ponto a favor, durante o cálculo do pivô existe uma separação de valores, o que facilita a ordenação. O pivô nesta ordenação funciona como o nome indica, de intermediário enviando valores inferiores a ele próprio para posições menores no *array* e valores superiores para posições mais elevadas no *array*.

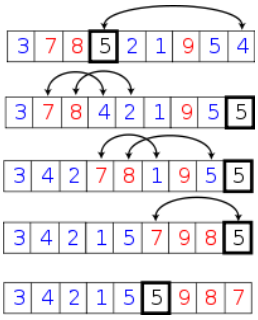

Figura 9 – Exemplo de particionamento no algoritmo *Quicksort* [9]

- Mergesort

  É um algoritmo de ordenação que se baseia na divisão de problemas em problemas de menor dimensão e na junção de todas as soluções obtidas de forma a chegar à solução desejada (*Divide-and-Conquer*). [Figura 10] Este algoritmo, ao contrário do *Quicksort*, faz uma divisão o mais equilibrada possível dos problemas mas com a desvantagem de criar cópias dos dados para a realização da divisão dos problemas. Devido a este prejuízo é necessitando de um maior consumo de recursos de memória.

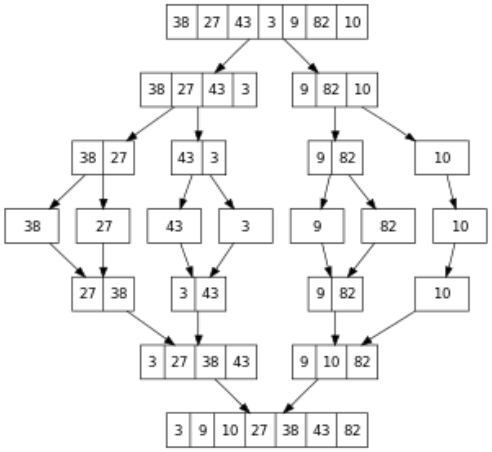

Figura 10 – Exemplo de *Divide-and-Conquer* [10]

[9] https://upload.wikimedia.org/wikipedia/commons/thumb/8/84/Partition_example.svg/200px-Partition_example.svg.png
[10] https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Merge_sort_algorithm_diagram.svg/300px-Merge_sort_algorithm_diagram.svg.png

- Pidgeonholesort

  É um algoritmo que se baseia no *pigeonhole principle*. Este algoritmo consiste em se colocar um número *n* de objetos em *m* repositórios. Caso o valor de *n* seja um valor superior a *m* então pelo menos um dos repositórios terá de conter mais do que um objeto. Esta explicação convertida em lógica de programação, indica que é necessário um repositório para armazenar a quantidade de chaves referentes a cada fechadura. Isto convertido em linguagem de programação indica que é necessário um repositório para atribuir chaves a valores. A Figura 11 será usada como exemplo desta explicação.

  Para ser possível a ordenação, o passo mais importante é o cálculo do maior e do menor valor presentes no contentor. De seguida, é elaborada a criação de um *array* "de pombos". Nos "buracos" é colocada a quantidade de "pombos" que corresponde a cada chave. Como ilustrado na Figura 11, o maior valor é 5 e o menor é 2, então foi criado um *array* cuja dimensão é igual ao valor do maior elemento menos o valor do menor. No final obtém-se um *array* que contém três "pombos" na posição chave dois, um na posição da chave três, zero para a chave quatro e dois pombos a representar os valores da chave cinco. Por fim é elaborado um *array* com os dados obtidos do *array* "pombos".

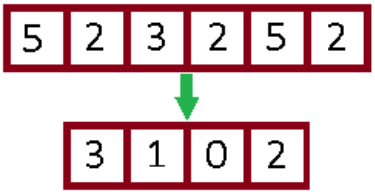

Figura 11 – Exemplo de criação de *array* de "pombos"

Foram preparados outros algoritmos de ordenação para se realizarem testes mas devido à incapacidade destes serem implementados com a utilização de mecanismos de concorrência, os algoritmos foram então ignorados neste estudo. Algoritmos com outros objetivos também passaram este processo e foram ignorados por serem insignificantes perante os casos de análise de ordenação.

Cada um destes algoritmos de ordenação foi implementado com os mecanismos da API Java referidos na secção 2.4. Estas implementações foram realizadas de forma a ser possível retirar conclusões relativamente a, quais os mecanismos presentes na atual versão do Java, que permitem tirar vantagem dos sistemas com múltiplo processamento. Para se conseguir elaborar uma análise mais aprofundada foi necessário decidir uma estrutura de armazenamento de dados e uma quantidade de testes a executar. Como repositório de valores para os diversos métodos de ordenação foi escolhida a utilização de *arrays* que contivessem valores do tipo inteiros. Os tamanhos estabelecidos para os mesmos foram mil, cinco mil, dez mil, cinquenta mil e cem mil. Estes valores foram estabelecidos com base no consumo de memória e na quantidade de tempo disponível nas máquinas de execução. Estes *arrays* foram preenchidos em cada situação de diferentes dimensões utilizando o mecanismo de produção de valores aleatórios que a API do Java 8 oferece. Foram definidos os limites inferiores e superiores da geração de valores com base na dimensão em cada instante de nova dimensão. Assim é possível manter um

nível de complexidade para o algoritmo de ordenação independente da dimensão do *array* no momento.

```
public static int[] FillRandom(int Tam) {
        int arr[] = new int[Tam];
        for (int i = 0; i < Tam; i++) {
            arr[i] = ThreadLocalRandom.current().nextInt(0-Tam/20, Tam/20);
        }
        return arr;
}
```

Código 1- Preenchimento de *arrays*

Utilizando o método ilustrado no Código 1 é possível criar um repositório com um conjunto de valores positivos ou negativos que podem assumir valores de maneira a que no mínimo 10% do *array* esteja preenchido com valores repetidos. Todo este processo de decisão de tamanhos, definição de parâmetros mínimos e máximos para sorteio de valores para o *array* e quantidade de testes a elaborar tiveram também em conta o tempo de execução que os testes demoram.

Para se determinar a dimensão da amostra de testes ao qual se pretende efetuar as operações matemáticas para criação das conclusões, foi necessário entender que os algoritmos criados funcionam sobre um sistema, *hardware* e *software*, que cria obstáculos para obter sempre os mesmos valores. Como primeiro passo para obter este valor, optou-se por analisar o que acontece ao efetuar múltiplos testes aos algoritmos implementados. Este processo levou à conclusão que, independentemente do algoritmo, o tempo de execução até à décima execução é muito mais elevado do que os restantes a partir desse número de testes. A Figura 12 seguinte ilustra os valores de um dos exemplos obtidos que prova esta conclusão.

ArraySort 1000 posicoes!	
NO TESTE	Tempo(Microseg)
0	991
1	1870
2	518
3	512
4	503
5	497
6	471
7	121
8	515
9	126
10	115
11	119
12	116
13	122
14	122
15	110

Figura 12 – Exemplo de problema de elevadas durações nas primeiras execuções

Estes valores provocam um impacto significativo na análise dos dados e para se conseguir criar uma métrica justa foi impossível utilizar a fórmula:

$$n = \left(\frac{Z\alpha/2^{\sigma}}{E}\right) \quad (1)$$

, onde n é o tamanho da amostra, $Z\alpha$ é o valor critico correspondente ao grau de confiança desejado, σ é o valor do desvio padrão e o valor de E é a margem de erro. Uma vez que não se consegue atribuir um valor aceitável ao analisar os valores obtidos como a média, mediana e margem de erro, de diferentes tamanhos de amostragem, optou-se por um valor de sessenta testes para cada caso de estudo. Esta decisão deveu-se ao valor de margem a partir desse número de testes não diminuir tão significativamente. Uma vez escolhido o valor de amostragem, são realizados um número de testes, igual ao valor da amostra, em cada nível escolhido de dimensão de *array*. De forma a arquivar os dados resultantes de cada teste foi elaborada uma estrutura de armazenamento de resultados num ficheiro do tipo ".csv" para permitir uma análise mais simples e organizada.

Para este estudo foi criado um projeto no IDE Netbens em que, para cada algoritmo a ser testado, foi criado um package e neste foram armazenadas as classes com a lógica necessária para o estudo dos mecanismos de concorrência. De maneira a provar a eficiência dos mecanismos de concorrência em diferentes sistemas foram escolhidas três máquinas. Estas máquinas diferem no número de processadores lógicos, uma com dois, outra com quatro e por fim uma com oito, de forma a demonstrar os diferentes tempos de processamento dos algoritmos de ordenação. Nestas máquinas foram executados todos os testes elaborados de algoritmo de ordenação com os diferentes mecanismos.

3.1.1 Mecanismos implementados

Com a análise dos algoritmos de ordenação escolhidos e dos mecanismos disponíveis referidos no capítulo 2.4, foi criado o plano de desenvolvimento dos mecanismos a utilizar. Dos mecanismos estudados foram implementados os seguintes em cada um dos algoritmos:

- *java.lang.Thread*, a escolha da implementação deste mecanismo foi tomada, pois em sistemas *multithread* é importante conhecer a quantidade de *threads* que se deseja criar para a solução de um problema. A abordagem geral na construção de algoritmos é criar um número de *threads* igual ao número de processadores lógicos presentes na máquina ou esse valor mais um.

- *ExecutorService* com três diferentes *threadpools*: *newWorkStealingPool*, *newFixedThreadPool* e *newCachedThreadPool*. Em cada um dos casos foi abordada a construção da *threadpool* com um número de *threads* igual ao resultado da análise do estudo *java.lang.Thread*. Outros pontos importantes a ter em consideração é, uma vez que se utilizou *threadpools*, é saber a quantidade de *tasks* a submeter e como obter o estado de conclusão das destas. Para estas questões foram desenvolvidas algumas soluções. Para determinar a quantidade de *tasks* a produzir foram estabelecidos os seguintes valores de produção de *tasks*: dois, quatro e oito *tasks* em sistemas de dois núcleos físicos e dois, quatro, oito, dezasseis e trinta e dois para quatro núcleos. É possível obter o número de processadores lógicos presentes na máquina foi possível ser

obtido através do mecanismo *Runtime* da API do Java. A declaração total pode ser analisada no seguinte estrado de código.

```
ThreadLvl = Runtime.getRuntime().availableProcessors();
```

Código 2 – Nível de Paralelismo

Para resolver a questão de verificação do estado da *task* para se poder concluir o processo, foram elaborados três novos casos de estudo em que cada um reflete um processo diferente de obtenção do estado da *task*. Os processos de aquisição do ponto de situação implementados foram os seguintes: o uso de uma variável do tipo booleano que representa o estado de conclusão da tarefa e o uso dos mecanismos *ExecutorCompletionService* e *CountDownLatch* separadamente.

Assim, criou-se o esquema de testes representado na seguinte Figura 13.

Analise de Tempos										
Sistema		CountdownLach			CompletionService			Boolean		
ThreadPool		Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
Tamanho	Nworkers									
1mil	2									
	4									
	8									
5mil	2									
	4									
	8									
10mil	2									
	4									
	8									
50mil	2									
	4									
	8									
100mil	2									
	4									
	8									

Figura 13 – Template de preenchimento de resultados dos testes ao *ExecutorService*

- A *ForkJoinPool* é uma *threadpool* que permite tarefas do tipo *RecursiveAction* ou *RecursiveTask* e funciona utilizando a estrutura *Divide-and-Conquer*. A *ForkJoinPool* ao contrário das *threadpools* utilizadas no *ExecutorService,* esta não necessita da gestão de estado das *tasks*. Ao realizar a divisão de trabalho é necessário cuidado ao definir um ponto de paragem para a divisão que dá inicio à computação das soluções. Este ponto é conhecido como *threshold*. No projeto é utilizado um *array* como repositório e o *threshold* é a dimensão máxima que o *array* pode deter até interromper a sua divisão e iniciar a ordenação. De forma a definir um *threshold* no projeto, foi criado um sistema com o objetivo de rentabilizar o máximo a velocidade de acesso aos dados já armazenados na cache. Para calcular o *threshold* na estrutura Fork/Join é realizado o seguinte cálculo:

$$n = \left(\frac{L1 * 1024 * 1024 * 8}{SO * 0.02}\right) \qquad (2)$$

Esta fórmula é composta pelos valores da cache da máquina ($L1$) em Megabytes e pela versão do sistema (SO) que representada em bits quanto um *integer* consome de memória.

3.1.2 Métricas de análise

Como referido no capítulo 3.1, foram analisados valores como a média e margem de erro. Para além destes valores, outros foram calculados e as suas métricas foram necessárias estipular para ser possível a análise e comparação dos dados.

Uma vez que o objetivo final é produzir uma análise aprofundada dos mecanismos de concorrência da API Java 8, foram selecionados os consumos de tempo e memória como ponto de comparação entre resultados de testes. Para os custos de tempo foi estabelecida a métrica microssegundo (μs), esta unidade de tempo permite obter valores entre as centenas e as dezenas de milhares. Esta quantidade de dígitos foi considerada apropriada para a análise pois, não é uma quantia muito reduzida de dígitos, permitindo obter diferenças significativas, e não é também muito elevada, possibilitando a obtenção de resultados irrelevantes. Os valores com esta unidade de tempo, são conseguidos através da invocação do método *System.nanoTime()*, já existente no JDK do Java 8, e a passagem deste como parâmetro para o método *convertNanoToMicro()*, criado para este projeto, onde é dividido o parâmetro por mil. Aplicando este método no instante prévio ao algoritmo e no momento de conclusão do mesmo, é possível obter o total de microsegundos que o algoritmo necessitou para completar através da subtração de ambos instantes de tempo.

```
start = Templates.convertNanoToMicro(System.nanoTime());
//... Código a ser temporizado ...
tempo = Templates.convertNanoToMicro(System.nanoTime()) - start;
```

Código 3 – Temporização de Código

Existem várias opções para cálculo de tempos de algoritmos, desde a utilização de ferramentas comerciais de *profiling* até a invocação de ferramentas da API. Optou-se pela utilização do mecanismo da API pela facilidade e simplicidade ao aproveitar uma ferramenta já disponível para ser utilizada.

De forma a se efetuar uma análise dos dados de memória foi designado o megabyte (MB) como unidade de medida. Os dados de consumo de memória de cada código são muito complicados de obter através do uso dos métodos da API pois, o pico de consumo pode ocorrer durante a execução ou não no momento de obtenção do dado. Esta dificuldade levou à análise de ferramentas externas ao JDK. Existe um grande conjunto de ferramentas que permitem o *profiling* da execução de código como, o JProfille, o YourKit e o Java VisualVM. Para determinar o mais apropriado para o estudo foram comparadas as três ferramentas. O VisualVM é o mais limitado quanto à quantia de opções de análise e incorpora uma interface menos intuitiva mas tem a vantagem de ser sem custos monetários. As duas ferramentas são versões comerciais com opção de obtenção de uma chave para um período de teste e avaliação que garante o acesso temporário às versões finais das ferramentas. Entre estas duas foi escolhida a utilização da ferramenta do JProfille pela sua organização na disposição das opções como a análise de dados de memória, gestão de *threads*, contagem das invocações dos métodos e visualização da performance do CPU. Com a ferramenta JProfiler foi possível elaborar a exportação dos dados para ficheiros do formato CSV e HTML, tornando possível a análise do consumo de memória. Ao verificar os primeiros resultados, foi possível averiguar que os consumos não são totalmente

incrementais ao longo da execução assim, decidiu-se utilizar o valor máximo de consumo como representante de cada análise.

Como referido no capítulo 3.1, em determinadas circunstâncias os valores de tempo podem ser superiores ao caso normal, causando aumentos na média final. Devido a este aumento foi utilizado o valor de mediana em simultâneo com a média assim, conseguiu-se alcançar o valor mais típico da duração de execução dos algoritmos [Figura 14]. Uma vez que a amostragem é representada por um número par, a mediana é calculada, depois de ordenados os resultados ascendentemente, pela média dos dois valores intermédios (resultado do teste número trinta e trinta e um).

QuickSort Sequencial					
Tamanho:	Mediama	Media:	Margem	ValorMax:	ValorMin
1000	137	159	32,89	126,11	191,89
5000	495	543	39,73	503,27	582,73
10000	1064	1077	8,6	1068,4	1085,6
50000	6211	6245	30,62	6214,38	6275,62
100000	13301	13790	407,64	13382,36	14197,64

Figura 14 – Exemplo de dados obtidos da execução de um conjunto de testes

Para uma análise de resultados dos mecanismos foram utilizados cálculos estatísticos como o cálculo do intervalo de confiança (iC). O iC permite criar uma estimativa de intervalos de tempos que um algoritmo, com um conjunto de mecanismos específicos, pode necessitar para terminar. Este intervalo é calculado em conjunto com o desvio padrão (σ), a media e o coeficiente de confiança ($Z\alpha/2$) foi definido com o valor de 95%. Através do uso da seguinte fórmula é possível obter o intervalo de confiança:

$$\bar{x} \pm Z\alpha/2 * \left(\frac{\sigma}{\sqrt{(n)}}\right) \qquad (2)$$

, onde $\bar{x}$ é a média e ao adicionar-se o resultado da expressão referente à margem de erro é retornado o valor máximo e ao subtrair-se é retornado o valor mínimo do intervalo de confiança.

No decorrer do estudo foram executados um conjunto de testes onde foi elaborada uma comparação de tempos relativos a um possível caso de execução do algoritmo durante uma hora. Com esta abordagem é possível realçar os possíveis ganhos ou perdas em sistemas empresariais, demonstrando a diferença em minutos entre o caso base e as diferentes implementações. O caso base é determinado do conjunto global de implementações como o mais comum de ser aplicado, tendo em conta fatores como simplicidade e falta de conhecimento da API.

3.2 Verificação de Algoritmos

De forma a verificar se os diferentes algoritmos ordenam corretamente os valores, foram elaborados dois testes. Estes métodos de verificação encontram-se no *package Super* na classe *Templates* do projeto de maneira a facilitar a sua invocação pelos múltiplos algoritmos. Para a execução destes foi estabelecido um tamanho estático de dez mil posições para os contentores.

Todos os métodos de ordenação analisados foram submetidos a estas verificações e em todas as situações obtiveram sucesso.

Os testes de verificação criados foram os seguintes:

- Verificação de ordenação, onde verifica-se se os *arrays* no seu estado final se encontram devidamente ordenados. Na circunstância de não se encontrarem ordenados, indica as posições onde as falhas foram encontradas. Esta verificação permitiu apoiar a correção de erros nos algoritmos através da indicação dos locais onde a ordenação falhou. Este fato é relevante uma vez que podem ocorrer erros na divisão do repositório de dados ou problemas na fase final na união de todos os resultados parciais.

```
public static boolean CheckOrder(int arr[]) {
        boolean flag = true;
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] > arr[j]) {
                    System.out.println("Log: Not Order at: " + i + "
because its >" + j);
                    flag = false;
                }
            }
        }
        return flag;
    }
```

Código 4 – Verificação final de ordenação

- Verificações da existência de todos os elementos produzidos antes da ordenação no *array* final. Com esta verificação é possível garantir que todos os valores originais existem no *array* depois execução da ordenação e permitiu depurar possíveis falhar nos métodos de ordenação e na divisão de problemas. Os valores iniciais são armazenados criando um clone do repositório previamente à ordenação. Uma vez que múltiplas tarefas são executadas sobre o mesmo espaço de memória problemas como a duplicação de dados ou falhas no preenchimento do contentor final podem ocorrer. No caso de ocorrências de erros são listadas as seguintes informações, a identificação do método o número de *tasks* ou *threads* que estavam a ser utilizados. Esta indicação permitiu significativamente a detenção de erros que surgiam na divisão de trabalho, uma vez que a divisão de trabalho não é equilibrada garantidamente em todos os casos, devido ao tamanho do *array* e/ou ao algoritmo implementado.

```
public static boolean CheckElements(int[] notO, int[] order) {
        boolean check = false;
        List<Integer> values = new ArrayList<Integer>();
        for (int r = 0; r < notO.length; r++) {
            values.add(notO[r]);
        }
        for (int i = 0; i < order.length; i++) {
            check = false;
            for (int r = 0; r < values.size(); r++) {
                if (order[i] == values.get(r)) {
                    values.remove(values.get(r));
                    check = true;
                    break;
                }
```

```
                }
                if (check == false) {
                    return check;
                }
            }
            return check;
        }
```

Código 5 – Verificação de totalidade de elementos

Ambos os testes foram essenciais no decorrer do estudo pois permitiram retirar quais queres contestações sobre a eficácia das implementações concebidas. Com este fato as experiências garantiram o sucesso necessário para alcançar o objetivo traçado de ordenação correta, independente do mecanismo de concorrência implementado e do algoritmo. De forma a possibilitar a verificação rápida das ordenações foram produzidos executáveis para cada algoritmo de ordenação, assim é possível verificar os métodos rapidamente e com maior facilidade nas múltiplas máquinas.

3.3 *Hardware* e *Sotware*

De modo a provar o sucesso dos resultados dos algoritmos foram realizados os testes em diferentes máquinas. O principal fator que se procurou diferenciar entre estas foi a quantidade de núcleos físicos, a dimensão da *cache* e a presença ou não de tecnologias como o *Hyper-Threading* e *Turbo Boost* nos processadores. Fatores como diferentes Sistemas Operativos e diferentes versões deste foi ignorado, devido à dificuldade de obtenção de grandes quantidades de máquinas.

Durante o desenvolvimento desta dissertação foram adquiridas três máquinas para testar as ferramentas de concorrência. Estas máquinas diferem em vários fatores de *hardware* e *software* mas o principal foco em que se prestou mais atenção foi nas especificações dos CPUs.

3.3.1 Processadores

Os processadores são compostos um conjunto de especificações que influenciam a duração integral da execução do código, o que torna incomparáveis os resultados entre diferentes processadores. Sendo assim as análises nos futuros capítulos serão fundamentadas maioritariamente entre os mecanismos, desenvolvendo possíveis conclusões para cada processador. No decorrer do estudo foi possível testar os algoritmos produzidos nos seguintes processadores. [Tabela 4]

Tabela 4 – Listagem de processadores e suas especificações

Marca	Modelo	Frequência Base	Número de Cores	Cache	Tecnologias
Intel® Core™	T4300	2,1GHz	2	1MB	Nenhumas
Intel® Core™	i5-4260U	1,4GHz	2	3MB	Hyper-Threading Turbo Boost (2.8GHz)
Intel® Core™	i7-3610QM	2,3GHz	4	6MB	Hyper-Threading Turbo Boost (2.8GHz)

Com a aquisição dos processadores foi possível produzir análises com dados tendo em consideração os seguintes aspetos:

- O Dual-Core e o i5 têm o mesmo número de cores físicos mas tecnologias e tamanho de *cache* que diferentes. Devido ao *Hyper-Threading* apesar de o número de cores físicos ser igual o i5 consegue executar o dobro das *threads* em simultâneo.
- O i5 e o i7 incorporam as mesmas tecnologias mas a dimensão da *cache* e número de cores são diferentes

O estudo da *cache* foi considerado importante, pois esta funciona como repositório de dados entre o processador e a memória do sistema (RAM) como ilustrado na Figura 15. A *cache* armazena e cópia os dados utilizados com maior frequência de modo a reduzir o tempo de acesso aos dados. Quando é necessário obter informação esta é pesquisada na cache e no caso de não ser encontrada é de seguida e pesquisada na RAM. Após ser encontrada a informação é devolvida ao CPU e este a armazena na Cache. No acesso à *cache* existem duas terminologias, *cache hit* e *cache miss*, o *cache hit* refere-se ao acesso com sucesso aos dados de parte de um pedido. O termo *cache miss* é utilizado para expor o insucesso ao tentar encontrar os dados. Quanto maior for a dimensão da cache menor a probabilidade de acontecer um *cache miss*, mas a duração dos pedidos será superior devido à duração das pesquisa dos dados.

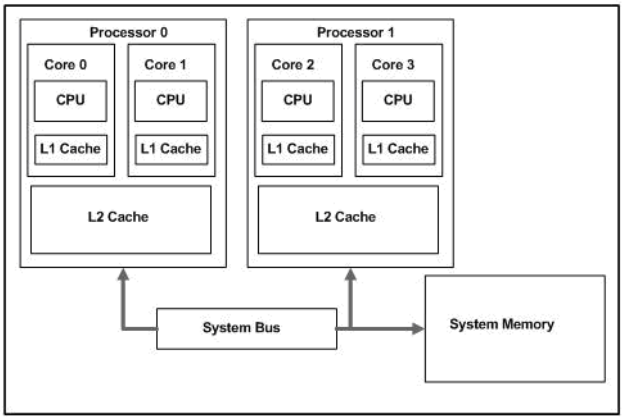

Figura 15 – Funcionamento entre CPU, Cache e RAM [11]

[11] https://software.intel.com/sites/default/files/m/d/4/1/d/8/286501_286501.gif

Todos os processadores mencionados possibilitam a utilização de arquiteturas 64 bits, possibilitando um endereçamento superior a 4 GB [Tabela 5], tanto de memória física como virtual. As principais diferenças entre os processadores em questão são a frequência base sobre o qual trabalham, o número de cores lógicos, a dimensão da cache e o funcionamento desta. Uma diferença que deve ser referida no funcionamento da cache acontece entre o Modelo T4300 que usa tecnologia L2 Cache que proporciona um ambiente de acesso *multicore* e os modelos i5-460M e i7-3610QM que utilizam Intel Smart Cache, que é uma versão L2 Cache que dinamicamente limpa a memória da cache em base requisição dos dados e dos períodos de inatividade.

Tabela 5 – Especificação dos CPUs e da memória presente nos computadores

Marca	Modelo	RAM
Intel® Core™	T4300	4GB
Intel® Core™	i5-460M	4GB
Intel® Core™	i7-3610QM	6GB

3.3.2 Java Virtual Machine

Com o decorrer dos testes devido ao consumo de memória dos algoritmos e dos mecanismos implementados foi necessário aumentar a dimensão da *heap*. A *heap* é um repositório dinâmico de dados, para quaisquer instâncias de dados produzidos da definição de classe s durante a execução da máquina virtual. Quando a *cache* encontra-se cheia é limpa pelo mecanismo *garbage collector*, o qual liberta espaço removendo objetos determinados como dispensáveis. Estes são determinados através de um sistema que sinaliza objetos como vivos, ainda têm utilidade, todos objetos não vivos são considerados dispensáveis. Todos objetos alocados na *heap* são geridos pela JVM. O *garbage collector* tem bastante impacto no sistema no decorrer da limpeza dos dados e pode conduzir a retardamentos de processos. Para prevenir esta circunstância, a programação efetuada foi concretizada com eficiência, ao não produzir objetos desnecessários.

Para aumentar a dimensão da *heap* foi realizada a inserção do argumento -Xmx4000m nas opções de execução do projeto, estabelecendo assim a dimensão da *heap* em 4000 MB.

3.3.3 Precauções tomadas

De modo a conceber um estudo o máximo possível credível e indisputável, procurou se criar um ambiente de execução realístico mas simples. Para produzir o ambiente desejado foram efetuadas algumas alterações na ferramenta de análise JProfiler. Uma vez que os portáteis utilizados pertencem a utilizadores comuns, é normal que existam serviços e processos a trabalhar em background que podem produzir interrupções e atrasos nos tempos. De modo a tentar garantir um ambiente estável foi instalado o programa AVG PC TuneUp. Este possibilitou

a limpeza de falhas no sistema e o desligar de ferramentas de arranque que poderiam levar a atrasos ao longo do funcionamento dos testes. Este concede prioridades a todos os processos constantemente para fornecer um desempenho ideal, diminuindo a prioridade de execução aos que estão a trabalhar em *background*, libertando recursos. Com a utilização deste programa permitiu que os tempos de análise dos testes representassem casos de funcionamento numa máquina nas condições de fabrico. Outra medida tomada para garantir o desempenho ideal foi a aproveitação das opções de energia, ao realizar continuamente os testes com o plano de consumo de bateria no nível de alto desempenho.

Como referido no capítulo 3.1.2 para a análise de consumos de memória foi necessário a utilização da ferramenta JProfiler. Esta dependendo das seleções de *profiling* definidas gera um *overhead* diferente, na aplicação levando a consumos de memória. Uma vez que ambos os executáveis e a aplicação funcionam sobre a mesma JVM foi necessário criar um perfil de análise que diminui-se o máximo o *overhead*. Para reduzir o *overhead* foi criado um "Custom Profile" com as seguintes características do perfil do tipo "Sampling" desligadas.

- *Method call recording*, pois não é necessário de análises de tempos através desta ferramenta.
- *Auto-tunning* pois este apenas executa *profiling* do CPU.
- *Record Payload call stacks in sampling mode*, uma vez que não é utilizado o *sampling mode*.
- *Thread Profiling Monitors*, uma vez que não são analisado os funcionamentos das *threads*.

Uma das mais importantes precauções tomadas foi elaborada no decorrer do desenvolvimento do projeto, ao seguir as boas práticas adquiridas e as sete regras de desenvolvimento de aplicações *multithread*.

4 Implementações

Para ser possível o desenvolvimento do projeto desta dissertação foi necessário elaborar o estudo da API do Java 8 apresentado no capítulo 2. Para além da análise das ferramentas da API, foi necessário investigar sobre algoritmos de ordenação. Com esta investigação foi possível selecionar os que melhor se enquadravam com o projeto e que permitissem um bom fundamento de conclusões sobre os mecanismos de concorrência. A informação sobre o funcionamento de ambos, mecanismos e algoritmos, permitiu uma base de estudo que tornou possível a junção dos mecanismos. Esta junção foi um passo importante e complicado devido à necessidade de garantir que o núcleo do algoritmo nunca era alterado de forma a não influenciar nem o algoritmo de ordenação base nem o mecanismo em teste. Mesmo compreendendo que a regra número sete das sete regras de desenvolvimento de aplicações *multithread*, afirma que se deve modificar o algoritmo para conseguir o melhor nível de concorrência, foi tomada a decisão de preservar o processos de ordenação e distribuição de trabalho o máximo inalterados. Optou-se por esta decisão pois, se for empregue como exemplo uma fórmula como a de *merge* do algoritmo *Mergesort* num outro algoritmo de ordenação como o *Bubblesort*, este de fato poderá funcionar mas unicamente devido à utilização do mecanismo *merge* e não devido à capacidade do algoritmo *Bubblesort* ser apropriado para uma implementação concorrente.

Para melhor compreender as especificações implementadas foi produzido este capítulo onde são referenciadas e justificadas as escolhas tomadas nos diferentes mecanismos de concorrência e nos algoritmos de ordenação. A leitura dos seguintes capítulos funcionará como base de apoio à compreensão das análises de resultados.

4.1 *Single Thread*

O estudo de funcionamento dos algoritmos de ordenação foi iniciado a partir da análise dos modelos tradicionais dos três algoritmos de ordenação escolhidos. A partir desta investigação foram elaborados modelos dos mesmos algoritmos com implementações utilizando as ferramentas de concorrência escolhidas.

4.1.1 *Quicksort*

Como referido no capítulo 3.1, o algoritmo *Quicksort* no decorrer da sua ordenação calcula um valor pivô e realiza uma preordenação durante o cálculo. Este princípio manteve-se inalterado em todas as implementações produzidas, pois este cálculo é o que destaca o algoritmo fazendo um preordenação e uma divisão de trabalho a partir do valor do pivô. Esta implementação contém desvantagens devido ao cálculo para se obter o pivô e deste ser capaz de indicar uma posição demasiado afastada do centro do repositório, produzindo uma divisão em nada balanceada. O processo de ordenação como ilustrado na seguinte excerto de código [Código 6] é invocado recursivamente até todas as partições se encontrarem devidamente ordenadas.

```
public static void orderQuickSort(int[] arr, int lowerIndex, int
higherIndex) {
        int i = lowerIndex;
        int j = higherIndex;
        int pivot = arr[lowerIndex + (higherIndex - lowerIndex) / 2];
        while (i <= j) {
            while (arr[i] < pivot) {
                i++;
            }
            while (arr[j] > pivot) {
                j--;
            }
            if (i <= j) {
                swapNumbers(i, j, arr);
                i++;
                j--;
            }
        }
        if (lowerIndex < j) {\
            orderQuickSort(arr, lowerIndex, j);
        }
        if (i < higherIndex) {
            orderQuickSort(arr, i, higherIndex);
        }
    }
```

Código 6 – Implementação *Quicksort*

Esta implementação corresponde com às necessidades de manter o custo de memória baixo, apesar da necessidade de cada chamada recursiva armazenar dados na *heap*. O algoritmo foi considerado um caso de estudo bastante relevante ao oferecer uma divisão de trabalho diferente da tradicional, ao ser uma implementação bastante casual de encontrar e ser bastante simples de compreender.

Variações do algoritmo tradicional foram elaboradas ao longo do tempo desde da criação do original, como o *Dual-Pivot Quicksort*. Este é atualmente empregado na API do Java 8 e foi aproveitado nos casos de investigação referenciados ao longo do capítulo atual. Não foram desenvolvidos projetos de estudo de variantes do *Quicksort* pois seria produção de conteúdo nada benéfico para a dissertação.

4.1.2 *Mergesort*

O caso de estudo do algoritmo *Mergesort* tradicional [Código 7] apresentou diferenças significativas em relação ao *Quicksort*. O algoritmo *Mergesort* emprega a abordagem *divide-and-conquer* e garante uma divisão bastante equilibrada ao dividir o peso do trabalho pela quantia de "trabalhadores". Este fato produz porções de trabalho com pesos idênticos com a exceção da situação da dimensão ou número de "trabalhadores" serem valores ímpares. Após a divisão do repositório, através de iterações recursivas, é elaborado o processo de conquista. Durante período de conquista o algoritmo compara os valores nas duas metades, de forma a ordenar corretamente. Assim que todas as iterações tenham terminado o seu processo de conquista, o *array* é considerado ordenado.

```
private static void merge_sort(int[] arr, int lowerIndex, int higherIndex)
{
        if (lowerIndex >= higherIndex) {
            return;
        }
        int midleIndex = (lowerIndex + higherIndex) / 2;
        merge_sort(arr, lowerIndex, midleIndex);
        merge_sort(arr, midleIndex + 1, higherIndex);
        int[] A = new int[midleIndex - lowerIndex + 1];
        int[] B = new int[higherIndex - midleIndex];
        for (int i = 0; i <= midleIndex - lowerIndex; i++) {
            A[i] = arr[lowerIndex + i];
        }
        for (int i = 0; i <= higherIndex - midleIndex - 1; i++) {
            B[i] = arr[midleIndex + 1 + i];
        }
        //Merge of Pieces
        int i = 0, j = 0;
        for (int k = lowerIndex; k <= higherIndex; k++) {
            if (i < A.length && j < B.length) {
                if (A[i] < B[j]) {
                    arr[k] = A[i++];
                } else {
                    arr[k] = B[j++];
                }
            } else if (i < A.length) {
                arr[k] = A[i++];
            } else if (j < B.length) {
                arr[k] = B[j++];
            }
        }
    }
```

Código 7 – Implementação do modelo tradicional *Mergesort*

A principal desvantagem desta implementação é a cópia de valores que origina um incremento nos custos de memória. Como vantagem o *Mergesort* ao empregar recursivamente *divide-and-conquer* possibilita a obtenção grandes benefícios nas durações na sua execução. A base do *divide-and-conquer* é repetir o processo de divisão dos problemas em porções de menor dimensão até que estes sejam de uma dimensão considerada. Esta técnica garante vantagens quando comparada a duração da resolução de um problema, com a duração da resolução de porções do problema utilizando paralelismo. Estes fatores destacam o algoritmo e tornaram-no

numa ótima fonte de informação de consumos de tempo e memória com a implementação de concorrência no modelo tradicional.

4.1.3 *Pidgeonholesort*

O *Pigeonholesort* apresenta a mesma desvantagem do mergersort, ao necessitar de um repositório adicional de dados, *array* de "pombos". Este algoritmo diferencia-se pela simplicidade do código e pelo princípio em que se baseia, como visível no Código 8. Uma vez que o *array* de apoio tem uma dimensão igual à diferença entre o valor mínimo e máximos presentes no *array*, este em determinados casos, pode ser uma vantagem. Se o *array* a ordenar contiver só valores binários (0 e 1), o *array* de pombos apenas terá uma dimensão de dois, o que provoca um aumento no consumo de memória quase impercetível. No entanto, se o *array* a ordenar for preenchido com os números de identificação social da população de uma grande cidade, o *array* de "pombos" terá uma dimensão, no mínimo, igual ao *array* que se pretende ordenar.

Para considerar as possíveis implementações concorrentes, é necessário verificar cuidadosamente o *array* de "pombos". Este *array* pode necessitar de ser produzido antes do paralelismo ser iniciado pois todas as *threads/tasks,* com o objetivo da ordenação, necessitam de conhecer os valores mínimos e máximos para a possível junção dos *arrays* de "pombos" gerados.

```
public static void pigeonhole_sort(int[] a) {

// size of range of values in the list (ie, nº of pigeonholes needed)
        int min = a[0], max = a[0];
        for (int x : a) {
            min = Math.min(x, min);
            max = Math.max(x, max);
        }
        final int sizeH = max - min + 1;

        // our array of pigeonholes
        int[] holes = new int[sizeH];

        // Populate the pigeonholes.
        for (int x : a) {
            holes[x - min]++;
        }

        // Put the elements back into the array in order.
        int i = 0;
        for (int count = 0; count < sizeH; count++) {
            while (holes[count]-- > 0) {
                a[i++] = count + min;
            }
        }
    }
```

Código 8 – Implementação *Pidgeonholesort*

No algoritmo apresentado no Código 8, a simplicidade é um benefício muito elevado. O algoritmo com uma implementação concorrente necessita que o cálculo dos valores mínimos e máximos seja concretizado antes da criação de *tasks/threads*. Desta maneira, é possível que o processo

de preenchimento dos *arrays* de "pombos" seja executado concorrentemente e no final a ordenação se encontre correta. Para finalizar a *thread* principal, assim que todos os objetos concorrentes tenham concluído a sua função, é responsável por juntar todos *arrays* de "pombos" produzidos.

4.2 Notas sobre as implementações concorrentes

O capítulo atual foi elaborado de forma a se compreender completamente a explicações do estudo referidas a partir deste momento. Uma vez que para a elaboração das implementações foram estipuladas regras, é necessário explicar devidamente os fatores e as soluções procuradas para garantir o melhor equilíbrio entre os mecanismos concorrentes. Como referido previamente no capítulo 4.1, os algoritmos tradicionais são utilizados como base do estudo. Os processos dos algoritmos de ordenação que não fossem possíveis de implementar de igual forma em todas as circunstâncias, foram reconstituídos para que as durações de execução não fossem afetadas. Os fatores como o cálculo do valor de pivô, o *merge* de resultados e a junção do *array* de "pombos", são referenciados ao longo da dissertação e explicados como influenciam a implementação dos mecanismos. As explicações foram consideradas necessárias uma vez que estes fatores influenciam os consumos, e a necessidade de compreender o impacto das divisões de problemas nas durações de execução.

Todo o raciocínio sobre o desenvolvimento do projeto foi realizado com a colaboração de uma estrutura de implementação de funcionalidades. Esta estrutura indica onde, como e quando é que pode ser efetuada uma etapa do algoritmo [Figura 16]. A estrutura foi utilizada independentemente da implementação do algoritmo ordenação. O processo deve ser iniciado com a criação e o preenchimento do *array* que se pretende ordenar e com a estipulação da quantia de *tasks*/*threads* que se pretende utilizar. Assim que seja invocado o método para iniciar a ordenação, este vai implementar um conjunto de diversos mecanismos de concorrência. Depois da declaração dos mecanismos, é iniciado o processo de criação das tarefas. Durante este processo, é elaborada a divisão do problema, de acordo com o algoritmo de ordenação, pela quantia de *tasks*/*threads* definidas. Em simultâneo com a divisão do problema, são atribuídas as porções do mesmo aos objetos de concorrência e é realizado sinal para estes iniciarem o processo de ordenação. Durante a resolução dos problemas, a *Main Thread* verifica o estado das *tasks*, se necessário, de forma a realizar trabalho extra em paralelo, caso seja possível. No final deste procedimento, são elaborados os procedimentos necessários para obter o *array* final devidamente ordenado e com quaisquer consumos extras terminados.

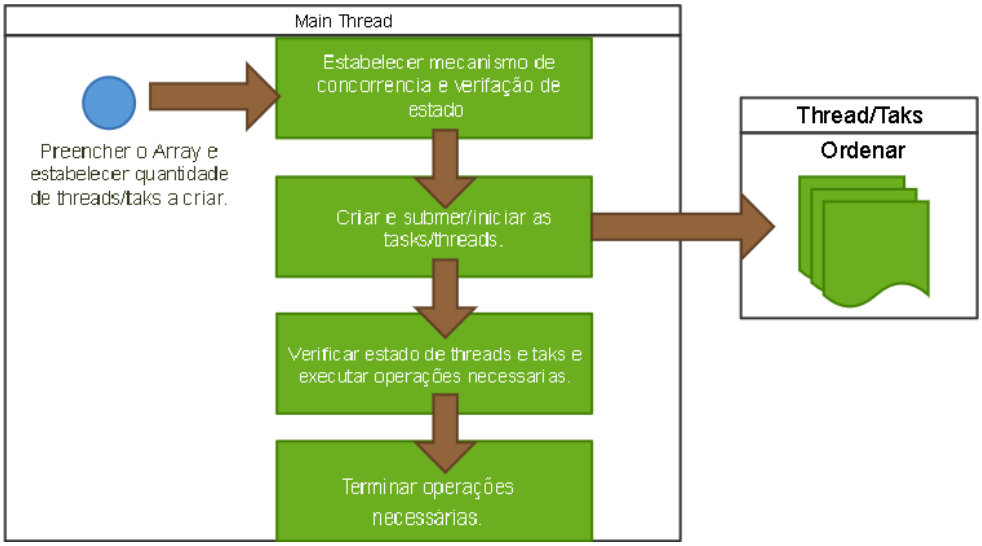

Figura 16 – Estrutura de elaboração de algoritmos com mecanismos de concorrência

4.3 *Threads*

De forma a estudar o funcionamento e os consumos dos diferentes algoritmos de ordenação, com a implementação de *threads*, foram criadas nos packages de cada algoritmo, uma classe que fizesse *extends Thread*. As classes elaboradas diferem em alguns pormenores, para além dos algoritmos de ordenação. Durante o estudo foi estabelecido a regra que, se possível, nas classes deve existir pelo menos um construtor que recebesse o número de *threads* a criar, de forma elaborar a criação dos objetos de concorrência e manter uma estrutura corretamente organizada. Para além deste procedimento, outras modificações foram realizadas no código entre as classes, devido às necessidades dos algoritmos de ordenação. Os seguintes subcapítulos fornecem informações sobre as diferenças existentes entre os algoritmos e elaboraram, em conjunto, um esclarecimento sobre o fluxo de funcionamento da implementações, elucidando sobre o porquê e quais as decisões tomadas. Um fato a ter em conta é apesar da *Main Thread* por vezes executar trabalho, esta não ser considerada como criada já que se encontra ativa com a execução do projeto.

4.3.1 Método de criação e inicialização

Este procedimento, nas versões dos algoritmos *Quicksort* e *Mergesort*, é implementado através de recursividade e da criação de uma estrutura em árvore que produz o número de *threads* desejadas. Este processo inicia-se através da invocação do método, da passagem do *array* que se pretende ordenar e da quantia de *threads* que se deseja criar. Uma vez que este processo vai ser recursivamente invocado, a verificação do valor vai ser a operação de paragem. De seguida, é calculada a dimensão das duas metades através do cálculo do pivô, *midleIndex*, de acordo com o algoritmo em execução.

Uma das validações necessárias mas não muito encontradas ao longo do estudo de possíveis implementações, foi como produzir um valor ímpar de *threads*. Esta verificação é possível

através da condição formulada na situação da quantia de *threads* ainda por criar ser igual a um, sendo este o valor impar mais baixo que se pode identificar. A condição leva a que uma *thread* já criada fabrique uma nova *thread* para executar metade do seu trabalho, em vez da implementação normal de produzir duas novas *threads*, ambas com metade do trabalho. Na implementação tradicional, a *thread* progenitora apenas é responsável pelo código após invocação do método *join()*. Este tipo de implementação não é muito produtivo em todas circunstâncias. Na implementação do algoritmo *Mergesort* a *thread* progenitora tem a responsabilidade do *merge* dos resultados mas no algoritmo *Quicksort* os resultados das *threads* são independentes, não existindo código a executar após a ordenação. Ao verificar a condição de ser necessário um número impar de *threads,* a *thread* em execução irá criar uma nova e dividir o seu trabalho com esta. De seguida a *thread* progenitora realiza a ordenação a sua metade e invoca o método *join()* da nova *thread*. O funcionamento deste processo pode ser verificado no fluxo ilustrado na Figura 17.

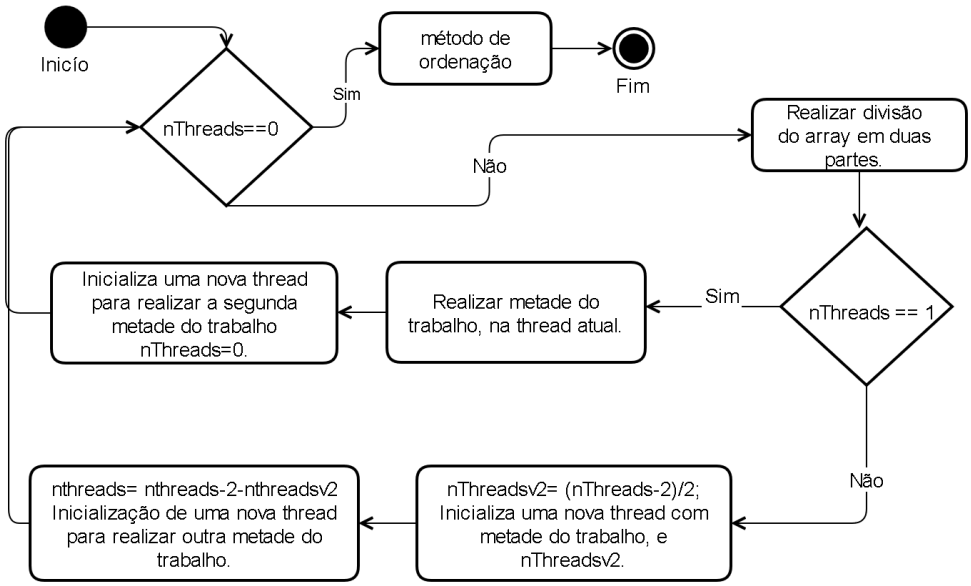

Figura 17 – Implementação da criação de *threads* no *Quicksort* e *MergeSort*

Na implementação do algoritmo *Pidgeonholesort* com o mecanismo *Thread*, para o processo de criação foi simplificado. Uma vez que é possível o cálculo da dimensão de um bloco do problema, através da divisão do repositório pela quantia de *threads* a criar, optou-se por uma implementação não recursiva. O processo de criação de *threads* é iniciado pelo cálculo do bloco de dados e do valor máximo e mínimo dos elementos presentes no *array*, utilizando os mecanismos de Arrays.copyofRange, *Math.min* e *Math.max*. Assim que obtidos todos estes valores se encontrem calculados, são enviados por parâmetro para a *thread* em criação e esta é inicializada. Todo o procedimento elaborado é observável no fluxo ilustrado na Figura 18.

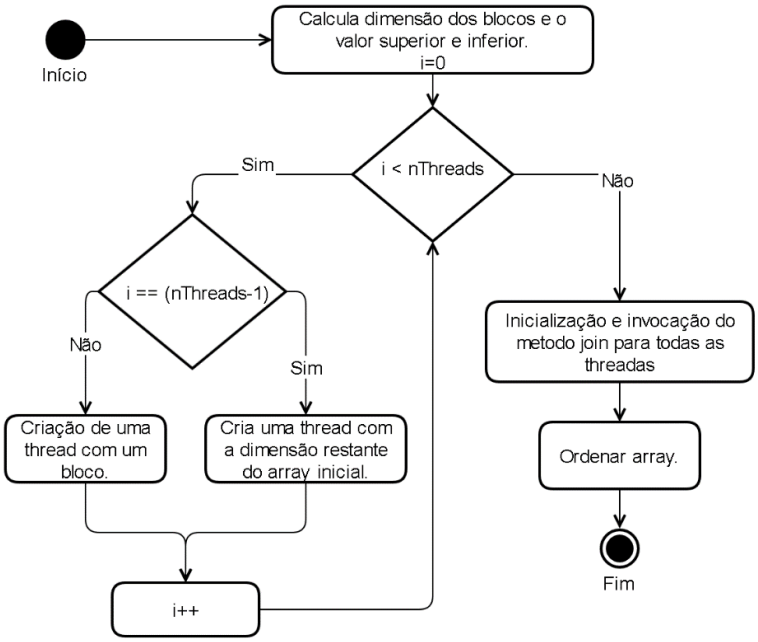

Figura 18 – Implementação da criação de *threads* no *Pidgeonholesort*

4.3.2 Obtenção de resultados

A obtenção de resultados refere-se a duas necessidades no desenvolvimento do projeto, a necessidade de execução de trabalho pelas *threads* progenitoras e ao processo de como o *array* ordenado é obtido. O quicksort, não tem nenhuma das necessidades, uma vez que este trabalha em todos os momentos sobre o mesmo *array*, este não necessita de executar código após o funcionamento das *threads*. Como todas as *threads* criadas são iniciadas e de seguida invocam o método *join(),* existe a garantia que assim o sistema se encontre livre o *array* se estará devidamente ordenado. Ao contrário do *Quicksort*, o *Mergesort* e o *Pidgeonholesort* têm a necessidade de realização de trabalho após o funcionamento das *threads*. O *Mergesort*, de acordo com o algoritmo previamente explicado, necessita de realizar *merge* das duas metades produzidas. Esta necessidade é efetuada, numa estrutura em árvore, pelo nó superior como ilustrado nas seguintes imagens Figura 19 e Figura 20.

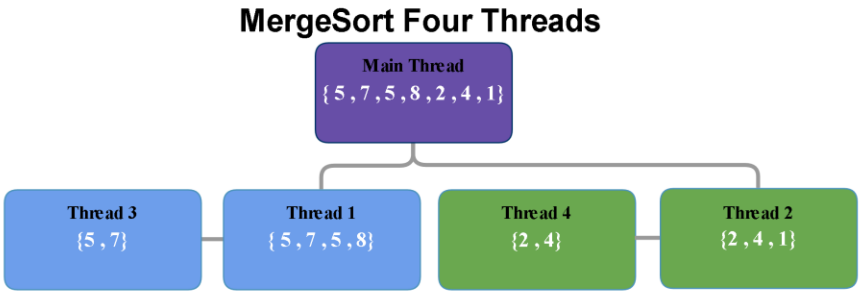

Figura 19 – Exemplo de criação de um número par de *threads*

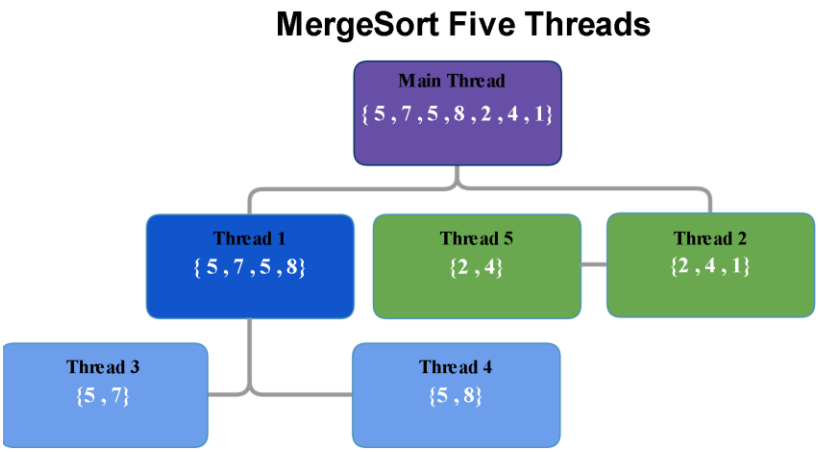

Figura 20 – Exemplo de criação de um número impar de *threads*

Como é possível analisar na Figura 20 a criação das *threads* é realizada segundo uma estrutura em árvore, possibilitando a criação de uma ou duas *threads* por uma já existente. Com base na ilustração, as *threads* progenitoras (ramos superiores) necessitam que as *threads,* diretamente descendentes, terminem a sua função de forma a prosseguir. Assim que os ramos inferiores terminem, o ramo superior adquire os resultados destes e executa o *merge*. Um ponto a realçar é o fato da possível criação de uma *thread* "filha" ao mesmo nível da thread "pai" (ex: A thread 2 cria a Thread 5). Nestes casos estas trabalham em simultâneo e assim que terminem a sua função, a *thread* progenitora irá realizar o *merge* dos resultados de ambas. Com a utilização do sistema de cores implementado é simples de perceber os *merges* realizados. Os resultados da *Thread 3* e da *Thread 4*, ambas coloridas a azul, originam os valores para *merge* a ser realizado pela *Thread 1*, colorida a azul escuro. A *Thread 2* inicialmente divide o seu trabalho a meio e cria a *Thread 5* para resolver metade, quando ambas terminarem a ordenação, a *Thread 2* realizará o *merge* dos resultados. Assim que a *Thread 1* e *Thread 2* terminaram os merges dos seus descentes, retornaram os seus resultados a *Main Thread* que realizara o *merge* final de forma a se obter o *array* ordenado.

O *Pidgeonholesort* funciona de forma diferente, este como o *Mergesort* funciona com um sistema de *merge* de resultados, onde são agregados os *arrays* de "pombos" das múltiplas *threads*. As principais diferenças entre algoritmo de ordenação e os restantes são o fato de todas as *threads* serem produzidas pela *Main Thread* e a realização dos *merges* serem unicamente realizadas por esta. Os *merges* são efetuados através da implementação de um método de acesso a resultados nas classes responsáveis pelas *threads*. Este método permite recolher os resultados das *threads*. No *Pidgeonholesort* uma vez obtidos os resultados, estes são somados até produzir o *array* de "pombos" completo e convertido no *array* ordenado.

4.4 *ExecutorService*

De forma a progredir com desenvolvimento do projeto, progrediu-se para o uso dos mecanismos de *taks* e *threadpool*. Através do mecanismo *ExecutorService* é possível estabelecer o tipo de *threadpool* e dimensão do paralelismo desta. Este mecanismo requer dois pontos de atenção,

que tipo de tarefas se vai submeter ao *executor* e como saber quando desligar este. É necessário saber quando é possível desligar o executor, uma vez que este continua o consumo de recursos até que seja invocado um dos métodos *shutdown*. Caso o *shutdown* seja executado prematuramente, as *tasks* ainda em espera de serem iniciadas serão removidas, provocando falhas na obtenção do resultado final. De forma a ser possível ter conhecimento do ponto de situação das tarefas submetidas foram utilizadas três implementações diferentes, uso de booleanos, a utilização do *CompletionService* e a implementação de um *CountDownLatch*. Cada um dos mecanismos de verificação de *tasks* foi também implementado com os três tipos de *threadpool*, elaborando um total de nove casos de estudo.

As *tasks* criadas, sempre que executadas realizam exatamente o mesmo código que o produzido para o caso de estudo com *threads*, de forma a ser possível uma comparação de eficiência entre os mecanismos. As *tasks* são produzidas para os algoritmos *Mergesort* e *Pidgeonholesort* sequencialmente através do cálculo da dimensão de um bloco de dados equivalente à dimensão do *array* inicial a dividir pelo total de *tasks* a criar. O mesmo não foi possível implementar no Quicksort, para este foi necessário elaborar a estrutura de criação de objetos em árvore, devido à estrutura que particionamento do *array* produz. Na divisão de trabalho vez que a divisão da dimensão do array com o número de *tasks* pode resultar num valor com casas decimais, e a API do Java arredonda estes valores quanto utilizado *integers*. Devido a este arredondamento foi necessário validar os blocos de dados atribuídos às *tasks* de forma que, não ocorrerem erros como *stackoverflow* durante a cópia dos blocos. A validação das dimensões dos blocos é executada durante a criação das *tasks*. Na situação da soma das dimensões dos blocos já atribuídos às *tasks* com a dimensão do bloco para a *task* em criação, ultrapassar a dimensão do *array* base, a *task* em criação apenas possuirá um bloco cuja dimensão seja igual aos dados ainda por atribuir. Caso esta verificação seja comprovada, a criação de *tasks* será terminada pois não é possível facultar mais dados às *tasks*.

Para iniciar a execução de um caso de estudo com *ExecutorService,* o número de tasks a criar tem de ser enviado como parâmetro acompanhado pelo *array* a ordenar. Ao iniciar o caso é declarado o objeto *ExecutorService*, a *threadpool* desejada e calculada a dimensão de um bloco de dados. Estes procedimentos são efetuados em qualquer uma das implementações, independentemente do mecanismo de verificação de estado de *tasks*. Para terminar todas as implementações invocam devidamente o método *shutdown()*.

4.4.1 Verificação com utilização de booleanos

A verificação utilizando booleanos é uma implementação muito casual devido à sua simplicidade. A implementação do estudo para esta verificação necessitou da criação de uma classe que implementasse *Runnable* em cada um dos diferentes packages. As classes foram implementadas com um construtor para criar o objeto *task* com os paramentos *array* a ordenar em todas as circunstâncias. Nos casos de ordenação com os algoritmos *Mergesort* e do *Quicksort* é enviado também por parâmetro o valor correspondente ao índex inicial e o valor de índex final. No caso do *Pidgeonholesort* são enviados o valor máximo e mínimo para que as *tasks* sejam capazes de construírem os seus *arrays* de "pombos". Nas classes implementadas por este estudo é estabelecido uma variável do tipo booleano a representar o estado da *task*. Assim que o método

run() terminar é estabelecido o valor do estado da *task* a *True*, de forma a ser possível informar que a *task* terminou. A partir do objeto a representar uma *task,* a *Main Thread* é capaz de invocar o método de consulta *getReaddy()*, que devolve o estado dessa *task*. Para os algoritmos *Mergesort* e *Pidgeonholesort* foi necessário também criar um método de acesso ao bloco de resultados.

No processo de verificação dos estados das *tasks* foram tomados alguns cuidados pois este procedimento pode ser muito demorado, uma vez que nunca é possível garantir a certeza que o estado de terminado será retornado com a invocação do *getReaddy()*. As *tasks* são verificadas sequencialmente até que todas retornem o seu estado como concluído. Este procedimento faz com que a implementação não seja muito eficiente, uma vez que a ordem pela qual as *tasks* são iniciadas pode não corresponder com a ordem de conclusão. De forma a rentabilizar o tempo de espera pelo sistema no caso de *Mergesort* e do *Pidgeonholesort* foram realizadas precauções. Uma vez que estes necessitam de juntar os resultados obtidos das *tasks,* foi elaborada a implementação de forma que uma *task* quando terminar é imediatamente alocada como pronta para *merge*. Assim a *task* que foi verificada como conluiada fornece os dados e é removida do processo de verificação. A partir do momento que duas *tasks* concluam a sua função é realizado o processo de *merge* dos resultados, interrompendo a verificação das restantes *taks* e possibilitando mais tempo às que ainda não concluíram até a próxima verificação.

No caso do *Quicksort* as *tasks* assim que retornem o seu estado como concluído são imediatamente removidas pois não fornecem qualquer utilidade a *Main Thread*, ao trabalhar diretamente no *array* final. Assim que todas as tasks tenham retornado o seu estado como concluído, o *array* encontra-se devidamente ordenado e o pode ser realizado o *shutdown* do executor com segurança.

Todo o processo de verificação de estados, até a invocação do método *shutdown()*, pode ser verificado no fluxo ilustrado na Figura 21. Apesar de ser um procedimento muito semelhante em todos os algoritmos de ordenação, algumas diferenças existem. No *Quicksort* a *Main Thread* não executa trabalho além da verificação de estados das *tasks*. No funcionamento do *Mergesort* o processo de verificação é executado duas vezes, de forma a construir uma estrutura em árvore semelhante ao *merge* do caso com *threads*. Na primeira verificação apenas a primeira *task* a terminar é verificada e removida, de modo a estabelecer o resultado desta como ramo esquerdo. Na segunda verificação são averiguados os estados das restantes *tasks* e quando uma destas é concluída o seu resultado passa a ser considerado o ramo direito. Quando um novo ramo da direita é obtido é efetuado o *merge* dos dois ramos e o resultado deste é considerado o novo ramo esquerdo. Assim que todas as *tasks* se encontrem concluídas o ramo esquerdo é convertido no *array* final.

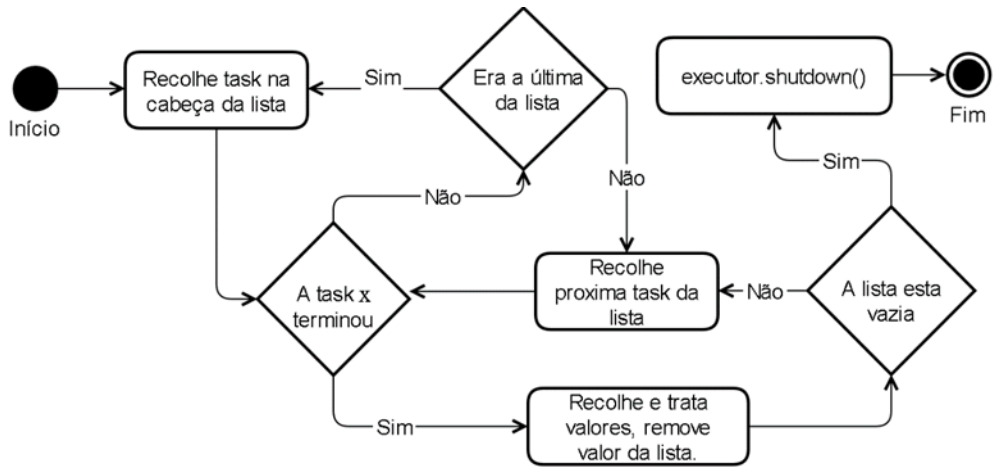

Figura 21 – Verificação de estado usando booleanos

4.4.2 *ExecutorCompletionService*

Como referido no capítulo 2.4.4, um *Executor* que implementa o *ExecutorCompletionService* aloca as *tasks* terminadas numa *queue*, à qual é possível aceder para retirar consoante a nossa necessidade. Este mecanismo fornece um apoio muito necessário na resolução do problema que é conhecer a situação das *tasks*. Com a implementação deste mecanismo a necessidade dos valores booleanos para representar estado torna-se nula. De modo a permitir o estudo do *ExecutorCompletionService* foram criadas três novas classes em cada package de algoritmo. Estas classes foram implementadas como *Callables* e como retorno nas classes dos *packages Mergesort* e *Pidgeonholesort* foi estabelecido um *array* com *integers*, e na classe do *package Quicksort* foi estabelecido *Void*, uma vez que este não necessita de qualquer retorno.

As classes foram construídas com uma configuração muito semelhante as classes da verificação com booleanos, utilizando construtores com os mesmos parâmetros, e o mesmo código de ordenação. Foram apenas removidos os métodos de acesso, uma vez que se tornaram desnecessários, já que os resultados e os estados são com grande facilidade adquiridos.

A invocação do método para começar o estudo é igualmente iniciado com a passagem dos parâmetros número de *tasks* e o *array* que se deseja ordenar. De seguida é inicialização o *ExecutorService* com o tipo de *threadpool* desejada e enviado o objeto *ExecutorService* por parâmetro na criação do *ExecutorCompletionService*. Um exemplo do código implementado que representa situação referida pode ser verificado no Código 9.

```
ExecutorService executor = Executors.newWorkStealingPool();
// QuickSort void Callable
CompletionService<Void> completionService = new ExecutorCompletionService
<Void> (executor);
```

Código 9 – Implementação de *ExecutorService* com *CompletionService*

Após a declaração referida foi implementado o processo de criação de *tasks*, começando pela divisão do problema pela quantidade de trabalhadores para o resolver. Esta implementação foi

mais complexa no algoritmo de ordenação *Quicksort*, para este foi optada uma abordagem recursiva *divide-and-conquer*. Devido à dificuldade do particionamento com concorrência, a implementação foi desenvolvida de acordo com a Figura 22. O problema é dividido e de seguida é consultada a quantia de *tasks* que o ramo da esquerda irá criar, permitindo ao ramo da direita criar apenas o número de *tasks* necessárias.

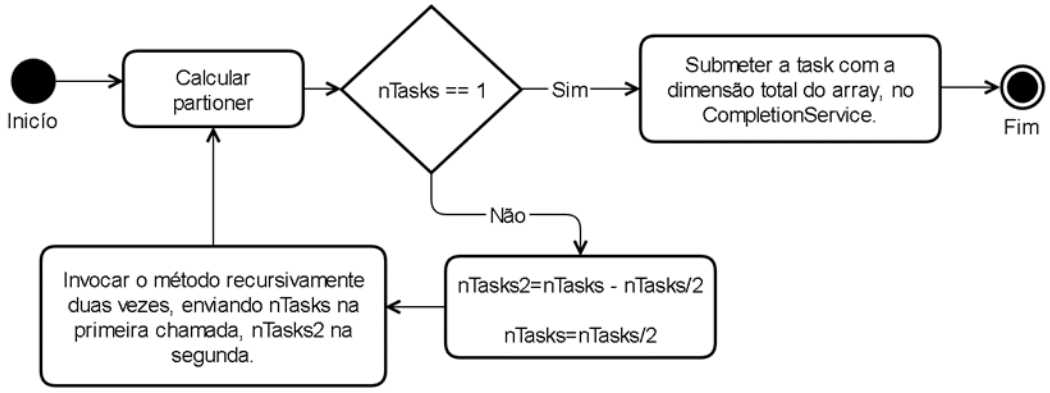

Figura 22 – Processo de criação de *tasks* com ordenação Quicksort

Posteriormente a declaração e início de todas as *tasks* é iniciado o procedimento de verificação de estado. Este é efetuado através de um ciclo de verificação, no qual utilizasse o objeto *CompletionService*, através do qual é invocado o método *take()* para adquirir o objeto do tipo Future que representar a próxima tarefa concluída. Sempre que uma *task* é concluída é decrementado o número de tarefas inicialmente desejadas, até que este valor seja igual a zero terminando o ciclo. O processo de verificação de estado necessitou de alguns ajustes dependendo do algoritmo de ordenação. A exceção é o *Quicksort*, uma vez que este apenas necessita de saber o estado das tarefas de forma a poder invocar o *shutdown* do executor com segurança, não foram realizados ajustes. Uma vez que é necessário efetuar operações sobre os resultados das *tasks*, no algoritmo de ordenação *mergesort* ou *pidgeonholesorte*, antes de remover a *task* do ciclo quando terminada é armazenado o seu resultado. Na implementação do *ExecutorCompletionService* com o algoritmo *Pidgeonholesort* assim que uma *task* termine é adquirido o seu *array* de pombos através de um método de acesso. De seguida são adicionados os seus elementos ao *array* de pombos final, incrementando os valores já presentes com os valores da *task*. Na implementação da ordenação com o algoritmo do *Mergesort* realizou-se um procedimento semelhante ao caso de estudo com *threads*. Ao obter o resultado da primeira *task* terminada é afirmado que estes são os valores do nosso ramo da esquerda e assim que a próxima *task* terminar, esta será o novo ramo da direita. De seguida é executado o *merge* destas, e esta junção de resultados será o novo ramo esquerdo.

4.4.3 *CountDownLatch*

O *CountDownLatch* é um objeto de contagem que permite a verificação do total de *tasks* em execução. Ao criar-se uma *task* é possível transmitir ao *CountDownLatch* que uma nova *task* foi produzida e que este necessita de incrementar o seu contador. Na conclusão de uma *task*, o *CountDownLatch* deve ser comunicado da conclusão, e deve decrementar o contador. Com a

implementação deste mecanismo e a garantia que este possui um valor final igual à quantidade de *tasks* produzidas, é possível elaborar um sistema de controlo de estado de tasks bastante simples. Ao invocar o método *wait()* a partir do objeto *CountDownLatch* é possível implementar um sistema de espera até que o contador atinja o valor de zero, garantindo assim que todas as *tasks* foram concluídas.

Para a implementação deste mecanismo foi necessário produzir uma nova classe, esta foi elaborada com um formato semelhante ao da classe com a implementação de booleanos para representar o estado. Esta nova classe foi implementada como *Runnable*, com um construtor onde o parâmetro booleano foi substituído por um objeto *CountDownLatch* de forma que todas as *tasks* informem o mesmo objeto do seu estado. Assim que uma *task* termine a sua porção de ordenação, esta invoca o método *countDown()* através do objeto *CountDownLatch*, decrementado assim o número de *tasks* que a *Main Thread* aguarda que terminem. Uma vez que o processo das *task* é executado através de *Runnables* foi necessária a implementação dos construtores de acesso a resultados. Quando todas as *task* terminem sua execução, a *Main Thread* retornará ao estado de execução, desenvolvendo o restante trabalho necessário. Ao implementar este mecanismo não é possível concretizar trabalho com os resultados adquiridos das *tasks* assim que estas concluam, ao contrário das restantes implementações com o *ExecutorService*.

Tanto nas implementações com *Quicksort* ou *Mergesort* a etapa final de obtenção do resultado final continuou bastante simples, em relação às restantes implementações no *ExecutorService*. Na implementação para ordenação através do *Mergesort* foi necessário uma modificação, uma vez que não é possível declarar o ramo da esquerda com o valor resultante da primeira *task* a concluir. Este procedimento foi substituído pela atribuição ao ramo esquerdo do valor referente à primeira *task* a ser estabelecida e submetida no executor. A escolha sobre qual a *task* a ser atribuída como valor inicial do ramo da esquerda é insignificante, uma vez que em nada modifica a obtenção do resultado final. Assim que atribuído o ramo da esquerda, inicia-se um ciclo de verificação das restantes *tasks*, e execução do *merge* entre a próxima *task* e o resultado do ramo esquerdo e atribuindo este novo resultado ao ramo da esquerda.

4.5 *ForkJoinPool*

Para o estudo do mecanismo *ForkJoinPool* foram estabelecidas *tasks* do tipo *RecursiveAction* ou *RecursiveTask<int[]>*, de acordo com as necessidades do algoritmo de ordenação. Uma vez que este mecanismo permite a utilização de *tasks* com ou sem retorno, optou-se pela implementação de acordo com o algoritmo de ordenação em questão. Nas implementações em que a obtenção de resultados das *tasks* era indispensável optou-se pela utilização da *RecursiveTask<int[]>*, em vez de desenvolver métodos de acesso. Esta decisão teve em consideração o aproveitamento das ferramentas disponíveis e a desnecessidade de desenvolver tarefas do mesmo tipo, uma vez que estes nunca serão comparados.

A utilização deste mecanismo foi bastante simples de implementar, necessitando apenas da criação de um objeto do tipo *ForkJoinPool*, com a passagem por parâmetro do nível de paralelismo, e da submissão de uma tarefa. Uma vez que a estrutura de funcionamento do

Fork/Join é fundamentada no *divide-and-conquer* a criação das tarefas é iniciada pela tarefa submetida.

Os diferentes tipos de *tasks* foram utilizados da seguinte maneira:

- *RecursiveAction*

  Este tipo de task foi estabelecido para efetuar as ordenações através do algoritmo *Quicksort*, uma vez que este nunca necessita de qualquer retorno de informações das tasks. A utilização da estrutura *divide-and-conquer* foi bastante simples devido a ser uma implementação muito comum, não necessitando de grandes cuidados.

- *RecursiveTask*

  As *task* do tipo *RecursiveTask<int[]>*, foram implementadas nas classes para a ordenação com *Mergesort* e *Pidgeonholesort*. No *Mergesort* o procedimento de *merge* é efetuado pelos ramos inferiores e o produto desta operação será retornado para o ramo superior. Este fato repete-se até alcançar a raiz/ramo mais elevado onde o resultado é devolvido obtendo assim o *array* final ordenado. Necessitando assim apenas da invocação do método *shutdown*() da *threadpool* na *Main Thread*. No caso da implementação para ordenação com pidgeonholesorte, nos ramos mais inferiores será calculado os *arrays* de pombos. Devolvendo estes para os ramos superiores onde serão somados, de acordo com a estrutura em árvore. Ao inverso do *Mergesort*, este assim que alcance a raiz/ramo máximo, este revolverá o *array* de pombos, através de todas as somas previamente efetuadas. Atribuindo assim a responsabilidade de criação do *array* devidamente ordenado à *Main Thread*.

Uma vez que não existe um número de *tasks* a criar de tarefas foi necessário implementar um sistema para paragem de criação. Para este sistema foi estabelecido um valor mínimo que um *array* pode atingir como dimensão até concluir a sua divisão e iniciar a sua resolução. Este valor é denominado de *threshold*, e é calculado através da fórmula (2) no capítulo 3.1.1.Esta define um valor mínimo consoante o *hardware* presente na máquina mas, com a implementação desta foi averiguada uma circunstância que poderia levar a não ser possível garantir o mesmo proveito do *threshold* caso a dimensão do *array* ultrapassa-se os 2750. Para ultrapassar esta circunstância foi estabelecida a condição de no caso da dimensão do *array* ultrapassar o *threshold*, este seria estabelecido com o menor valor da comparação do *threshold* com 45% da dimensão do *array* a ordenar. Esta condição foi obtida na duração do estudo do mecanismo e da tentativa de estipulação de regras de forma a garantir qualidade nos resultados. Na criação das tarefas caso o valor da dimensão do problema seja superior ao valor de *threshold*, duas novas tasks serão criadas dividindo a dimensão do problema de acordo com o algoritmo de ordenação. Assim que criadas as novas *tasks*, estas passam pela mesma condição de verificação de dimensão de problema. Com a criação de ramos inferiores, o ramo que os gerou efetua o invoke destes para determinar o momento em que estes concluam as suas operações. Assim que determinado estado de concluído dos ramos inferiores, estes irão retornar os seus resultados, até atingir o ramo máximo terminando a ordenação. Quando a *Main Thread* tiver conhecimento que a *task*

que produziu se encontra concluída, através do invoke inicial realizado sobre esta, irá efetuar o *shutdown* da *threadpool* dando por concluída a ordenação.

4.6 *JavaSort*

O estudo *JavaSort* foi fundamentalmente a investigação das ferramentas presentes na API do Java 8, que tenham preencham o objetivo de ordenação de *arrays* de forma a se obter uma base de análise. A partir do estudo do código implementado na API, foi possível adquirir conhecimento de várias ferramentas e mecanismos que permitem a ordenação de um *array* de *integers*. Estes mecanismos presentes na API são derivados das classes *Arrays* e *IntStream*, estes ordenam os *arrays* passados por parâmetro e permitem através de métodos já implementados a ordenação com paralelismo. No caso da classe *Arrays*, esta permite-nos uma ordenação utilizando o algoritmo Dual-Pivot *Quicksort* ao invocar o método *sort()*. Esta ordenação é possível de ser aplicada de forma paralela invocando o método parallelSort() da classe. A classe *IntStream* permite a ordenação com a invocação do método *sorted()* ao qual é possível agrupar a invocação *parallel(),* tornando a excução paralela. Ao utilizar esta classe, o resultado obtido será uma *stream* que é possível ser convertida em *array* invocando o método *toArray()*. As quatro implementações permitem assim obter uma base de implementações para o estudo de consumos de tempos e de memória, com utilização ferramentas bastante trabalhas e especificadas.

4.7 Demonstrações dos resultados dos testes funcionais

Uma vez que este estudo necessita de certeza absoluta de que todo o processo de ordenação e de utilização dos mecanismos funciona sem qualquer margem para erros, todos os processos foram devidamente validados. Tal como referido no capítulo 3.2, os *arrays* são validados de duas formas, se estão devidamente preenchidos, e se todos os elementos presentes no *array* pré ordenação ainda presidem após esta. Para produzir um documento viável com os resultados destas validações, foi desenvolvido o método denominado de *AlgorithmCheck()*. Este método vai efetuar ambos os testes sobre todas as implementações efetuadas de um algoritmo de ordenação. Todas as verificações executadas no projeto foram comprovadas de acordo com as informações retiradas ou dos ficheiros de registo de resultados ou da ferramenta de análise JProfiler.

Devido à quantidade de resultados obtidos dos testes efetuados através do método *AlgorithmCheck()*, foi composta a seguinte tabela [Tabela 6] de indiciação de qual o anexo onde pode ser visualizados os resultados pretendido.

Tabela 6 – Listagem dos resultados dos testes funcionais

Algoritmo de ordenação	Dual-Core	I5	I7
Quicksort	Anexo A	Anexo D	Anexo G
Mergesort	Anexo B	Anexo E	Anexo H
Pidgeonholesort	Anexo C	Anexo F	Anexo I

Demonstrando através destes documentos que os algoritmos implementados realizam o seu objetivo inicialmente traçado com eficácia. Não foi realizado nenhum teste aos mecanismos de ordenação da API do Java pois acredita-se na competência da empresa Oracle.

Para se demonstrar que os algoritmos com mecanismo *Thread* criam a quantia de *threads* pretendida, foram realizados testes utilizando ferramenta JProfiler. A partir desta foi possível obter bastantes dados que comprovam que de fato as implementações foram realizadas com sucesso.

A seguinte imagem [Figura 23] representa um dos resultados obtidos, nesta é possível verificar a execução do estudo do mecanismo de *threads* do package *Quicksort*. Este teste realiza a ordenação de *array* em sete situações, cada uma com um valor diferente de *threads* a criar, começando em duas e incrementando até oito. Antes da realização do teste é efetuado uma mudança no estado da *Main Thread* durante um segundo para facilitar a leitura da imagem. Antes da criação do *array* na fase de execução, representada pela cor verde, da *Main Thread*, está irá estar em estado de repouso, representado pela cor amarelo, durante um segundo.

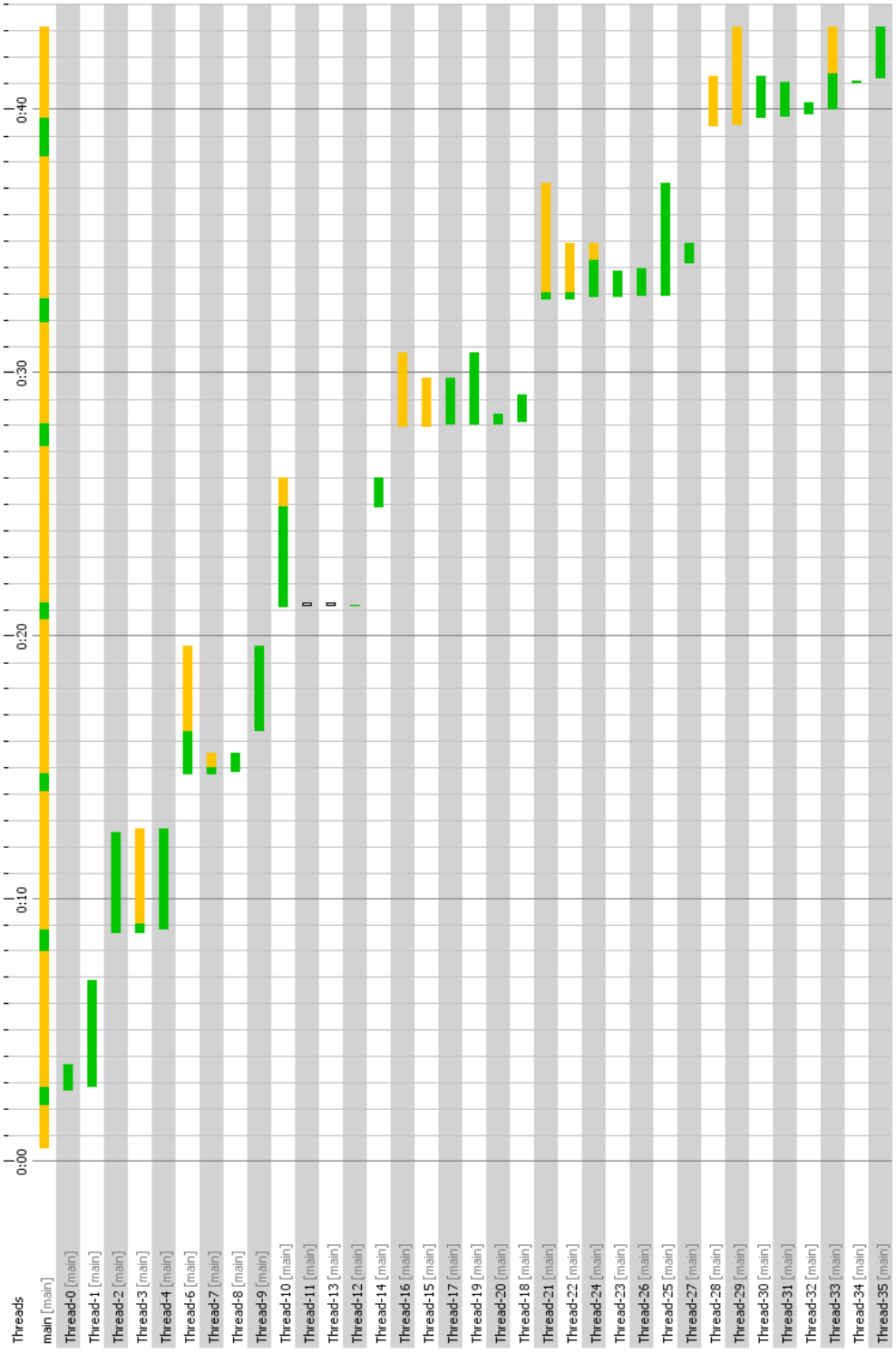

Figura 23 – Ilustração de criação de *threads*

De modo a demonstrar que as *tasks* são produzidas de acordo com a necessidade estabelecida pelo teste, foram analisadas as monitorizações possíveis do JProfiler. Das monotorizações foi conseguível adquirir informações do processador sobre a execução das *threadpools* que operam as *tasks*. Ao analisar execução das *threadpools* foi possível adquirir valores sobre a quantidade de invocações realizadas a determinadas classes. Com estes dados foi possível demonstrar que a quantidade desejada de *tasks* a serem produzidas é satisfeita por completo. A seguinte

imagem [Figura 24] representa graficamente o funcionamento de uma experiência, de criação de threadpools e de *tasks*. Esta é efetuada ao mecanismo *ExecutorService* com verificação ao estado utilizando o *CountDownLatch* e empregando uma *newFixedThreadPool*. Nesta *threadpool* é comprovada a produção de *tasks,* realizando cinco testes de criação de tasks. Em cada teste é submetido um valor de criação de *tasks* diferente, começando por gerar duas e multiplicar o valor por dois em cada nova situação.

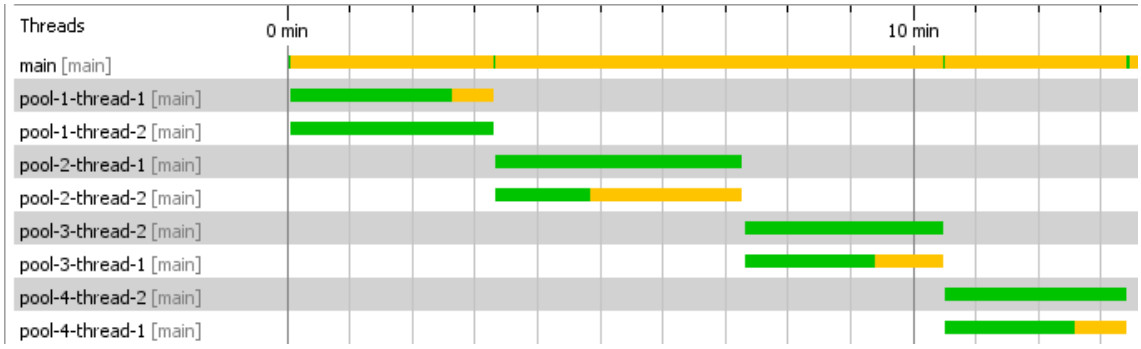

Figura 24 – Exemplo de criação de *threadpools* e *tasks*

Para completar a análise da imagem anterior foram adquiridas informações da execução de cada *thread* presente na *threadpool*. A Figura 25, situada na página seguinte, foi conseguida a partir do JProfiler, esta representa as mesmas *threads* da Figura 24, e apresenta as invocações realizadas às classes. Através das invocações é possível verificar a quantia de *tasks* em execução. Com a Tabela 7 é possível verificar um resumo rápido de ambas figuras, uma vez que estas são de grandes dimensões, e complexas de compreender separadamente.

Tabela 7 – Invocações de *tasks* pelas *threads* existentes nas *threadpools*

Teste	**Threads atribuidas**	**Total de tasks a criar**	**Cálculo de invocações (thread-1 + thread-2)**
pool-1	pool-1-thread-1 pool-1-thread-2	2	1 + 1 = 2
pool-2	pool-2-thread-1 pool-2-thread-2	4	1 + 3 = 4
pool-3	pool-3-thread-1 pool-3-thread-2	8	6 + 2 = 8
pool-4	pool-4-thread-1 pool-4-thread-2	16	8 + 8 = 16

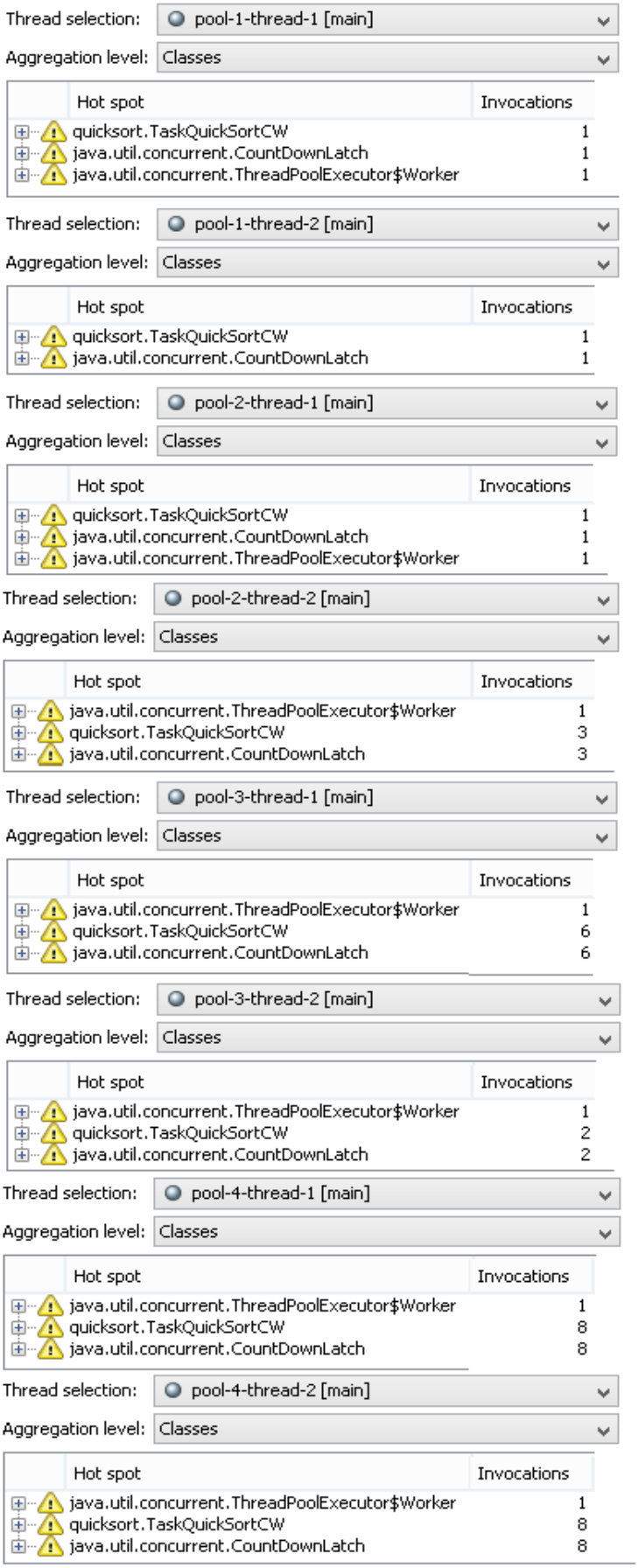

Figura 25 – Demonstração de invocações de *tasks* pelas *threadpools*

5 Análise de testes

Os seguintes testes foram realizados de forma a produzir informações que, facultassem uma base de conhecimento sobre as vantagens e desvantagens de cada mecanismo de concorrência escolhido. A informação representada pelos consumos de memória e tempo de duração do algoritmo é representada ao longo da dissertação por ilustrações dos ficheiros produzidos pela execução das várias experiencias. Um aspeto considerado na apresentação das comparações dos dados obtidos foi a simplicidade do algoritmo, tanto a nível de quantidade de código necessário, como conhecimento da API e experiencia a programar e a detetar de erros.

Para o desenvolvimento da tese foram executados testes nas diferentes máquinas, referidas no capítulo 3.3.1. Os testes realizados foram efetuados a partir da execução dos ficheiros de extensão. ".jar" produzidos na realização de múltiplos *builds* do projeto. Cada um dos ficheiros representa uma invocação de um dos diferentes métodos com mecanismos de concorrência, nos três algoritmos ordenação. Os subcapítulos são apresentados pela designação do algoritmo de ordenação, nestes serão apresentados capítulos sobre os mecanismos utilizados, expondo os consumos de memória e tempo obtidos. As máquinas irão ser referidas pela identificação dos processadores Dual-Core, i5 e i7.

5.1 *Quicksort*

Uma vez que o algoritmo de ordenação *Quicksort* implementa o padrão *divide-and-conquer*, como foi apresentado no capítulo 3.1, este é um algoritmo que incorpora eficácia e eficiência. Uma vez que é um algoritmo que não implementa qualquer necessidade de sincronização e de interrupções, durante a espera de resultados de outros objetos, é possível rentabilizadar porções de tempo sobre algoritmos que requeiram estes procedimentos.

As seguintes implementações apresentam os dados obtidos da amostragem de testes definida, com apresentação das razões sobre os prejuízos e rendimentos encontrados através das comparações.

5.1.1 *Single Thread*

Os resultados da seguinte análise são referentes à execução do algoritmo *Quicksort* utilizando apenas a *Main Thread*, representando assim a execução de uma única *thread*. Ao invocar a ordenação tradicional *Quicksort* diretamente, foi possível estabelecer como case base, para este algoritmo, os resultados de tempo nas diferentes máquinas.

A partir de análise dos dados representados nas seguintes figuras [Figura 26, Figura 27 e Figura 28], é possível verificar a diferença nas velocidades dos processadores ao utilizar o mesmo algoritmo e, sem a utilização das tecnologias de avançadas destes. As imagens foram constituídas apenas pelos dados já calculados, pois a representação dos valores em cada teste da amostragem, levaria a um excesso de informação desnecessária.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	136	148	22,77	125,23	170,77
5000	501	504	5,06	498,94	509,06
10000	1075	1082	10,88	1071,12	1092,88
50000	6275	6295	22,27	6272,73	6317,27
100000	13323	13655	358,55	13296,45	14013,55

Figura 26 – Resultados da execução do teste *Quicksort Single Thread* na máquina Dual-Core

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	109	119	17,46	101,54	136,46
5000	344	356	5,31	350,69	361,31
10000	726	729	3,04	725,96	732,04
50000	4262	4272	25,81	4246,19	4297,81
100000	9095	9237	120,19	9116,81	9357,19

Figura 27 – Resultados da execução do teste *Quicksort Single Thread* na máquina i5

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	85	86	10,88	75,12	96,88
5000	305	306	2,78	303,22	308,78
10000	639	643	4,55	638,45	647,55
50000	3767	3840	66,04	3773,96	3906,04
100000	8029	8057	57,94	7999,06	8114,94

Figura 28 – Resultados da execução do teste *Quicksort Single Thread* na máquina i7

Como é possível verificar existe uma diferença entre as durações bastante evidente ao comparar os resultados em diferentes máquinas. Devido a isto a única comparação possível entre elas é a sua posição quando verificada a duração na mesma situação. Um exemplo evidente é o fato de quando comparadas as medianas da ordenação *Quicksort* no Dual-Core demonstrar sempre os piores resultados, no i7 os melhores e no i5 apresentar os resultados intermédios. Os valores apresentados nestes casos provam que a utilização de uma amostragem de sessenta testes é eficaz ao reduzir a margem de erro, atingindo valores inferiores a 10% da média correspondente.

Os dados obtidos em cada processador são utilizados ao longo da dissertação no auxílio à compreensão das vantagens dos mecanismos de concorrência.

5.1.2 *Threads*

O estudo de *threads* implementa o mesmo sistema de ordenação em cada uma das *threads* produzidas mas antes da criação destas é realizada a divisão do problema. A partir das threads é possível obter informações sobre as vantagens de produção destas em dimensões diferentes, utilizando o número de processadores lógicos ou físicos presentes nas máquinas. Com a análise dos resultados desta implementação, é possível obter dados sobre a utilização de sistemas *multithreading* e da utilização de *Hyper-threading*, e como estes influenciam a performance do sistema.

Uma vez que os dados dos testes são derivados da criação de um ficheiros do tipo Arquivo de Valores Separados por Vírgulas do Microsoft Excel (.csv), a apresentação do ficheiro não seria a mais adequada, para resolver foram elaborados ficheiros de resumo para cada um dos testes. No caso de estudo do algoritmo *Quicksort* com implementação de *threads* os valores máximos e mínimos foram removidos da apresentação pois, podem ser facilmente calculados pela adição ou subtração da margem de erro à média. Os valores de média, margem de erro, mediana, dimensão do *array* e o número de *threads* produzidas, são todos apresentados nos estudos.

Os resultados obtidos da execução dos testes no Dual-Core encontram-se ilustrados na Figura 29.

Dual-Core QuickSort		Número de threads Criadas						
Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	447	1118	730	1271	1075	1260	1404
	Média	647	1253	840	2199	1289	1308	1488
	Margem E.	107,54	267,71	72,37	615,13	211,03	71,86	96,66
5 mil	Mediana	673	859	967	1426	1494	1482	1616
	Média	681	847	1009	1499	1554	1561	1640
	Margem E.	27,58	29,86	76,67	99,44	77,18	81,48	45,55
10 mil	Mediana	1137	1487	2898	3244	3404	1846	1914
	Média	1151	1720	2945	3336	3473	1887	2006
	Margem E.	44,53	140,18	93,12	93,37	120,7	59,46	82,24
50 mil	Mediana	5139	5418	5432	5273	5133	5144	5510
	Média	5091	5398	5438	5527	5312	5275	5660
	Margem E.	215,33	224,44	199,14	235,58	184,97	184,21	209,51
100 mil	Mediana	11124	10947	10362	9684	10116	10861	10353
	Média	10757	11012	10729	9903	10479	10790	11228
	Margem E.	446,35	513,66	473,93	391,45	424,59	432,18	623,73

Figura 29 – Resultados da execução dos testes *Quicksort* com *threads* na máquina Dual-Core

Como é possível observar na imagem a dimensão da amostragem definida continua a demonstrar que foi uma boa escolha, uma vez que apenas num caso o valor de margem de erro é verificado com um valor elevado. Na Figura 29 pode-se verificar que a escolha de apresentar as medianas dos testes foi uma boa decisão pois os valores continuam de fato inferiores à média, demonstrando ser o melhor valor para comparações. Uma vez que os valores são apresentados

em microssegundos, a comparação de tempos pode ser compreendida como insignificante. Para resolver esse dilema, foi realizada a comparação de tempos em que foi escolhido como caso base o número de execuções ao longo de uma hora que é possível realizar com duas *threads*. A partir dos valores obtidos do caso base foi elabora a comparação com os restantes casos de número de *threads* de forma a compreender a implementação mais vantajosa num sistema empresarial. A comparação efetuada é ilustrada na Figura 30 e apresenta o número de minutos ganhos em verde, e os prejuízos a vermelho, da execução da ordenação de um *array* com a mesma dimensão mas com número de *threads* diferentes. Esta comparação é realizada através do uso das medianas obtidas, uma vez que estes valores representam as durações de execução intermedias.

Dual-Core	Número de threads Criadas						
Tamanho	2	3	4	5	6	7	8
mil	Casos Base	90,1	38,0	110,6	84,3	109,1	128,5
5 mil		16,6	26,2	67,1	73,2	72,1	84,1
10 mil		18,5	92,9	111,2	119,6	37,4	41,0
50 mil		3,3	3,4	1,6	-0,1	0,1	4,3
100 mil		-1,0	-4,1	-7,8	-5,4	-1,4	-4,2

Figura 30 – Análise da execução ao longo de uma hora do *Quicksort* com *threads* no Dual-Core

É possível verificar com os dados da Figura 30 que a utilização mais comum de se implementar, em que o número *threads* é igual ao número de processadores ou este número com a adição de um, é uma implementação bastante viável. Apesar de no caso da ordenação cem mil elementos o caso base e a utilização de mais uma *thread* demonstrarem os piores resultados, em relação às restantes. Os minutos possíveis de rentabilizar destes casos são muito inferiores quando comparados com os prejuízos nas restantes dimensões do *array*. No Dual-Core é inexistênte a presença de mecanismos como *Hyper-Threading* ou *Turbo Boost*, fatores por norma influenciam a execução dos processos do sistema. Quando colocadas as durações obtidas no Dual-Core para a execução do caso *Single Thread* com o caso base escolhido para a implementação de paralelismo, é possível entender a grande vantagem de sistemas *multithread*. A seguinte imagem [Figura 31] apresenta a comparação entre esses casos, apresentando a diferença em minutos, utilizando a comparação de execuções ao longo uma hora.

Tamanho	Single Thread	2 Threads	Comparação
mil	136	447	137,21
5 mil	501	673	20,60
10 mil	1075	1137	3,46
50 mil	6275	5139	-10,86
100 mil	13323	11124	-9,90

Figura 31 – Comparação das durações do *Quicksort* entre o *Single Thread* e 2 *threads* no Dual-Core

Como é possível verificar a partir da Figura 31, a utilização de threads torna-se rentável com o aumento da dimensão do problema. Um pormenor de importância é a diminuição do ganho de tempo quando existe no aumento final da dimensão do *array*. Esta redução de rendimento com o fato de na análise da Figura 30, a implementação com duas *threads* demonstrar prejuízos na comparação com as restantes é interessante. A razão da redução pode ser justificável, uma vez que no sistema, por muitas precauções que se tome, não se consegue garantir que em todos os testes os dados obtidos se referem ao melhor caso obtido. Os prejuízos nas dimensões mais reduzidas ao utilizar paralelismo ocorrem devido ao *overhead* de criação das *threads*. Uma vez que a ordenação com múltiplas *threads* requer cálculo extra mais o processo de criação dos objetos *thread*, o acumular destes tempos pode não se conseguir rentabilizar mesmo com duas *threads* a realizarem a ordenação em execução simultânea.

Uma vez analisados os dados no CPU Dual-Core, foi elaborado o estudo na máquina com um i5. Este foi realizado para medir a performance e os custos das mesmas implementações em máquinas que incorporassem tecnologias como *Hyper-Threading* e o *Turbo Boost*. Os CPUs i5 e i7 uma vez que apresentam ambas as tecnologias mas quantidades diferentes de cores, permitiram determinar conclusões sobre influência das tecnologias nos mecanismos de concorrência. Ao realizar os testes de análise de funcionamento de *threads* no processador i5 obteve-se um grande conjunto de dados. Estes foram expostos de forma a apresentar os resultados de forma a ser possível uma leitura clara. Este tratamento originou a Figura 32, onde são exibidas as durações da execução da ordenação *Quicksort* com múltiplas *threads*, no i5.

i5 QuickSort		Número de threads Criadas						
Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	357	502	469	390	363	442	481
	Média	395	593	494	432	392	468	529
	Margem E.	44,53	104,5	18,72	53,39	46,81	44,79	62,5
5 mil	Mediana	555	583	694	628	704	805	800
	Média	553	624	712	646	710	845	824
	Margem E.	24,04	60,48	34,16	33,65	27,83	51,37	36,44
10 mil	Mediana	852	900	1020	1003	1156	1301	1179
	Média	844	934	1059	1009	1188	1461	1213
	Margem E.	29,1	35,68	40,49	24,04	62,25	106,53	55,67
50 mil	Mediana	3835	4070	4096	3678	3372	3434	3757
	Média	3820	4107	4176	3766	3494	3648	4314
	Margem E.	142,46	173,58	186,23	164,47	133,1	176,11	423,33
100 mil	Mediana	8941	7771	8356	8218	6410	7173	6582
	Média	8982	7733	8257	8350	6591	7236	6875
	Margem E.	444,84	307,69	301,11	503,54	277,83	345,65	316,29

Figura 32 – Resultados da execução do teste *Quicksort* com *threads* na máquina i5

Ao analisar a Figura 32, é possível verificar que a regra de implementação de um número de *threads* igual ao número de processadores, ou esta quantia mais um, não se evidencia muito claramente. Nos casos de ordenação inferiores a cinquenta mil, as ordenações com quatro e cinco *threads* nunca apresentam durações inferiores à ordenação com duas *threads*. Isto deve-se principalmente a dois fatores, o primeiro reflete-se na ordenação ser tão rápida com um número menor de *threads*, tirando proveito do *Turbo Boost*, que nega os lucros possíveis da ordenação com um maior número de threads. O segundo é o fato do *Hyper-Threading* apesar

de permitir quatro *threads* no i5, estas executam em grupos de dois, alternando o tempo de execução entre elas de modo a completar as tarefas. Estas alternações são conhecidas por *context switch, e* podem levar a um *overhead* no consumo de tempo suficientemente grande que não possibilita a rentabilização de ordenação múltiplas ordenações, com porções mais pequenas do problema.

De modo a verificar com mais exatidão as rentabilizações das diferentes implementações foi elaborado a comparação de execução de ordenações ao longo de uma hora. Nesta comparação foram calculadas as quantidades de operações que o caso base escolhido, consegue elaborar durante uma hora. Nos restantes casos, são calculadas os tempos que demoram a executar o mesmo número de operações, e apresentam a diferença quando comparados com o caso base. O caso base escolhido foi o de utilização de quatro *threads* uma vez que a implementação mais casual é a de utilização do número de processadores lógicos na máquina. As diferenças de tempo são apresentadas em minutos e são possíveis de ser visualizadas na Figura 33.

i5 QuickSort	Número de threads Criadas						
Tamanho	2	3	4	5	6	7	8
mil	-14,3	4,2	Casos Base	-10,1	-13,6	-3,5	1,5
5 mil	-12,0	-9,6		-5,7	0,9	9,6	9,2
10 mil	-9,9	-7,1		-1,0	8,0	16,5	9,4
50 mil	-3,8	-0,4		-6,1	-10,6	-9,7	-5,0
100 mil	4,2	-4,2		-1,0	-14,0	-8,5	-12,7

Figura 33 – Análise da execução ao longo de uma hora do *Quicksort* com *threads* no i5

A Figura 33 indica que a regra, utilização de um número *threads* igual à quantia de processadores lógicos, não apresenta os melhores resultados. Já a segunda forma da regra, utilização de um número de *threads* igual ao número de processadores mais uma *thread*, apresenta valores mais benéficos ao longo de todas dimensões. Ao verificar os resultados ao utilizar cinco *threads*, é possível afirmar que apesar de em todas as situações apresentar resultados positivos, estes nunca foram os melhores. Um ponto interessante é o fato da utilização de duas *thread*, o valor de cores físicos, demonstrar resultados positivos nas primeiras quatro dimensões, e com o aumento destas, os lucros baixarem em simultâneo. Este fato indica que existe um ponto de separação entre os proveitos do *Hyper-Threading* e os do *Turbo Boost* na máquina, demonstrando a força do segundo nas ordenações baixas dimensões.

De forma a possibilitar a elaboração de conclusões sobre o mecanismo Thread, foi elaborado um estudo de comparação de execuções ao longo de uma hora. Neste foram realizadas as comparações, entre o caso base de utilização de *threads* para o i5, e os resultados obtidos da execução *Single Thread*. A Figura 34 apresenta em minutos os resultados obtidos das comparações.

Tamanho	Single Thread	4 Threads	Comparação
mil	109	469	198,17
5 mil	344	694	61,05
10 mil	726	1020	24,30
50 mil	4262	4096	-2,34
100 mil	9095	8356	-4,88

Figura 34 – Comparação das durações do *Quicksort* entre o *Single Thread* e 4 *threads* no i5

A Figura 34 demonstra que apesar da implementação de quatro *threads* não ser a mais eficiente das implementações com múltiplas *threads*, esta mesmo assim consegue superar por duas vezes a velocidade da versão *Single Thread*. Apesar da implementação de quatro *threads* apresentar lucros nas maiores dimensões dos *arrays*, estes são quase nulos quando comparados com os prejuízos nas restantes dimensões. Um ponto possível de se verificar é que, nas dimensões inferiores ou iguais a dez mil o caso de quatro *threads* demonstra prejuízos. Mas quando observada a Figura 33 é visível que nenhuma outra quantia de *threads* consegue superar esses prejuízos. Com este fato é possível afirmar que, apesar da implementação demonstrar resultados negativos nas três menores dimensões, nenhuma outra quantidade de *threads* na mesma situação apresentaria resultados positivos nestas dimensões.

A investigação no processador i7 representa a situação de uma ordenação com a criação de *threads* utilizando um mecanismo de grande capacidade. Esta investigação é de grande importância, para compreender qual a extensão dos possíveis proveitos ou carências do mecanismo. A investigação é exequível ao averiguar os resultados conseguidos da execução do teste em uma máquina que detenha um processador i7, correspondendo a um total possível de oito *threads*. A seguinte imagem [Figura 35] apresenta os dados recolhidos.

Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	221	333	342	401	454	548	601
	Média	273	422	343	412	469	579	619
	Margem E.	35,42	75,91	7,34	28,85	24,54	41,24	36,69
5 mil	Mediana	373	428	477	518	567	619	688
	Média	369	437	499	539	574	637	701
	Margem E.	12,15	13,66	36,18	28,85	18,47	25,56	34,41
10 mil	Mediana	726	832	888	816	910	957	902
	Média	713	833	934	845	936	993	951
	Margem E.	21,25	34,41	53,9	36,69	45,55	38,21	41,75
50 mil	Mediana	3213	3206	3259	3005	2825	2675	2959
	Média	3149	3285	3306	3013	2890	2752	3021
	Margem E.	140,69	133,1	142,71	145,5	148,78	143,98	153,85
100 mil	Mediana	6417	6190	7086	5812	5498	5246	5290
	Média	6454	6467	6904	6041	5764	5415	5522
	Margem E.	301,36	283,65	273,02	311,49	324,39	311,99	287,95

Figura 35 – Resultados da execução do teste *Quicksort* com *threads* na máquina i7

Ao observar a Figura 35, é possível averiguar que os testes continuam a manter margens de erro aceitáveis e os valores do cálculo das médias continuam superiores aos valores das medianas obtidas. Com o foco direcionado para as durações de ordenação de *arrays* de dimensão inferiores ou iguais a dez mil posições, é possível afirmar que apesar do processador possuir oito processadores lógicos, as durações mais longas são as dessa implementação. Isto deve-se a dimensão dos *arrays* ser demasiado pequena nesses testes, produzindo uma situação em que o *overhead* gerado na duração de criação de *threads,* e divisão de trabalho ultrapassa por completo o tempo que o sistema rentabilizou pela utilização máxima do CPU.

Uma vez que o mecanismo *Runtime* da API do Java considera o valor total de processadores lógicos como número de cores presentes. Para a comparação da execução ao longo de uma hora foi escolhido o caso de criação de oito *threads* como caso base. Os valores resultantes do cálculo das diferenças da execução encontram-se apresentados na Figura 36.

Tamanho	2	3	4	5	6	7	8
mil	-37,9	-26,8	-25,9	-20,0	-14,7	-5,3	
5 mil	-27,5	-22,7	-18,4	-14,8	-10,6	-6,0	Casos
10 mil	-11,7	-4,7	-0,9	-5,7	0,5	3,7	Base
50 mil	5,2	5,0	6,1	0,9	-2,7	-5,8	
100 mil	12,8	10,2	20,4	5,9	2,4	-0,5	

Figura 36 – Análise da execução ao longo de uma hora do *Quicksort* com *threads* no i7

Ao analisar a Figura 36 é possível verificar que de fato o caso de oito *threads* não é o mais adequado em *arrays* de dimensões inferiores ou iguais a dez mil. Um pormenor mais facilmente visível nesta ilustração é o fato que na maior parte dos restantes casos, os benefícios de tempo foram perdendo a sua força com o aumentar da dimensão do array, quando comparadas com o caso base. Ao analisar com cuidado os resultados respetivos aos casos de quatro *threads*, o valor de núcleos físicos, é afirmável que este caso não se destaca em relação aos restantes.

Para a comparação do caso de estudo de uma hora de execução com a implementação *Single Thread* foi optados os resultados obtidos da execução com oito *threads*. Esta preferência foi considerada a mais correta pois é o valor utilizado nas implementações em que se utiliza o valor de *Runtime* cores. A Figura 37 apresenta os resultados da comparação entre o *Single Thread* e o caso base de *threads* optado para o i7.

Tamanho	Sequencial	8 Threads		Comparação
mil	85	601		364,24
5 mil	305	688		75,34
10 mil	639	902		24,69
50 mil	3767	2959		-12,87
100 mil	8029	5290		-20,47

Figura 37 – Comparação das durações do *Quicksort* entre o *Single Thread* e 8 *threads* no i7

De acordo com a Figura 37 o caso *Single Thread* é o mais lucrativo quando comparada a duração para se obter o *array* devidamente ordenado, nas três menores dimensões. Ao analisar a figura é verificado que a duração do caso *Single Thread,* quando aumentada a dimensão do *array* de dez mil para cinquenta mil posições, aumenta em mais cinco vezes. Enquanto que no mesmo aumento de dimensão no caso de oito *threads* a duração aumenta pouco mais de três vezes. Estes valores indicam que o *overhead* da duração de criação e atribuição de trabalho às oito *threads* foi menor que a rentabilização obtida pela execução dessas *threads*.

Os algoritmos com a sua execução vão produzindo consumos de memória com a declaração de novos objetos, estes podem ser removidos pelo *garbage collector* antes do algoritmo terminar a sua execução. Para ser possível analisar as necessidades do mecanismos foram recolhidos os valores de consumo de memória, estes são a representação de ponto mais elevado de consumo durante a execução. Para o estudo do mecanismo *Thread* foram analisados os consumos das diferentes quantias de *threads* produzidas. A dimensão do *array* a ordenar foi estabelecida em cem milhões, para ser possível obter o máximo de valores de consumo pois a aplicação JProfiler só retorna o consumo a cada segundo decorrido. Uma vez que o consumo não varia com a execução em máquinas diferentes, com a mesma arquitetura, os testes de consumo de memória foram apenas realizados no Dual-Core. Os dados obtidos encontram-se presentes na Tabela 8, cada valor representam o consumo normal esperado, em megabytes.

Tabela 8 – Representação dos consumos de memória utilizando *threads* no *Quicksort*

1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
377,55	377,87	378,18	378,50	378,50	378,49	378,81	380,08

Uma vez que todas as máquinas representam sistemas de 64 bits, o *array* a ordenar consome 8 bytes para o objeto, mais 24 bytes de *overhead* mais a soma de 4 bytes por cada referência de elemento no *array*. Este cálculo origina um consumo aproximadamente de 381 MB, para o *array* de cem milhões de elementos, o que não e refletido no dados obtidos. Assim foi necessário considerar que os consumos referidos não representam uma verdade universal, mas uma aproximação do real o mais próximo possível. Com uma observação cuidada ao analisar os consumos ao longo da criação de novas *threads*, notou-se um aumento perto de 0.32 (MB). Este aumento de consumo encontra-se presente em quatro dos sete casos de incremento de número

de *threads*. Nos restantes casos o aumento ou não atingiu por completo a mesma quantia ou nem existiu, demonstrando o valor ser inalterado ou perto disso. Uma possível razão para o aumento não ser preciso em todas as situações deve-se ao fato da existência GC (*garbage colector*). Este ao longo da execução pode entrar em funcionamento e realizar a limpeza dos dados das *threads* que terminaram a sua execução, reduzindo o consumo de memória. Uma vez que o Dual-Core só garante um máximo de duas *threads* em execução simultânea, a criação de quatro ou mais *threads* pode já não refletir o *overhead* completa. Devido a este fato o valor mais próximo que foi possível concluir de *overhead* gerado, com a criação de *threads* na implementação *Quicksort* foi de 0.32 (MB) por *thread*.

Uma vez que a utilização do *Hyper-Threading* é um grande benéfico, para os restantes estudos casos de estudos, exceto casos de estudo do mecanismo *Thread*, foi estabelecido o uso padrão do valor de processadores lógicos presentes nos CPU.

5.1.3 *ExecutorService*

Para a demonstração de resultados do estudo dos múltiplos mecanismos de verificação de estado e das diferentes *threadpools* possíveis de implementar, não foram ilustrados todos os dados. Os dados de margem de erro e media não foram representados uma vez que estes, nesta fase de estudo só iriam provar que a dimensão de amostragem continua correta. Uma vez que existem um aglomerado número de dados a apresentar, estes foram recolhidos e tratados de forma a serem de fácil leitura. A Figura 38 apresenta os dados dos múltiplos mecanismos implementados em conjunto com o *ExecutorService*, na máquina Dual-Core. Os testes foram executados utilizando *threadpools* com duas *threads* atribuídas.

Parametros		CountdownLach			CompletionService			Boolean		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	978	944	893	930	1125	1258	845	740	883
	4	992	775	1445	1156	956	2273	757	726	1367
	8	658	631	1872	706	702	2966	683	724	1714
5mil	2	882	941	936	928	929	1487	1017	1011	1016
	4	843	986	1213	869	830	2267	982	964	1459
	8	797	793	1644	810	839	3348	980	935	2030
10 mil	2	1456	1300	1323	1265	1327	2517	1521	1501	1549
	4	1175	1160	1478	1154	1175	1652	1523	1472	1851
	8	1118	1139	2143	1045	1115	2052	1542	1479	2575
50 mil	2	5497	5726	5764	5585	5695	5556	6861	6811	6753
	4	4902	5194	5061	7475	4628	5927	6809	6718	7147
	8	4569	4942	5366	7488	6698	10117	6925	6724	7658
100 mil	2	11953	11386	11472	11400	10885	11930	14197	14091	13851
	4	9658	14473	9712	9467	9174	9604	14122	13918	14214
	8	8908	11292	9995	8624	8846	9600	14053	14233	14804
Total (Threadpool):		54386	61682	60317	58902	54924	72554	72817	72047	78871
Total (Mecanismo):		176385			186380			223735		

Figura 38 – Resultados dos testes *ExecutorService* com *Quicksort* máquina Dual-Core

Como é visível na Figura 38 foram adicionados dois novos cálculos a representação de dados. O Total (Threadpool) representa a soma de todos os valores do mecanismo em questão com uma *threadpool,* e o total (Mecanismo) apresenta o resultado da soma de todos os dados obtidos de um mecanismo independentemente. Com estes é possível ter mais conhecimento sobre dos dados obtidos. De forma a complementar o auxílio na análise foi utilizado um sistema de cores por linhas em que os valores variam de verde, os mais favoráveis, para vermelho, os de maior

duração. Este sistema de cores permite detetar com mais precisão as situações mais vantajosas e dispendiosas dos mecanismos e das *threadpools*. Para o estudo foram utilizados *tasks* consoante o tipo necessário para o mecanismo, submetendo as quantias de duas, quatro ou oito tasls para o executor, no caso do Dual-Core.

A análise de dados apresentou um conjunto elevado de situações mais prejudiciais nos casos em que foram utilizados booleanos como controlo de estado de *tasks*. Estes valores devem-se à porção de tempo que o sistema perde a verificar cada uma das *tasks*, mesmo sendo possível tirar proveito da execução de trabalho nas pausas da verificação. Um acontecimento interessante que é possível verificar é ao comparar a utilização de duas *tasks* com quantias superior, estas não apresentam os melhores valores. O resultado mais comum seria que duas *tasks*, cada uma a ser executada por uma *thread*, fosse o melhor caso uma vez que não existiriam pedidos à *queue* de trabalho extra e menos verificações de estado. Este acontecimento indica que uma *threadpool* tira maior proveito nestes casos de múltiplas porções de um problema de forma mais vantajosa que o mecanismo *Thread*. As situações mais vantajosas de acordo com estudo são as implementações *CountdownLach* e *CompletionService*. A utilização do *CountdownLach* demonstra resultados muito positivos apesar de não permitir a execução de trabalho extra durante a verificação de estados, ao contrário do *CompletionService.* Apesar da utilização do *CompletionService* demonstrar também bons resultados, a utilização deste demonstra durações bastante elevadas com a utilização da *Cached threadpool.*

De forma a possibilitar a comparação da execução das diferentes implementações ao longo de uma hora, foi escolhido como caso base a implementação do uso da Fixed *threadpool* com verificação de estados através de booleanos. Foi optada esta combinações pois são as implementações mais comuns e que requerem o menor conhecimento da API. Os resultados da comparação podem ser verificados na seguinte ilustração [Figura 39].

Parametros		CountdownLach			CompletionService			Boolean		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	9,4	7,0	3,4	6,0	19,9	29,3	Casos Base	-7,5	2,7
	4	18,6	1,4	54,5	31,6	15,8	120,2		-2,5	48,3
	8	-2,2	-4,6	104,5	2,0	1,7	200,6		3,6	90,6
5mil	2	-8,0	-4,5	-4,8	-5,3	-5,2	27,7		-0,4	-0,1
	4	-8,5	0,2	14,1	-6,9	-9,3	78,5		-1,1	29,1
	8	-11,2	-11,4	40,7	-10,4	-8,6	145,0		-2,8	64,3
10 mil	2	-2,6	-8,7	-7,8	-10,1	-7,7	39,3		-0,8	1,1
	4	-13,7	-14,3	-1,8	-14,5	-13,7	5,1		-2,0	12,9
	8	-16,5	-15,7	23,4	-19,3	-16,6	19,8		-2,5	40,2
50 mil	2	-11,9	-9,9	-9,6	-11,2	-10,2	-11,4		-0,4	-0,9
	4	-16,8	-14,2	-15,4	5,9	-19,2	-7,8		-0,8	3,0
	8	-20,4	-17,2	-13,5	4,9	-2,0	27,7		-1,7	6,4
100 mil	2	-9,5	-11,9	-11,5	-11,8	-14,0	-9,6		-0,4	-1,5
	4	-19,0	1,5	-18,7	-19,8	-21,0	-19,2		-0,9	0,4
	8	-22,0	-11,8	-17,3	-23,2	-22,2	-19,0		0,8	3,2

Figura 39 – Comparação das durações dos mecanismos *ExecutorService* com *Quicksort* no Dual-Core

A partir dos resultados obtidos das comparações é visível que o caso base nas ordenações das duas menores dimensões continua a ser uma implementação viável. Apesar de na implementação de booleanos com a *threadpool* de *WorkStealing* apresentar resultados positivos em grande parte das situações, os seus proveitos não são bastante elevados. A ordenação de um *array* de dimensão de 100 mil, utilizando quatro e oito *tasks* no sistema

Countdownlach com uma *threadpool* de *WorkStealing* apresenta valores contestáveis. Estes valores apresentam ser demasiado baixos quando comparados com os restantes casos, na mesma dimensão para o *array*, indicando a possibilidade que os seus dados foram influenciados pelo sistema. De forma provar esta afirmação foi elaborada a comparação das margens de erro, ilustrada na Tabela 9, das *threadpools Fixed* e *WorkStealing*, uma vez que os valores destes casos se encontravam aproximados até esse momento.

Tabela 9 – Margens de erro obtidas ao utilizar a *Fixed* e a *WorkStealing threadpool*

Nº Tasks	**Fixed**	**WorkStealing**
4	376,26	933,95
8	360,83	494,18

Como é possível verificar as margens de erro encontram-se muito mais altas na *threadpool* de *WorkStealing*, indicando que esta *threadpool* poderá ser capaz de apresentar valores muito mais lucrativos, com a possibilidade de ultrapassar os proveitos da *Fixed threadpool*.

Para apoiar a decisão de escolha de mecanismos a implementar foi também elaborada a comparação com o mecanismo *Thread*. De forma a elaborar a comparação foi realizado a análise de execução ao longo de uma hora. Como objetos de comparação foram escolhidas às implementações de duas *threads*, e a de oito *tasks* numa *Fixed threadpool* com mecanismo *CoundownLach*. Este último foi escolhido uma vez que é o que apresenta o melhor conjunto de durações, apresentando o menor número de casos de prejuízo e a melhor soma de durações. Esta comparação é possível ser verificada na Figura 40.

Tamanho	2 Threads	8 Tasks numa Fixed TP com CountDownL	Comparação
mil	447	658	28,32
5 mil	673	797	11,05
10 mil	1137	1118	-1,00
50 mil	5139	4569	-6,65
100 mil	11124	8908	-11,95

Figura 40 – Comparação das durações dos casos base *Quicksort Thread* e *ExecutorService* do Dual-Core

O estudo da API demonstrou benefícios ao possibilitar com uma implementação de maior complexidade uma rentabilização nas três dimensões superiores, sobre a implementação de uma *thread* por cada núcleo disponível. Se adicionados os aproximadamente 12 minutos adquiridos na dimensão de cem mil com, os aproximadamente 10, obtidos da comparação Single Thread com duas threads, é possível verificar benefícios iguais a um terço de uma hora de execução.

Com a continuidade do estudo foram executados os testes dos mecanismos definidos pelo *ExecutorService* no processador i5. Para ser possível apresentar os dados adequados ao teste foi necessário aumentar os valores de *tasks* a criar. Isto levou a um aglomerado número de valores

que dificultou a apresentação de resultados de forma clara e cuidada. De novo a análise dos resultados nesta máquina foi necessária para possibilitar a determinação de conclusões sobre as tecnologias presentes nos processadores mais modernos. Os resultados obtidos do estudo dos mecanismos definidos encontram-se presentes na Figura 41.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	467	609	452	478	621	468	447	692	462
	4	658	610	538	637	620	523	720	599	574
	8	633	631	618	572	699	662	630	646	563
	16	597	573	698	551	541	743	513	499	654
	32	605	522	906	548	623	1121	462	671	859
5 mil	2	533	650	566	633	697	648	561	691	545
	4	689	627	565	848	650	595	647	670	588
	8	648	636	658	628	678	621	617	657	624
	16	670	653	876	692	684	750	681	665	770
	32	888	643	1162	820	667	1287	876	1096	1290
10 mil	2	1093	1015	1039	1226	1022	1039	1265	1031	1316
	4	998	989	976	1147	963	1085	1126	1027	911
	8	1135	1050	860	1127	867	950	919	1024	873
	16	870	869	926	936	909	978	941	1043	909
	32	893	803	1128	948	1043	1196	927	981	1112
50 mil	2	4014	4166	4038	3847	4487	3799	4667	4935	4752
	4	3483	3384	3295	3590	3768	3388	3948	3815	3715
	8	2940	3345	2978	3082	2997	3072	3491	3534	3230
	16	2880	2831	2899	2971	2830	2932	3480	3475	3430
	32	2776	2736	3066	2827	2786	2931	3459	3416	3208
100 mil	2	7957	8229	8386	8443	8535	8088	9437	9015	9718
	4	6683	7057	6782	7127	6796	6529	8460	8169	7349
	8	5811	5631	5852	6295	6114	6149	6974	6690	7199
	16	5731	5442	6072	5728	5751	5821	6511	6400	6821
	32	5457	5375	5516	5974	5341	6133	6383	6537	6027
Total (Threadpool):		59109	59076	60852	61675	60689	61508	68142	67978	67499
Total (Mecanismo):		179037			183872			203619		

Figura 41 – Resultados dos testes *ExecutorService* com *Quicksort* máquina i5

Com uma inspeção dos resultados obtidos do cálculos dos totais dos mecanismos, é possível concluir que o mecanismo que nos permite obter maior rendimento, independentemente da *threadpool* utilizada, é o *CountdownLach*. Ao analisar os resultados dos totais das *threadpools* deste mecanismo foi concluído que existem grandes vantagens com a implementação do *CountdownLach*. Foi concluído este fato uma vez que duas das três *threadpools* definidas para este mecanismo, apresentam os melhores resultados quando comparadas com as restantes. A única *threadpool* com a implementação do *CountdownLach* que foi ultrapassada foi, a do tipo *Cached*. Esta demonstra um total superior a combinação da *threadpool* de *WorkStealing* no mecanismo *CompletionService*. As *threadpools* do tipo *WorkStealing* apresentam numa visão alargada um bom conjunto de resultados, independentemente do mecanismo de verificação de estados. Demonstrando uma forte constante a apresentar resultados superiores às *threadpools* do tipo *Fixed* e *Cached* em quase todos os casos.

Uma vez que não é muito simples de compreender os ganhos possíveis das implementações apresentadas na Figura 41, foram calculadas as diferenças de consumo de tempo ao longo de uma hora. Com estes cálculos é possível entender não só as vantagens sobre os diferentes mecanismos e *threadpools* mas também obter informações sobre a quantidade de *tasks* a produzir na resolução de problemas. Esta comparação permite também examinar qual dos casos apresenta mais constantemente resultados lucrativos. Os resultados obtidos destas comparações encontram-se apresentados na Figura 42.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	2,7	21,7	0,7	4,2	23,4	2,8	Casos Base	32,9	2,0
	4	-5,2	-9,2	-15,2	-6,9	-8,3	-16,4		-10,1	-12,2
	8	0,3	0,1	-1,1	-5,5	6,6	3,0		1,5	-6,4
	16	9,8	7,0	21,6	4,4	3,3	26,9		-1,6	16,5
	32	18,6	7,8	57,7	11,2	20,9	85,6		27,1	51,6
5 mil	2	-3,0	9,5	0,5	7,7	14,5	9,3		13,9	-1,7
	4	3,9	-1,9	-7,6	18,6	0,3	-4,8		2,1	-5,5
	8	3,0	1,8	4,0	1,1	5,9	0,4		3,9	0,7
	16	-1,0	-2,5	17,2	1,0	0,3	6,1		-1,4	7,8
	32	0,8	-16,0	19,6	-3,8	-14,3	28,2		15,1	28,4
10 mil	2	-8,2	-11,9	-10,7	-1,8	-11,5	-10,7		-11,1	2,4
	4	-6,8	-7,3	-8,0	1,1	-8,7	-2,2		-5,3	-11,5
	8	14,1	8,6	-3,9	13,6	-3,4	2,0		6,9	-3,0
	16	-4,5	-4,6	-1,0	-0,3	-2,0	2,4		6,5	-2,0
	32	-2,2	-8,0	13,0	1,4	7,5	17,4		3,5	12,0
50 mil	2	-8,4	-6,4	-8,1	-10,5	-2,3	-11,2		3,4	1,1
	4	-7,1	-8,6	-9,9	-5,4	-2,7	-8,5		-2,0	-3,5
	8	-9,5	-2,5	-8,8	-7,0	-8,5	-7,2		0,7	-4,5
	16	-10,3	-11,2	-10,0	-8,8	-11,2	-9,4		-0,1	-0,9
	32	-11,8	-12,5	-6,8	-11,0	-11,7	-9,2		-0,7	-4,4
100 mil	2	-9,4	-7,7	-6,7	-6,3	-5,7	-8,6		-2,7	1,8
	4	-12,6	-10,0	-11,9	-9,5	-11,8	-13,7		-2,1	-7,9
	8	-10,0	-11,6	-9,7	-5,8	-7,4	-7,1		-2,4	1,9
	16	-7,2	-9,9	-4,0	-7,2	-7,0	-6,4		-1,0	2,9
	32	-8,7	-9,5	-8,1	-3,8	-9,8	-2,3		1,4	-3,3

Figura 42 – Comparação das durações dos mecanismos *ExecutorService* com *Quicksort* no i5

Quando comparados os diferentes casos é visível que a implementação do caso base apresenta grandes quantidades de desfechos desfavoráveis. Ao analisar as implementações nas duas maiores dimensões a ordenar, é visível que qualquer caso, exceto os que utilização booleanos como mecanismo de estado, são lucrativos. Assim de forma a se prosseguir com a comparação dos mecanismos de concorrência foram ignorados os resultados dos booleanos. De seguida foram comparados os mecanismos *CountdownLach* e *CompletionService*. Ambos apresentam um bom conjunto de rendimentos, mas quando comparadas as quantidades de momentos resultados negativos são apresentados o *CompletionService* destaca-se ligeiramente mais. O *CompletionService* quando visto como um todo apresenta trinta situações negativas, enquanto o *CountdownLach* apresenta apenas vinte e três, demonstrando este ser a opção mais constante. Prosseguindo com a escolha da *threadpool* a utilizar no mecanismo escolhido, foram analisados as quantidades de resultados negativos em cada uma das *threadpools*. Este caso foi uma comparação complicada, uma vez que a *threadpool* de *WorkStealing* apresenta sete resultados negativos e as restantes apresentam oito. Sendo a diferença mínima, para se tomar a decisão mais correta possível, foram calculados os totais de prejuízo e de lucro. Este cálculo levou aos resultados apresentados na Tabela 10.

Tabela 10 – Cálculo possíveis rentabilizações no *Quicksort* com as três *threadpools*

	Fixed	WorkStealing	Cached
Rentabilizações (minutos)	125,9	151,3	131,5
Prejuízos (minutos)	53,2	56,5	134,3

Após observados os resultados da Tabela 10, foi escolhida a *threadpool* de *WorkStealing* como caso base, devido à sua quantidade de rentabilizações ser a mais elevada, e a de prejuízos ser das mais reduzidas. Para ser possível a comparação com o caso de *threads* foi necessário escolher uma quantia exata de *tasks* a submeter. A determinação da quantidade de *tasks* a implementar como caso base foi bastante simples, uma vez que apenas na situação de quatro *tasks* é que o caso base escolhido apresenta 100% de resultados positivos. A partir das escolhas tomadas foi possível comparar os diferentes mecanismos. No final desta comparação obteve-se os resultados apresentados na Figura 43.

Tamanho	4 Threads	4 Tasks,Tp Workstealing com ContdownLach	Comparação
mil	469	631	20,72
5 mil	694	627	-5,79
10 mil	1020	989	-1,82
50 mil	4096	3384	-10,43
100 mil	8356	7057	-9,33

Figura 43 – Comparação das durações dos casos base *Quicksort Thread* e *ExecutorService* do i5

Ao comparar os casos de estudo escolhidos é facilmente observáveis as vantagens da implementação do *ExecutorService* com os mecanismos *CountdownLach* e *threadpool* de *WorkStealing*. Esta combinação de mecanismo consegue garantir em quatro das cinco dimensões rendimentos vantajosos. Um ponto interessante visível nestes dados foi a inconstância dos rendimentos, pois ao contrário do esperado estes não aumentam exponencialmente com o aumento da dimensão. Uma das possíveis explicações esperada seria interrupções efetuadas por processos no sistema, mas estas seriam apresentadas nas margens de erro, e ao verificar estas nada de relevante foi encontrado. A explicação encontrada como mais correta derivou do estudo realizado ao longo da dissertação sobre a ordenação através do *Quicksort*. Este em nada garante que exista uma divisão de trabalho equilibrada entre as *tasks*, possibilitando casos em que um dos *cores* efetua operações pesadas, enquanto o segundo opera uma quantidade de tempo muito inferior ao primeiro núcleo. Mesmo com a utilização de *work-stealing* e 4 *tasks*, nunca é garantido que a divisão realizada seja duas *task* por *core*. Pode ocorrer situações em que um dos *cores* efetuar 3 *tasks* de curta duração enquanto o segundo *core* realiza uma *task* que ultrapassa a soma das durações de todas as restantes, como visível na Figura 24 e Tabela 7 do capítulo 4.7.

Para o estudo dos diferentes mecanismos de verificação de estado de *tasks* com diferentes *threadpools* no processador i7,foi produzida a Figura 44, onde está representada a informação obtida, com uma apresentação semelhante às anteriores.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	278	385	288	298	365	298	294	421	308
	4	403	561	336	426	570	350	443	657	408
	8	819	798	458	804	811	465	742	811	466
	16	765	780	547	741	775	547	713	829	569
	32	775	841	796	895	901	919	736	826	802
5 mil	2	403	463	421	496	481	431	413	491	424
	4	488	521	446	527	574	434	478	579	483
	8	802	738	531	807	829	521	795	891	591
	16	787	777	674	770	843	618	737	884	735
	32	1108	870	1101	997	843	878	1216	1009	1030
10 mil	2	775	677	749	774	706	770	858	774	739
	4	822	773	690	761	760	714	810	792	707
	8	1056	907	871	987	831	630	1100	976	796
	16	946	843	944	892	748	679	878	940	939
	32	930	980	1006	927	813	798	888	982	985
50 mil	2	3257	3230	3146	3145	3216	3221	3389	3331	3185
	4	2788	3090	2814	2680	2786	2635	3207	2782	2988
	8	2840	2681	2530	2319	2498	2098	3082	2725	2989
	16	2571	2486	2596	2268	2095	2124	2457	2564	2700
	32	2624	2445	2845	2166	1910	2144	2485	2646	2698
100mil	2	6589	6278	6825	6265	6763	6805	6311	6804	6840
	4	5459	5719	5435	5642	5418	5100	5592	5671	5658
	8	5248	4848	5399	4381	4481	4363	5446	5476	5308
	16	4833	4692	5025	4325	4099	4007	4756	4779	4912
	32	4749	4547	4739	4072	4125	4000	4550	4624	4804
Total (Threadpool):		52115	50930	51212	48365	48241	45549	52376	53264	52064
Total (Mecanismo):		154257			142155			157704		

Figura 44 – Resultados dos testes *ExecutorService* com *Quicksort* máquina i7

Apesar da dimensão de dados presentes na Figura 44 através do esquema de cores implementado, a leitura de informação consegue ser realizada de forma acessível. Ao analisar esta ilustração é possível verificar que em particamente nenhuma situação demonstrada o uso de oito *tasks* numa *threadpool* com oito *threads* é o caso mais benéfico. Através de múltiplos testes com variadas dimensões de problema é possível demonstra que não é possível determinar uma quantidade de tarefas a criar unicamente dimensão dos recursos da máquina. Um caso interessante de valores obtidos foi das implementações de *threadpools* do tipo *Cached*, pois apresentam totais muito lucrativos. De forma a investigar os resultados obtidos foram observadas as margens de erro de todas as restantes implementações. Nesta investigação nada de relevante foi encontrado, pois em nenhum caso foram encontrados valores de margem de erros que apresentassem ser elevadas ao ponto de demonstrar conclusões. Uma possível justificação é o fato da velocidade com que sistema resolve os problemas ser bastante elevada, e esta através da *Cached* threadpool não produzir recursos desnecessários. A *Cached threadpool* aumenta o seu nível de paralelismo à medida que vai necessitando, e reutiliza as *threads* previamente construídas assim que estas se encontrem disponíveis. Ao contrário das restantes *threadpools* que empregam oito *threads* logo à partida, está pode simplesmente requisitar um valor muito menor de *threads,* pois quando o trabalho é submetido para ser executado uma das *threads* já existentes pode ser reutilizada, assim reduzindo no tempo inicial de requisição de *threads*. O mecanismo que prova ser o mais benéfico ao longo do estudo continua a ser a implementação do *CompletionService*. Na execução no processador i7, este demonstra ainda mais força, uma vez que a execução de verificação de estado é quase certa de ser garantida como concluída a primeira verificação, devido à velocidade com que o CPU funciona. A vantagem na verificação de estado com *CompletionService* é que esta implementação não necessita de objetos de controlo de estado nem verificação deste. O oposto do *CountdownLach* que, necessita de reduzir o sinal com a finalização das tarefas até que este atinja o valor de 0, e

da verificação com booleanos que necessita da edição dos objetos de estado e da verificação constantemente destes.

De forma a completar o estudo dos mecanismos de concorrência a partir do *ExecutorService* no processador i7 foi elaborada a Figura 45, onde foi realizada a comparação de execuções ao longo de uma hora. Para este foi optado pelo mesmo caso base já definido ao longo da dissertação.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	-3,3	18,6	-1,2	0,8	14,5	0,8	Casos Base	25,9	2,9
	4	-5,4	16,0	-14,5	-2,3	17,2	-12,6		29,0	-4,7
	8	6,2	4,5	-23,0	5,0	5,6	-22,4		5,6	-22,3
	16	4,4	5,6	-14,0	2,4	5,2	-14,0		9,8	-12,1
	32	3,2	8,6	4,9	13,0	13,5	14,9		7,3	5,4
5 mil	2	-1,5	7,3	1,2	12,1	9,9	2,6		11,3	1,6
	4	1,3	5,4	-4,0	6,2	12,1	-5,5		12,7	0,6
	8	0,5	-4,3	-19,9	0,9	2,6	-20,7		7,2	-15,4
	16	4,1	3,3	-5,1	2,7	8,6	-9,7		12,0	-0,2
	32	-5,3	-17,1	-5,7	-10,8	-18,4	-16,7		-10,2	-9,2
10 mil	2	-5,8	-12,7	-7,6	-5,9	-10,6	-6,2		-5,9	-8,3
	4	0,9	-2,7	-8,9	-3,6	-3,7	-7,1		-1,3	-7,6
	8	-2,4	-10,5	-12,5	-6,2	-14,7	-25,6		-6,8	-16,6
	16	4,6	-2,4	4,5	1,0	-8,9	-13,6		4,2	4,2
	32	2,8	6,2	8,0	2,6	-5,1	-6,1		6,4	6,6
50 mil	2	-2,3	-2,8	-4,3	-4,3	-3,1	-3,0		-1,0	-3,6
	4	-7,8	-2,2	-7,4	-9,9	-7,9	-10,7		-8,0	-4,1
	8	-4,7	-7,8	-10,7	-14,9	-11,4	-19,2		-7,0	-1,8
	16	2,8	0,7	3,4	-4,6	-8,8	-8,1		2,6	5,9
	32	3,4	-1,0	8,7	-7,7	-13,9	-8,2		3,9	5,1
100mil	2	2,6	-0,3	4,9	-0,4	4,3	4,7		4,7	5,0
	4	-1,4	1,4	-1,7	0,5	-1,9	-5,3		0,8	0,7
	8	-2,2	-6,6	-0,5	-11,7	-10,6	-11,9		0,3	-1,5
	16	1,0	-0,8	3,4	-5,4	-8,3	-9,4		0,3	2,0
	32	2,6	0,0	2,5	-6,3	-5,6	-7,3		1,0	3,3

Figura 45 – Comparação das durações dos mecanismos *ExecutorService* com *Quicksort* no i7

Ao analisar os resultados obtidos na Figura 45, foi possível verificar que existe um aglomerado número de casos em que a junção do *CompletionService* com a *threadpool* do tipo *Cached* produz rentabilizações. Uma outro ocorrência sobre a junção dos mecanismos é grandeza dos rendimentos, chegando mesmo a produzir lucros superiores a 25 minutos. Afirmando assim que efetua em 35 minutos o mesmo trabalho que uma *threadpool* do tipo *Fixed* com um sistema de booleanos para verificar os estados das tarefas executa numa hora.

Tal como nos casos anteriores foram comparados os melhores casos de estudo do mecanismo em questão com o caso mecanismo de concorrência mais próximo. Neste caso foram comparados os resultados da implementação de oito *threads*, com o caso de estudo de trinta e duas *tasks* numa *Cached threadpool* com o mecanismo de verificação de estado *CompletionService*. Os resultados desta comparação podem ser verificados na Figura 46.

Tamanho	8 Threads	8 Tasks numa Cached TP com CompletionService	Comparação
mil	601	775	17,37
5 mil	688	1108	36,63
10 mil	902	930	1,86
50 mil	2959	2624	-6,79
100 mil	5290	4749	-6,14

Figura 46 – Comparação das durações dos casos base *Quicksort Thread* e *ExecutorService* do i7

Os resultados obtidos na Figura 46, indicam que o uso de threads é favorável nas três menores dimensões, e nas restantes é mais adequado o uso do caso base do *ExecutorService*. Nenhuns dos casos implementados com o *ExecutorService* demonstram valores que fossem capazes de competir com os resultados obtidos com o mecanismo *Thread* nas três primeiras dimensões. Nas dimensões superiores ao avaliar os resultados das restantes quantias de *threads* também nenhuma foi capaz de ultrapassar os lucos do caso base *ExecutorService*. Uma vez que nenhuma das implementações, independente do mecanismo, seria capas de modificar qual a melhor numa determinada dimensão, é seguro afirmar-se que os casos bases foram bem escolhidos e representam seguramente resultados confiáveis. Com esta análise, foi determinada que devido ao *overhead*, e a eficiência de cada mecanismo depender da dimensão inicial do problema, a abordagem da implementação de um algoritmo deve considerar ambas as implementações.

As diferentes implementações produzem consumos de memória diferentes, consoantes as suas necessidades como os mecanismos de verificação de estado e *threadpools*. De maneira a ser possível analisar as necessidades em cada situação, foram obtidos os consumos em cada implementação elaborada. Como parâmetros foram definidos a dimensão em cem milhões, pelas mesmas razões do estudo de consumos do mecanismo T*hread*, o número de *tasks* a submeter foi escolhido utilizar o valor de dois, pois a máquina onde foram realizados os testes de consumos de memora foi a do Dual-Core. Assim o estudo utilizará um valor de *tasks* igual ao valor de paralelismo disponível no processador. Os dados obtidos estão expostos em megabytes e podem ser visualizados na Tabela 11.

Tabela 11 – Representação dos consumos de memória utilizando *threadpools* no *Quicksort*

CountdownLach			**CompletionService**			**Booleano**		
Fixed	**WS**	**Cache**	**Fixed**	**WS**	**Cache**	**Fixed**	**WS**	**Cache**
377,88	377.23	379,50	378,18	378,87	378,18	894,20	1092,93	944,85

Os valores obtidos nos casos em que se utilizou o *CompletionService* e o *CountdownLach* apresentam valores bastante aproximados. O mesmo não se pode afirmar da utilização da implementação da verificação através de booleanos, esta apresenta um aumento de duas a quase três vezes superior ao consumo dos outros mecanismos. Tal acontece devido à necessidade das variáveis booleanas em cada objeto, e do armazenamento destes objetos num *ArrayList* de forma a ser possível aceder a cada um, para obter o seu estado. Para provar a existência de um grande consumo de parte do *ArrayList* foi obtida a seguinte imagem [Figura 47], que apresenta um dos consumos obtidos durante o teste com o sistema de verificação através de booleanos.

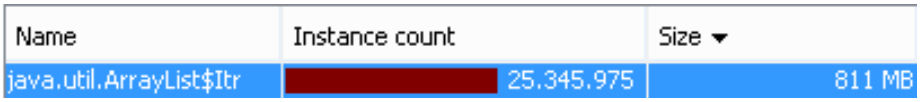

Figura 47 – Consumo de *ArrayList* na implementação *Quicksort ExecutorService*

Quando comparados os consumos com os da implementação de duas *threads*, 377,87 Megabytes, os valores não apresentam grandes discordâncias exceto quando utilizados booleanos de estado. Indicando assim que a utilização dos mecanismos de verificação de estado, exceto o de Booleanos, e o uso de *threadpools* apresentam resultados muito positivos na sua execução, devido à estabilidade no consumo de memória e redução nas durações de execução.

5.1.4 ***Fork/Join***

Como ultimo conjunto de testes para o algoritmo de ordenação *Quicksort* foi realizado um estudo dos mecanismos, *RecursiveAction* e da *threadpool* do tipo *ForkJoinPool*. A investigação destes mecanismos foi bastante simples, uma vez que a implementação não difere muito das restantes implementações com *threadpools*, sendo a principal variação a extensão da classe para as *tasks*. Para a execução dos testes no *Dual-Core* foi calculado o valor da *threshold* de criação de tarefas a partir definição da quantia de 1 MB na variável respetiva a cache do CPU. Estes resultados obtidos encontram-se apresentados na Figura 48, de um forma muito simples e semelhante à representação dos resultados do estudo da *Single Thread*, uma vez que é a maneira mais compreensível

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	792	1025	203,95	821,05	1228,95
5000	1200	1345	130,57	1214,43	1475,57
10000	1233	1436	96,66	1339,34	1532,66
50000	4351	5385	440,03	4944,97	5825,03
100000	8489	10708	1013,4	9694,59	11721,41

Figura 48 – Resultados do teste *Fork/Join Quicksort* no Dual-Core

Aproveitando a informação demonstrada na Figura 48, é possível afirmar que o uso da margem de erro demonstra à eficácia da amostragem, uma vez que o valor mais elevado apresentado não ultrapassa os 20% da média. Para finalizar o estudo do algoritmo *Quicksort* no processador Dual-Core, foram comparados os resultados de todos os casos base. Esta comparação é visível na seguinte Figura 49.

Tamanho	Single Thread	2 Threads	Executor Service	ForkJoin
mil	136	447	658	792
5 mil	501	673	797	1200
10 mil	1075	1137	1118	1233
50 mil	6275	5139	4569	4351
100 mil	13323	11124	8908	8489

Figura 49 – Comparação das medianas dos casos base *Quicksort* no Dual-Core

Como é possível observar a estrutura *Fork/Join* produz valores desvantajosos nas ordenações dos *arrays* de menor dimensão. Isto deve-se à estruturação definida para o cálculo de *threshold* não ser a mais eficiente em todos os momentos. Com esta análise e com consciência dos dados obtidos anteriormente no mesmo processador, é possível afirmar que nenhuma implementação

concorrente foi mais veloz que a versão Single Thread, nas dimensões entre mil a dez mil. Nas versões de dimensão superior o uso de ForkJoinPool e RecursiveAction de fato foi um benefício pois demonstraram os resultados mais lucrativos. Esta implementação tem grandes benefícios devido à utilização de *work-stealing* e *divide-and-conquer* em simultâneo, e da divisão do problema ser realizada não por quantidades de objetos de concorrência mas sim pelo cálculo do *threshold*.

Realizando o estudo *Fork/Join* no processador i5, com a definição de 3 MB de *cache* na fórmula de cálculo de *threshold*. Foi possível obter resultados sobre o funcionamento da *threadpool* do tipo *ForkJoinPool* e utilização de *RecursiveActions*, numa máquina com um número de *cores* físicos que Dual-Core mas com um total de quatro processadores lógicos. Através da execução dos testes foram obtidos os seguintes resultados apresentados na Figura 50.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	368	459	109,06	349,94	568,06
5000	941	1266	221,15	1044,85	1487,15
10000	1073	1131	96,91	1034,09	1227,91
50000	3053	3581	299,34	3281,66	3880,34
100000	5562	6236	425,35	5810,65	6661,35

Figura 50 – Resultados do teste *Fork/Join Quicksort* no i5

Os resultados obtidos da Figura 50 são valores interessantes para verificar a eficácia do uso de uma amostragem de sessenta testes, e verificar se ouve interferência do sistema dos resultados. Um exemplo de possível interferência é possível ser verificado nas dimensões de mil e cinco mil, pois apresentam margens um pouco elevadas. No caso da dimensão de mil, como referido no capítulo 3.1, é normal os primeiros testes produzam durações superiores, elevando as margens dos primeiros casos de teste. Ao verificar a dimensão de cinco mil, apesar da margem apresentar um porção mais reduzida esta continua elevada, nesta situação já é considerado que pode ter existido processos a causar interrupções. Estes resultados por si só representam uma pequena parte do estudo. De forma a dar continuidade à investigação, foram comparados estes resultados com os restantes casos base obtidos no processador i5. Esta análise é possível através da Figura 51, onde se encontram todas as medianas de duração.

Tamanho	Single Thread	4 Threads	ExecutorService	ForkJoin
mil	109	469	631	368
5 mil	344	694	627	941
10 mil	726	1020	989	1073
50 mil	4262	4096	3384	3053
100 mil	9095	8356	7057	5562

Figura 51 – Comparação das medianas dos casos base *Quicksort* no i5

De forma semelhante aos resultados no processador Dual-Core, os mecanismos de concorrência só provam a sua eficiencia a partir de dimensões iguais ou superiores a cinquenta mil. As vantagens da estrutura do mecanismo *Fork/Join* continuam a ser evidentes quando comparadas

as durações. No final é possível concluir que os melhores casos a implementar dependem bastante das dimensões dos problemas.

A partir da execução dos testes de *Fork/Join* do processador i7, foi possível obter os resultados apresentados na Figura 52. Para os testes nesta máquina, foi definido na fórmula de cálculo do *threshold* o valor de cache em 6 MB.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	238	305	84,26	220,74	389,26
5000	727	801	78,19	722,81	879,19
10000	1002	1038	55,92	982,08	1093,92
50000	1765	1975	196,1	1778,9	2171,1
100000	3430	4576	1324,13	3251,87	5900,13

Figura 52 – Resultados do teste *Fork/Join Quicksort* no i7

A partir dos resultados da execução do teste de ordenação *Quicksort* com *ForkJoinPool* e *RecursiveActions*, é possível observar pela primeira vez a execução da ordenação de um *array* com dimensão de cem mil posições, em menos de quatro mil microssegundos. Este fato torna a implementação muito benéfica quando comparada com as restantes. Para verificar esta afirmação foi produzida a seguinte ilustração de dados [Figura 53].

Tamanho	Single Thread	8 Threads	ExecutorService	ForkJoin
mil	85	601	775	238
5 mil	305	688	1108	727
10 mil	639	902	930	1002
50 mil	3767	2959	2624	1764
100 mil	8029	5290	4749	3430

Figura 53 – Comparação das medianas dos casos base *Quicksort* no i7

Como é possível verificar a versão *Single Thread* é a mais rentável de implementar nas versões de *arrays* de dimensão inferior a cinquenta mil. A cima dessa dimensão a *ForkJoinPool* torna-se de todas, a implementação mais lucrativa a nível de consumo de tempo. Este acontecimento é refletido ao longo do estudo do mecanismo, devidos às vantagens referidas na sua implementação.

Uma vez que os algoritmos requerem consumos de memória não é justo dizer que o mecanismo é o melhor, sem mostrar também as suas necessidades de memória. De modo a completar o estudo foi produzida a seguinte tabela [Tabela 12], onde são apresentados os consumos das implementações dos casos base escolhidos para a ordenação *Quicksort*.

Tabela 12 – Consumos de memória obtidos das implementações com *Quicksort*

Single Thread	**2 Threads**	**ExecutorService**	**FJ**
377,55(MB)	377,87(MB)	378,18 (MB)	382,29(MB)

A tabela apresenta os consumos máximos registados durante a execução de cada um dos casos. Nestes é possível observar que com as mesmas dimensões de *array*, apenas existe um aumento mínimo na implementação FJP, este incremento representa uma subida de 1,26% do consumo de memória. Este valor obtido quando comparado os consumos da versão *Fork/Join* com o da versão *Single Thread*. Nos consumos de tempos existe um rendimento superior a 25% quando comparados as durações da FJP com a segunda versão mais rápida, de acordo com a Figura 53. Com a análise efetuada é possível afirmar que o consumo de memória é facilmente suportado, em situações de execução em máquinas comuns nos tempos atuais, e rentável quanto ao lucro obtido nas durações de execução.

5.2 Resultados da ordenação em *Mergesort*

Nesta secção encontram-se apresentadas as análises efetuadas as implementações que utilizaram o algoritmo de ordenação *Mergesort*. Os resultados originados das análises podem ser estudados através da apresentação cuidada, realizada de forma semelhante ao estudo elaborado no capítulo 5.1. O algoritmo *Mergesort* é uma implementação que a partir do estudo efetuado no decorrer do desenvolvimento do projeto, demonstra ser de maior consumo de memória que o *Quicksort*. Com este fato é esperado que as comparações de rendimentos duração de execução, e consumos de memória apresentem resultados diferentes dos apresentados até o momento.

5.2.1 *Single Thread*

O estudo *Single Thread*, baseia-se na execução única do Código 7, apresentado no capítulo 4.1.2, pela *Main Thread*. Desta análise foi estabelecida toda a base de comparação com os restantes mecanismos implementados. Os dados obtidos da execução do algoritmo *Mergesort* encontram-se presentes nas figuras [Figura 54, Figura 55 e Figura 56].

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	324	353	43,78	309,22	396,78
5000	1137	1521	354,5	1166,5	1875,5
10000	2292	2590	287,7	2302,3	2877,7
50000	13881	14411	914,22	13496,79	15325,21
100000	25692	26290	329,45	25960,55	26619,45

Figura 54 – Resultados da execução do teste *Mergesort Single Thread* na máquina Dual-Core

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	259	284	28,34	255,66	312,34
5000	928	1011	67,81	943,19	1078,81
10000	1459	1511	66,8	1444,2	1577,8
50000	8330	9036	297,82	8738,18	9333,82
100000	17299	18577	581,47	17995,53	19158,47

Figura 55 – Resultados da execução do teste *Mergesort Single Thread* na máquina i5

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	318	315	19,23	295,77	334,23
5000	718	706	39,22	666,78	745,22
10000	1172	1234	54,4	1179,6	1288,4
50000	6912	7258	172,82	7085,18	7430,82
100000	14387	14810	253,79	14556,21	15063,79

Figura 56 – Resultados da execução do teste *Mergesort Single Thread* na máquina i7

Com base nas três figuras apresentadas, foi possível desenvolver o estudo dos mecanismos concorrência com uma ordenação *Mergesort*. Estes três casos representam as durações das ordenações utilizando apenas a *Main Thread* do Java. A partir destes valores foi possível estipular o tempo base que uma ordenação poderia demorar. Através das comparações efetuadas ao longo do documento entre estes valores com os diferentes mecanismos implementados, foi possível explorar os fatores que tornavam rentáveis ou prejudiciais os mecanismos de concorrencia.

5.2.2 *Threads*

Para obter conclusões sobre o mecanismo *Thread*, foi desenvolvido um estudo em é efetuada a divisão do *array* em porções, e cada uma destas irá ser resolvida por uma *thread*. Como referido no capítulo 4.3.1, esta implementação pode possibilitar a que uma *thread* divida o seu *array* a meio, e fique responsável por ordenar uma metade enquanto a segunda metade é ordenada por uma nova *thread*. A partir dos resultados obtidos foram retiradas conclusões sobre as vantagens desta implementação. Os dados finais do teste executados num Dual-Core encontram-se apresentados na Figura 57.

Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	1304	1044	889	1078	1239	1491	1624
	Média	1454	1401	1092	1155	1276	1526	1672
	Margem E.	213,06	221,15	149,54	103,74	44,03	45,55	47,57
5 mil	Mediana	1063	1361	1481	1703	1881	2091	2142
	Média	1067	1390	1518	2010	1942	2225	2203
	Margem E.	13,92	29,86	40,23	268,22	55,41	120,19	52,88
10 mil	Mediana	1791	1980	4286	4439	2821	2561	2679
	Média	2782	2326	4424	4436	3535	2676	2752
	Margem E.	473,18	171,3	125	369,94	313,76	119,18	55,67
50 mil	Mediana	7048	7474	7927	9746	8946	8559	8614
	Média	7199	7632	8319	10359	9215	9085	9763
	Margem E.	122,97	95,9	236,59	744,18	305,41	399,8	1166,74
100 mil	Mediana	16256	15201	16232	16025	16612	16559	16787
	Média	18231	16583	18452	18623	19302	19049	19287
	Margem E.	1218,11	799,34	1149,79	1946,85	1258,85	1281,62	1181,93

Figura 57 – Resultados da execução dos testes *Mergesort* com *threads* na máquina Dual-Core

Ao observar com atenção a Figura 57, é possível afirmar que a mediana do caso cem mil com duas *threads,* encontra-se mais próxima da mediana de duração da *Single Thread* a ordenar um

array de dimensão de cinquenta mil, do que a mediana obtida na dimensão de cem mil. Este fato indica que existe um grande benefício presente na implementação concorrente deste algoritmo de ordenação. Para compreender se esta vantagem se mantem, quando o *array* é dividido por uma quantia superior de *threads,* foi elaborada a Figura 58. Esta ilustração apresenta a diferenças de tempo de execução ao longo de uma hora de trabalho.

Tamanho	2	3	4	5	6	7	8
mil	Casos Base	-12,0	-19,1	-10,4	-3,0	8,6	14,7
5 mil		16,8	23,6	36,1	46,2	58,0	60,9
10 mil		6,3	83,6	88,7	34,5	25,8	29,7
50 mil		3,6	7,5	23,0	16,2	12,9	13,3
100 mil		-3,9	-0,1	-0,9	1,3	1,1	2,0

Figura 58 – Análise da execução ao longo de uma hora do *Mergesort* com *threads* no Dual-Core

Como é possível verificar na comparação de resultados, a utilização de um número de *threads* superior ao número de cores não é muito eficiente. Mesmo com as vantagens apresentadas nas ordenações de *arrays* com mil e cem mil elementos, nas implementações com o número de *threads* superior a dois, é possível refutar os seus resultados. As implementações com quatro e cinco *threads*, na ordenação de mil elementos obtiveram uma margem de erro muito inferior à de duas *threads*. Este fato leva a acreditar que os testes tiram um maior proveito do sistema após o arranque inicial ser realizado pelo estudo de duas *threads*. Na situação de ordenação de cem mil posições, o único valor que se destaca é na utilização de três *threads* pois este obteve um valor de margem de erro muito inferior aos restantes. É necessário afirmar que todas as implementações com o aumenta da dimensão do *array* vão diminuindo a diferença de durações em relação ao caso base. Na última dimensão é visível que as diferenças de duração variam desde o prejuízo de 2 minutos até atingir ganhos de aproximadamente 4. Diferenças que nem atingir 10% da duração de execução mas mesmo assim podem demonstrar o ponto neutro de execução onde o algoritmo e o mecanismo de concorrência. Onde ambos conseguem apresentar rendimentos muito semelhantes, independentemente do *overhead* gerado, e do rendimento obtido ao aplicar ordenações de pequenas porções do problema.

De modo a obter mais informações sobre a vantagem da implementação concorrente foram comparados os tempos de execução ao longo de uma hora dos principais casos. Esta comparação permite estabelecer uma base de possíveis rendimentos de tempo com a implementação de duas *threads*, quando comparada com a versão *Single Thread*. Os resultados desta análise encontram-se apresentados na Figura 59.

Tamanho	Single Thread	2 Threads	Comparação
mil	324	1304	181,48
5 mil	1137	1063	-3,91
10 mil	2292	1791	-13,12
50 mil	13881	7048	-29,54
100 mil	25692	16256	-22,04

Figura 59 – Comparação das durações do *Mergesort* entre o *Single Thread* e 2 *threads* no Dual-Core

Como é visível na comparação da Figura 59, a implementação concorrente é muito mais rápida do que a versão *Single Thread* em quatro das cinco dimensões. Apenas na primeira instancia a versão de *Single Thread* é superior, demonstrando assim que na menor dimensão, o tempo de ordenação com duas *threads* não compensa o *overhead* gerado pela criação das *threads*, mais a duração da ordenação. Um fato interessante é a descida de lucro na dimensão de cem mil inteiros, isto pode refletir que o tempo obtido na ordenação, nessa dimensão com duas *threads*, não é o mais otimizado e pode ter acontecido interrupções por outros processos.

Ao continuar o estudo do funcionamento das *threads* foram executados os testes de ordenação, através do *Mergesort*, com múltiplas *threads*, no processador i5. Com análise dos resultados dos testes foi esperado elaborar conclusões sobre o conjunto de resultados obtidos. Os resultados relativos ao estudo *Mergesort* num i5 encontram-se presentes na Figura 60. Está ilustração foi preparada de forma semelhante aos restantes estudos de *threads*, de forma a manter uma estrutura de fácil compreensão.

Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	541	595	586	579	548	580	736
	Média	644	710	638	658	572	617	748
	Margem E.	88,06	91,09	51,37	98,68	31,38	36,69	36,94
5 mil	Mediana	863	846	1022	799	982	1115	1072
	Média	856	884	1062	828	1000	1136	1073
	Margem E.	38,21	31,63	39,73	34,16	31,38	33,4	31,12
10 mil	Mediana	1316	1382	1749	1708	1981	2030	1998
	Média	1373	1505	1757	1771	2480	2352	2116
	Margem E.	44,03	104,76	32,89	88,82	338,31	223,43	131,58
50 mil	Mediana	6030	6224	6143	5353	4638	4525	4489
	Média	6414	6176	6094	5709	4742	4756	4654
	Margem E.	256,32	155,87	131,33	198,63	123,48	121,2	91,85
100 mil	Mediana	10466	11445	11835	10128	8373	8481	8628
	Média	11058	11514	11646	10339	8696	8796	8890
	Margem E.	332,99	296,56	272,01	155,87	170,55	179,91	144,74

Figura 60 – Resultados da execução dos testes *Mergesort* com *threads* na máquina i5

Ao verificar as diferentes medianas em cada dimensão, pode-se verificar que quanto maior a quantidade de *threads* criadas melhores os resultados, apesar de algumas exceções. Na dimensão de mil, em que a utilização de duas *threads* produz os resultados mais rapidamente,

na dimensão de cinco mil os rendimentos são superiores na dimensão de três *threads* e na dimensão de dez mil o melhor caso volta a ser representado por duas *threads*. A partir da dimensão de cinquenta mil, os melhores casos são representados pelas quantidades de *threads* superiores ou iguais a seis. Um pormenor que se destacou ao avaliar os resultados foi, na observação das margens de erro na dimensão de dez mil, nos casos superiores ou iguais a seis, estes apresentam valores de margem de erro mais elevados do que esperado. Mas uma vez que o valor de margem de erro reflete um intervalo, é possível que este valor não efetue grande impacto na comparação, pois a subtração da margem na media não reflete um valor próximo das medianas dos casos com melhores resultados.

Para garantir a compreensão dos lucros ou perdas com as diferentes quantias de *threads* a implementar, foi elaborado o estudo de comparação de execução de uma hora. Com apresentação das diferenças em minutos é possível entender o verdadeiro impacto da implementação em situações no funcionamento empresarial. As diferenças nos resultados estão apresentadas na Figura 61. Para o caso base de estudo foi escolhido de novo o valor do número de processadores lógicos presentes na máquina.

Tamanho	2	3	4	5	6	7	8
mil	-4,6	0,9		-0,7	-3,9	-0,6	15,4
5 mil	-9,3	-10,3	**Casos**	-13,1	-2,3	5,5	2,9
10 mil	-14,9	-12,6	**Base**	-1,4	8,0	9,6	8,5
50 mil	-1,1	0,8		-7,7	-14,7	-15,8	-16,2
100 mil	-6,9	-2,0		-8,7	-17,6	-17,0	-16,3

Figura 61 – Análise da execução ao longo de uma hora do *Mergesort* com *threads* no i5

Como é observável na Figura 61, ao utilizar um número de processadores lógicos os resultados obtidos não são os melhores, uma vez que as suas durações são superadas pelas implementações de duas e cinco *threads*. Com a utilização de duas ou cinco *threads* é obtido rendimento em todas as opções de dimensão, com a utilização de duas *threads* a produzir rendimentos superiores à utilização de cinco *threads*. As justificações para os ganhos destas versões são o fato de a implementação utilizar um esquema de divisão de trabalho bastante equilibrado, e do desenvolvimento do código para ser possível a utilização de um número ímpar de *threads*, como referido no capítulo 4.3.1.

Para obter informações sobre os rendimentos da implementação de múltiplas *threads*, foram comparados os resultados do caso base com os resultados da versão *Single Thread*. Este estudo é ilustrado pela Figura 62, onde é possível obter a diferença em minutos da execução de uma hora entre os dois casos.

Tamanho	Single Thread	4 Threads	Comparação
mil	259	586	75,75
5 mil	928	1022	6,08
10 mil	1459	1749	11,93
50 mil	8330	6143	-15,75
100 mil	17299	11835	-18,95

Figura 62 – Comparação das durações do *Mergesort* entre o *Single Thread* e 4 *threads* no i5

A implementação *Single Thread*, mais uma vez apresenta resultados superiores nas menores dimensões. A versão de quatro *threads* apenas consegue superar as durações de execução da versão *Single Thread* nas duas maiores dimensões, demonstrando o ponto onde a concorrência se torna bastante útil para este tipo de ordenações. Outro pormenor que pode ser observado ao verificar os resultados obtidos ao longo do estudo é que ambas as versões, de duas e cinco *threads*, produzem resultados superiores à versão *Single Thread*, exceto na dimensão de mil elementos.

Continuando com o estudo do funcionamento da ordenação *Mergesort* utilizando *threads*, foram executados os testes na máquina com o processador i7. Com esta análise foi possível obter um conjunto de resultados, que provem os benefícios nas implementações concorrentes, de forma semelhante aos casos deste estudo em máquinas diferentes. Com os dados obtidos do processador com maior número de núcleos lógicos, foi possível a pesquisa de indícios de rendimentos superiores. Estes dados encontram-se representados na Figura 63.

Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	221	333	342	401	454	548	601
	Média	273	422	343	412	469	579	619
	Margem E.	35,42	75,91	7,34	28,85	24,54	41,24	36,69
5 mil	Mediana	373	428	477	518	567	619	688
	Média	369	437	499	539	574	637	701
	Margem E.	12,15	13,66	36,18	28,85	18,47	25,56	34,41
10 mil	Mediana	726	832	888	816	910	957	902
	Média	713	833	934	845	936	993	951
	Margem E.	21,25	34,41	53,9	36,69	45,55	38,21	41,75
50 mil	Mediana	3213	3206	3259	3005	2825	2675	2959
	Média	3149	3285	3306	3013	2890	2752	3021
	Margem E.	140,69	133,1	142,71	145,5	148,78	143,98	153,85
100 mil	Mediana	6417	6190	7086	5812	5498	5246	5290
	Média	6454	6467	6904	6041	5764	5415	5522
	Margem E.	301,36	283,65	273,02	311,49	324,39	311,99	287,95

Figura 63 – Resultados da execução dos testes *Mergesort* com *threads* na máquina i7

Ao analisar a figura é possível afirmar que nas menores dimensões as quantias mais baixas de *threads* são mais rentáveis. Com o incrementar da dimensão do *array* as quantidades superiores

de *threads* vão se tornando as mais rentáveis mas nunca o valor de oito *threads* consegue atingir a melhor duração quando comparado com os restantes. Para melhor compreender o porque de tal ocorrer foi elaborada a comparação de resultados ao longo de uma hora, assim permitindo uma análise com maior profundidade do impacto das *threads* nas durações. Esta análise encontra-se apresentada na Figura 64, e para a produção desta foi escolhido o valor de oito *threads* para caso base, uma vez que este representa o número de processadores lógicos presentes na máquina.

Tamanho	2	3	4	5	6	7	8
mil	-37,9	-26,8	-25,9	-20,0	-14,7	-5,3	
5 mil	-27,5	-22,7	-18,4	-14,8	-10,6	-6,0	**Casos Base**
10 mil	-11,7	-4,7	-0,9	-5,7	0,5	3,7	
50 mil	5,2	5,0	6,1	0,9	-2,7	-5,8	
100 mil	12,8	10,2	20,4	5,9	2,4	-0,5	

Figura 64 – Análise da execução ao longo de uma hora do *Mergesort* com *threads* no i7

Como é verificável nas três menores dimensões a duração de execução com oito *threads* produz sempre resultados negativos, quando comparada com as quantias de menores de *threads*. Isto é um acontecimento recorrente neste estudo, variando nas dimensões o ponto de utilização onde se torna lucrativa a utilização máxima de processadores lógicos. Tal acontece devido ao *overhead* de tempo gerado pela criação de *threads*, o que é bastante visível quando observada a linha representante à dimensão dos mil elementos, uma vez que quantas mais *threads* são utilizadas, menor são os rendimentos. Com o aumento da dimensão a força da ordenação com oito *threads* vai elevando, mas até no maior caso de dimensão esta não se destaca com grande relevância.

Uma vez obtidos os dados das durações das múltiplas *threads,* foram comparados os rendimentos do caso base para o CPU i7 com a versão *Single Thread*. Esta comparação foi realizada com o caso de oito *threads*, apesar de este não representar o melhor caso mas sim o caso mais comum. Os resultados da comparação encontram-se disponíveis para melhor estudo na Figura 65, apresentando os rendimentos em minutos.

Tamanho	Single Thread	8 Threads	Comparação
mil	318	601	53,40
5 mil	718	688	-2,51
10 mil	1172	902	-13,82
50 mil	6912	2959	-34,31
100 mil	14387	5290	-37,94

Figura 65 – Comparação das durações do *Mergesort* entre o *Single Thread* e 8 *threads* no i7

Ao comparar o caso de oito *threads* para com a versão *Single Thread*, foi possível obter rendimentos bastante interessantes. Com a exceção da ordenação de *arrays* de mil posições, todos os resultados apontaram para que a implementação com múltiplas *threads* seja a mais benéfica de se utilizar. Isto deve-se ao algoritmo *Mergesort* realizar operações pesadas no sistema como cópia de valores e junções de resultados. Ao distribuir essas operações pelos diferentes processadores é permitido ultrapassar o *overhead* gerado pela criação de *threads*, com os rendimentos obtidos pela execução em paralelo. A partir da dimensão de cinquenta mil os rendimentos obtidos ao utilizar paralelismo, conseguem ultrapassar os 50%, permitindo executar a mesma resolução de problemas em metade do tempo.

De maneira a ser possível classificar a verdadeira força do paralelismo, foram executados testes de consumo de memória, sobre o algoritmo. Os resultados obtidos permitem comparar os rendimentos obtidos de tempo com os custos que produzem. A seguinte Tabela 13, apresenta os diferentes consumos em Megabytes, da ordenação em paralelo e utilização da Main Thread.

Tabela 13 – Representação dos consumos de memória utilizando *threads* no *Mergesort*

1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
0,934	1,927	2,199	2,343	2,325	2,356	1,617	1,571

Como é observável nos resultados obtidos, os consumos de memória são muito superiores ao algoritmo de ordenação *Quicksort*, devido à necessidade de cópias de *arrays* para divisão do problema. Quanto maior o número de divisões/*threads*, maior o número de cópias necessárias de executar, isto é verificável na Tabela 13 até serem criadas quatro *threads*. A partir da criação da quinta *thread* é verificado um período quase de estabilidade do consumo seguido por um decréscimo, isto deve-se ao momento em que novas divisões são executadas e o consumo deveria aumentar, as *threads* já existentes terminam e os seus consumos são limpos. Para provar esta afirmação foram retiradas informações durante a execução que, demonstram o ocurrimento de decréscimos de consumo de memória e a atividade do *Garbage Collector*. A seguinte imagem [Figura 66] ilustra a afirmação, e foi produzida através do estudo de execução de oito *threads* na ferramenta JProfiler.

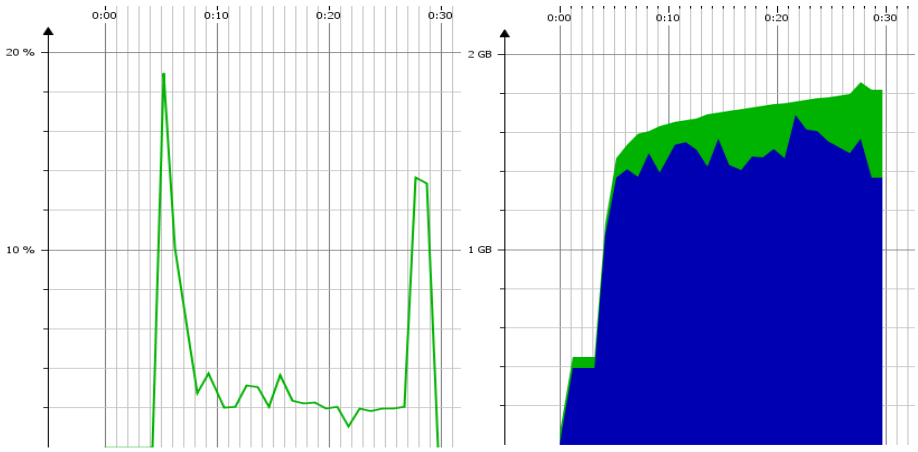

Figura 66 – Representação GC e consumo de memória

Os gráficos da Figura 66 apresentam informações sobre a JVM durante a execução do teste. No gráfico da esquerda é apresentado o funcionamento do GC, em que este apresentou um período de atividade durante toda a execução, diminuindo os custos de memória. Estes custos encontram-se apresentados no gráfico da direita, pela área azul e como verificados estes tendem a diminuir ou a aumentar consoante a atividade do GC.

5.2.3 *ExecutorService*

De forma a prosseguir no estudo foram realizados os testes aos múltiplos mecanismos de controlo de estado, e das diferentes *threadpools*. Com estes testes é possível comparar as diferentes durações de execução, das implementações que utilizam o algoritmo *MergeSort* e numa fase final comparar os custos de memória. Os resultados obtidos da execução dos testes no Dual-Core estão apresentados de forma cuidada na Figura 67, onde se encontram os dados separados por *threadpools* e mecanismos de verificação de estado.

Parametros		CountdownLach			CompletionService			Boolean		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	1145	1800	1647	1702	1822	1749	1125	860	1015
	4	1349	1485	1892	1377	1351	2005	1043	772	1452
	8	830	799	1824	928	842	1908	832	763	1752
5 mil	2	1306	1737	1265	1319	1265	1228	1796	1824	1843
	4	1389	1266	1765	1362	1366	1877	1695	1693	2276
	8	1400	1368	2422	1268	1236	2436	1651	1624	2820
10 mil	2	1814	1912	1897	2015	2008	2024	3134	3053	3203
	4	2030	2039	2454	1901	1951	2451	2874	2883	3373
	8	2079	2186	3168	1942	2155	3164	2688	2656	4009
50 mil	2	8137	8024	8171	8167	7898	8569	13933	14407	14862
	4	8991	8247	9828	9419	8150	8627	12653	13132	13693
	8	8436	8481	9646	9714	8395	8762	11847	12347	13678
100 mil	2	15178	15198	15387	15990	15700	14849	27389	27963	28358
	4	15810	15838	16365	15564	15483	15334	25478	26585	27326
	8	17021	18336	18383	16651	16532	17106	24303	24913	27625
Total (Threadpool):		86915	88716	96114	89319	86154	92089	132441	135475	147285
Total (Mecanismo):		271745			267562			415201		

Figura 67 – Resultados dos testes *ExecutorService* com *Mergesort* máquina Dual-Core

Com análises de resultados, foi possível concluir mais uma vez que a utilização de booleanos, para verificar estados, e *threadpools* do tipo *Cached* não são implementações muito rentáveis. Nos resultados obtidos da Figura 67 é visível que as implementações com o *CompletionService* e *CountdownLach* apresentam resultados muito lucrativos. De todas as implementações as que mais se destacam são a com a *threadpool* do tipo *Fixed* no CountdownLach, e a do tipo *WorkStealing* no *CompletionService*, ao analisar os totais de *threadpool*.

Uma vez que a Figura 67 não possibilita uma análise com grande facilidade foi produzido o estudo de comparação entre os diferentes mecanismos. Neste foram calculados os valores de execução ao longo de uma hora do caso base, e comparadas as execuções dos restantes mecanismos com esse caso. O caso base determinado foi a *Fixed threadpool* com verificação através de booleanos, utilizando o mesmo padrão de escolha de casos base já estabelecido. Os resultados de comparação deste estudo encontram-se presentes na Figura 68, ilustrando em minutos as diferenças obtidas.

Parametros		CountdownLach			CompletionService			Flag		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	1,1	36,0	27,8	30,8	37,2	33,3	Casos Base	-14,1	-5,9
	4	17,6	25,4	48,8	19,2	17,7	55,3		-15,6	23,5
	8	-0,1	-2,4	71,5	6,9	0,7	77,6		-5,0	66,3
5 mil	2	-16,4	-2,0	-17,7	-15,9	-17,7	-19,0		0,9	1,6
	4	-10,8	-15,2	2,5	-11,8	-11,6	6,4		-0,1	20,6
	8	-9,1	-10,3	28,0	-13,9	-15,1	28,5		-1,0	42,5
10 mil	2	-25,3	-23,4	-23,7	-21,4	-21,6	-21,3		-1,6	1,3
	4	-17,6	-17,4	-8,8	-20,3	-19,3	-8,8		0,2	10,4
	8	-13,6	-11,2	10,7	-16,7	-11,9	10,6		-0,7	29,5
50 mil	2	-25,0	-25,4	-24,8	-24,8	-26,0	-23,1		2,0	4,0
	4	-17,4	-20,9	-13,4	-15,3	-21,4	-19,1		2,3	4,9
	8	-17,3	-17,0	-11,1	-10,8	-17,5	-15,6		2,5	9,3
100 mil	2	-26,8	-26,7	-26,3	-25,0	-25,6	-27,5		1,3	2,1
	4	-22,8	-22,7	-21,5	-23,3	-23,5	-23,9		2,6	4,4
	8	-18,0	-14,7	-14,6	-18,9	-19,2	-17,8		1,5	8,2

Figura 68 – Comparação das durações dos mecanismos *ExecutorService* com *Mergesort* no Dual-Core

Como visível na Figura 68, a utilização do *CountdownLach* com uma *threadpool* do tipo *Fixed* ou de *WorkStealing* garante o maior número de situações com resultados positivos. A utilização da *threadpool* do tipo *Cached* continua a demonstrar bastantes momentos de prejuízos nos resultados, devido à sua inconstância da quantidade de *threads* atribuídas à *threadpool* durante a sua execução. Mesmo com esta desvantagem, a implementação da *threadpool* do tipo *Cached* apresenta grandes rendimentos em ordenações de *arrays* de dimensão de cinco e cem mil posições, utilizando duas *tasks*. Um acontecimento interessante é na utilização da *threadpool* do tipo *WorkStealing*, com booleanos a gerir os estados das *tasks*. Os resultados obtidos desta implementação apresentam benefícios semelhantes aos obtidos no estudo do algoritmo Quicksort no mesmo processador. Ao comparar a Figura 68 com a Figura 39, pode-se verificar que em ambos os estudos, apesar dos algoritmos de ordenação serem diferentes, os resultados apresentam se em grande porção das vezes, lucrativos ou prejudicais nos mesmos momentos. Apesar de da maior quantidade de resultados lucrativos pertencer à implementação *CountdownLach*, os maiores rendimentos apresentam-se na implementação da *threadpool* de *WorkStealing* com *CompletionService*. Devido a este fato, como caso base para os futuros testes do *ExecutorService* no Dual-Core foi escolhida a implementação *CompletionService* com uma *threadpool* do tipo *WorkStealing*. Uma vez que para elaborar uma comparação entre diferentes mecanismos de concorrência é necessário escolher uma quantidade de *tasks* específicas, foram analisados os rendimentos de cada situação. No final da comparação foi escolhido optar pela utilização de quatro *tasks*, pois apresentam a melhor média de rendimentos quando somados os diferentes resultados

Com análise de rendimentos dos múltiplos mecanismos no *ExecutorService*, foram desenvolvidas as comparações do caso base escolhidos entre do mecanismo *Threads* e do *ExecutorService*. Os resultados da execução ao longo de uma hora entre ambos os casos encontram-se apresentados na Figura 69, ilustrando as diferenças em minutos.

Tamanho	2 Threads	4 Tasks, TP de Workstealling com CompletionService	Comparação
mil	1304	1351	2,16
5 mil	1063	1366	17,10
10 mil	1791	1951	5,36
50 mil	7048	8150	9,38
100 mil	16256	15483	-2,85

Figura 69 – Comparação das durações dos casos base *Mergesort Thread* e *ExecutorService* do Dual-Core

Ao analisar os resultados tudo apresenta que a utilização de *threads* é uma implementação muito mais eficiente. Apenas na dimensão de cem mil posições é que a ordenação, utilizando os mecanismos escolhidos para o caso do *ExecutorService,* demonstra rendimentos mas muito reduzidos, apenas 3 minutos de lucro na execução de uma hora. Esta falta de eficiencia dos mecanismos *ExecutorService* deve-se ao aumento da complexidade do problema, na verificação de estados, e ao tratamento dos resultados para se obter a solução final. Uma vez que o *Mergesort* necessita da execução do *merge* de forma a obter resultados finais, e a divisão de trabalho é realizada de forma muito balanceada, a *Main Thread* pode não conseguir rentabilizar. Uma vez que o Dual-Core só permite a execução em simultâneo de duas *threads*, com a verificação de estados na *Main Thread,* e a execução de uma operação pesada como o *merge* em simultâneo com a execução da *threadpool*, esta pode levar a atrasos de *context switch*. Com a verificação de quatro *tasks* podem ocorrer três situações em que uma *thread* da *threadpool* é interrompida para a execução do *merge* de resultados.

De seguida é efetuado o mesmo estudo, mas agora realizado no processador i5. Os valores de duração, estão apresentados na Figura 70, organizados com a estrutura igual aos casos de estudo do *ExecutorService* em máquinas diferentes.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	792	942	673	769	1118	731	944	1249	946
	4	787	814	628	934	856	752	1037	1108	728
	8	818	764	755	670	879	752	720	1191	626
	16	752	706	819	637	663	795	708	642	809
	32	710	688	1049	806	826	1326	717	658	1041
5 mil	2	843	1069	866	1173	1135	1189	971	1123	1012
	4	857	1028	896	821	1008	868	943	1117	932
	8	951	1024	945	865	941	880	838	1005	875
	16	1048	1142	1154	986	1094	1087	1020	1075	1151
	32	1890	1737	1909	1674	1810	2171	1808	1771	1904
10 mil	2	1827	1878	1740	1949	1961	1846	2293	1866	1736
	4	1406	1466	1258	1510	1413	1509	1643	1621	1504
	8	1358	1461	1369	1241	1377	1283	1456	1377	1354
	16	1447	1537	1820	1447	1512	1796	1482	1427	1623
	32	2046	2138	2384	1963	2093	2097	1876	1641	2250
50 mil	2	8082	7059	7220	7202	7125	7050	7675	7698	7637
	4	6269	5206	4570	4585	4423	4457	6177	6229	6238
	8	4790	4961	4755	4444	4632	4475	5118	5188	4688
	16	5197	5235	5183	4928	5057	4948	4666	4904	4787
	32	6185	6125	6385	6136	6130	6054	5221	5855	5945
100 mil	2	13218	12712	12652	11686	13286	12831	13616	14282	13903
	4	8700	8896	8427	8465	8739	8341	12019	11732	9049
	8	9225	9558	9102	8937	9047	8495	11530	10039	9390
	16	10221	10446	10160	9988	10054	9511	9377	9437	9762
	32	12427	12275	12460	12265	12129	11833	10378	11788	12009
Total (Threadpool):		101846	100867	99179	96081	99308	97077	104233	106023	101899
Total (Mecanismo):		301892			292466			312155		

Figura 70 – Resultados dos testes *ExecutorService* com *Mergesort* máquina i5

Os resultados deste estudo demonstraram uma variação menos acentuada nas durações. Graças ao cálculo dos totais é possível concluir que o *CompletionService* apresenta o melhor conjunto de resultados e deste mecanismo a *threadpool* do tipo *Fixed* apresenta os mais rentáveis. Os valores mais interessantes são os das implementações da *threadpool* do tipo *Cached* ao demonstrar ser capaz de apresentar resultados lucrativos. Aos valores curiosos pode-se adicionar os casos de utilização de Booleanos, que apesar de continuarem a ser os piores casos, demonstraram uma aproximação das durações das restantes implementações. Para facilitar a conclusão de resultados foi executada a análise de execução de uma hora, aos resultados obtidos. O caso base determinado para este estudo foi a implementação de booleanos e uma *threadpool* do tipo *Fixed*. A análise das comparações é possível ser realizada através da Figura 71.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	-9,7	-0,1	-17,2	-11,1	11,1	-13,5		19,4	0,1
	4	-14,5	-12,9	-23,7	-6,0	-10,5	-16,5		4,1	-17,9
	8	8,2	3,7	2,9	-4,2	13,3	2,7		39,3	-7,8
	16	3,7	-0,2	9,4	-6,0	-3,8	7,4		-5,6	8,6
	32	-0,6	-2,4	27,8	7,4	9,1	51,0		-4,9	27,1
5 mil	2	-7,9	6,1	-6,5	12,5	10,1	13,5		9,4	2,5
	4	-5,5	5,4	-3,0	-7,8	4,1	-4,8		11,1	-0,7
	8	8,1	13,3	7,7	1,9	7,4	3,0		12,0	2,6
	16	1,6	7,2	7,9	-2,0	4,4	3,9		3,2	7,7
	32	2,7	-2,4	3,4	-4,4	0,1	12,0		-1,2	3,2
10 mil	2	-12,2	-10,9	-14,5	-9,0	-8,7	-11,7		-11,2	-14,6
	4	-8,7	-6,5	-14,1	-4,9	-8,4	-4,9		-0,8	-5,1
	8	-4,0	0,2	-3,6	-8,9	-3,3	-7,1		-3,3	-4,2
	16	-1,4	2,2	13,7	-1,4	1,2	12,7		-2,2	5,7
	32	5,4	8,4	16,2	2,8	6,9	7,1		-7,5	12,0
50 mil	2	3,2	-4,8	-3,6	-3,7	-4,3	-4,9		0,2	-0,3
	4	0,9	-9,4	-15,6	-15,5	-17,0	-16,7		0,5	0,6
	8	-3,8	-1,8	-4,3	-7,9	-5,7	-7,5		0,8	-5,0
	16	6,8	7,3	6,6	3,4	5,0	3,6		3,1	1,6
	32	11,1	10,4	13,4	10,5	10,4	9,6		7,3	8,3
100 mil	2	-1,8	-4,0	-4,2	-8,5	-1,5	-3,5		2,9	1,3
	4	-16,6	-15,6	-17,9	-17,7	-16,4	-18,4		-1,4	-14,8
	8	-12,0	-10,3	-12,6	-13,5	-12,9	-15,8		-7,8	-11,1
	16	5,4	6,8	5,0	3,9	4,3	0,9		0,4	2,5
	32	11,8	11,0	12,0	10,9	10,1	8,4		8,2	9,4

Casos Base

Figura 71 – Comparação das durações dos mecanismos *ExecutorService* com *Mergesort* no i5

Como esperado pelos resultados iniciais, existe uma quantia vasta de momentos de resultados positivos e negativos, que complicam a análise dos dados. Através da Figura 71 é possível verificar que a *Fixed threadpool* no *CompletionService*, para além de conter a melhor soma de resultados também é a situação que apresenta o maior número de resultados lucrativos. Esta apresenta resultados positivos em 68% das situações, ao verificar apenas os resultados das situações com quatro *tasks* é obtido 100% de resultados positivos. Com estes dados foi escolhido como implementação de caso base a combinação da *Fixed threadpool* com o *CompletionService* utilizando quatro *tasks*.

Com o caso base escolhido apenas foi necessário comparar os resultados deste com o caso base escolhido para o mecanismo *Thread*. Ao comparar os resultados de ambos, através da operação de comparação ao longo da execução de uma hora, foram obtidas as diferenças em minutos. Os dados obtidos no final da comparação podem ser analisados na Figura 72.

Tamanho	4 Threads	4 Tasks, Fixed Threadpool com CompletionService	Comparação
mil	586	934	35,63
5 mil	1022	821	-11,80
10 mil	1749	1510	-8,20
50 mil	6143	4585	-15,22
100 mil	11835	8465	-17,08

Figura 72 – Comparação das durações dos casos base *Mergesort Thread* e *ExecutorService* do i5

Com análise efetuada na Figura 72, os lucros da utilização do caso base do *ExecutorService* escolhidos são bastante óbvios, ao garantir lucros entre 8 a 18 minutos, em todas as dimensões menos na de mil elementos. Estes resultados promovem fortemente o uso dos mecanismos do *ExecutorService.* Este demonstra a sua força nos resultados através dos ganhos de tempo na criação de *tasks*, realizada mais rapidamente que a criação de *threads*, e na resolução do processo *merge* na *Main Thread* durante a execução da *threadpool* em simultâneo. Uma vez que só são produzidas quatro tasks, numa threadpool com quatro threads, assim que uma task conclua, a *thread* não terá trabalho a executar. Assim que duas *tasks* terminem, a *Main Thread* poderá executar com grande liberdade o *merge* dos resultados obtidos, pois com duas *threads* terminadas no CPU i5, duas novas podem realizar operações.

Na continuação do estudo foram desenvolvidos os testes aos mecanismos de concorrência com o *ExecutorService*, no processador i7. Os resultados obtidos dos testes encontram-se apresentados na Figura 73. Esta apresenta todos os resultados organizados pelos mecanismos, dimensões de problemas, quantidades de *tasks* criadas e os resultados das somas de tempos por mecanismos de verificação de estado e por *threadpools*.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	547	728	520	632	647	632	570	781	601
	4	519	663	491	444	632	444	529	708	464
	8	861	1028	558	547	907	547	916	1000	576
	16	825	1030	737	641	819	641	756	902	670
	32	898	1041	943	1029	887	1029	838	882	926
5 mil	2	694	633	599	606	630	606	586	624	575
	4	693	748	605	564	638	564	656	731	662
	8	1100	1053	844	748	915	748	988	981	787
	16	1173	1095	1144	983	1014	983	1041	1007	1096
	32	1874	1347	1731	1630	1365	1630	1369	1265	1480
10 mil	2	1168	997	1002	933	1035	933	961	992	941
	4	1071	946	857	791	840	791	883	924	890
	8	1442	1347	1239	1075	995	1075	1251	997	1082
	16	1318	1276	1494	1315	1036	1315	1121	1046	1231
	32	1659	1618	1864	1754	1455	1754	1407	1304	1656
50 mil	2	4537	4283	4328	4157	4165	4157	4292	4172	4198
	4	2899	2807	2867	2596	2715	2596	3328	3382	3283
	8	3120	2968	2941	2713	2649	2713	2845	2686	3074
	16	3642	3572	3730	3254	2859	3254	3138	2906	3438
	32	4717	4599	4962	4828	3904	4828	4205	3918	5060
100 mil	2	8165	8172	8054	8050	8173	8050	8166	8263	8106
	4	5309	5472	5187	5222	5303	5222	6718	6725	6714
	8	5564	5488	5701	5291	5130	5291	5350	5144	5265
	16	7110	6593	7062	6084	5448	6084	5904	5451	6372
	32	8824	8782	9571	8972	7627	8972	8208	7557	9593
Total (Threadpool):		69729	68286	69031	64859	61788	64859	66026	64348	68740
Total (Mecanismo):		207046			191506			199114		

Figura 73 – Resultados dos testes *ExecutorService* com *Mergesort* máquina i7

Ao prestar atenção aos esquemas de cores, é possível verificar uma grande concentração dos resultados mais positivos, demonstrados pela cor verde, nas colunas correspondentes ao *CompletionService*. Ao verificar os resultados totais é possível verificar que a rapidez global do mecanismo *CompletionService* quando comparado com os restantes mecanismos de estado, ao apresentar o resultado de somas mais baixa. Uma ocorrência que ainda não tinha sobressaído antes, é a soma dos resultados da verificação através de booleanos não apresentar o pior resultado. Isto acontece uma vez que ao utilizar o *CountdownLach* com a *threadpool* do tipo *Fixed* ou *WorkStealing*, foram ilustrados por várias vezes as piores durações. A razão deste prejuízo nas durações, segundo o estudo do funcionamento, reflete-se no fato do *CountdownLach* não permitir a execução de trabalho em paralelo com a *threadpool*. Assim o processo de *merge* de resultados só ser executado após todas as tarefas terminarem, podendo levar a desperdício de recursos do CPU. Devido a espera que as *tasks* terminem, a *Main Thread* não pode aproveitar a libertação de *threads* no CPU para executar os *merges* em simultâneo, ao contrário dos restantes mecanismos de verificação de estado.

De forma semelhante aos restantes estudos, foram efetuadas as comparações da execução de uma hora entre o caso base e todos os restantes de forma a determinar um valor em minutos de possíveis rendimentos ou prejuízos. Os resultados das comparações efetuadas encontram-se visíveis na Figura 74.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	-2,4	16,6	-5,3	6,5	8,1	6,5	Casos Base	22,2	3,3
	4	-1,1	15,2	-4,3	-9,6	11,7	-9,6		20,3	-7,4
	8	-3,6	7,3	-23,4	-24,2	-0,6	-24,2		5,5	-22,3
	16	5,5	21,7	-1,5	-9,1	5,0	-9,1		11,6	-6,8
	32	4,3	14,5	7,5	13,7	3,5	13,7		3,2	6,3
5 mil	2	11,1	4,8	1,3	2,0	4,5	2,0		3,9	-1,1
	4	3,4	8,4	-4,7	-8,4	-1,6	-8,4		6,9	0,5
	8	6,8	3,9	-8,7	-14,6	-4,4	-14,6		-0,4	-12,2
	16	7,6	3,1	5,9	-3,3	-1,6	-3,3		-2,0	3,2
	32	22,1	-1,0	15,9	11,4	-0,2	11,4		-4,6	4,9
10 mil	2	12,9	2,2	2,6	-1,7	4,6	-1,7		1,9	-1,2
	4	12,8	4,3	-1,8	-6,3	-2,9	-6,3		2,8	0,5
	8	9,2	4,6	-0,6	-8,4	-12,3	-8,4		-12,2	-8,1
	16	10,5	8,3	20,0	10,4	-4,5	10,4		-4,0	5,9
	32	10,7	9,0	19,5	14,8	2,0	14,8		-4,4	10,6
50 mil	2	3,4	-0,1	0,5	-1,9	-1,8	-1,9		-1,7	-1,3
	4	-7,7	-9,4	-8,3	-13,2	-11,1	-13,2		1,0	-0,8
	8	5,8	2,6	2,0	-2,8	-4,1	-2,8		-3,4	4,8
	16	9,6	8,3	11,3	2,2	-5,3	2,2		-4,4	5,7
	32	7,3	5,6	10,8	8,9	-4,3	8,9		-4,1	12,2
100 mil	2	0,0	0,0	-0,8	-0,9	0,1	-0,9		0,7	-0,4
	4	-12,6	-11,1	-13,7	-13,4	-12,6	-13,4		0,1	0,0
	8	2,4	1,5	3,9	-0,7	-2,5	-0,7		-2,3	-1,0
	16	12,3	7,0	11,8	1,8	-4,6	1,8		-4,6	4,8
	32	4,5	4,2	10,0	5,6	-4,2	5,6		-4,8	10,1

Figura 74 – Comparação das durações dos mecanismos *ExecutorService* com *Mergesort* no i7

Mais uma vez resultados lucrativos são visíveis em grande quantidade no *CompletionService*, apesar da distribuição ser bastante dispersa pelas três *threadpools*, a de *WorkStealing* apresenta a maior quantia de resultados positivos, e o melhor conjunto de resultados. Para determinar o melhor caso do *ExecutorService,* para ser possível elaborar a comparação entre diferentes mecanismos, foram analisados os resultados pelas quantias de *tasks* utilizadas. A única situação que provou 100% resultados positivos foi ao utilizar oito *task*, estabelecendo assim este conjunto de escolhas como o caso base do *ExecutorService*.

Para se obter informação sobre os diferentes mecanismos de concorrência foram comparados os diferentes casos escolhidos. Para a comparação foram consideradas as implementações de oito *threads,* e as de oito *tasks* numa *threadpool* de *WorkStealing* com a utilização de *CompletionService*. Os resultados obtidos representam os lucros ou prejuízos em minutos, ao longo da execução de uma hora, estes encontram-se presentes na Figura 75.

Tamanho	8 Threads	8 Tasks em WS threadpool com CompletionService	Comparação
mil	601	907	30,55
5 mil	688	915	19,80
10 mil	902	995	6,19
50 mil	2959	2649	-6,29
100 mil	5290	5130	-1,81

Figura 75 – Comparação das durações dos casos base *Mergesort Thread* e *ExecutorService* do i7

Como é possível verificar a utilização do caso escolhido para o ExecutorService, quando comparados as durações com a utilização de *threads,* este não garante benefícios relevantes. Ao verificar que nas menores dimensões garante prejuízos nas durações, desde os cinco minutos, até mesmo atingindo a meia hora de atraso. Com benefícios nas duas maiores dimensões que apenas rondam valores entre um e aproximadamente os sete minutos, a implementação com *threads* dá garantias de melhores tempos de resolução de problemas no formato mais generalizado. Isto deve-se ao fato da utilização de uma *threadpool* ser menos genérica que a utilização de threads. A divisão de um problema em oito partes é efetuada em ambos os mecanismos, o que difere as durações é método como os dados são tratados. No mecanismo *Thread* é aplicado o algoritmo *divde-and-conquer*, que permite que ao efetuar as mesmas divisões que os mecanismos no caso do *ExecutorService*. A diferença entre eles é o fato do algoritmo *divde-and-conquer* permitir que as primeiras ordenações assim que terminem elaborem o *merge* dos seus resultados, e que estes facilitem o trabalho futuro. Assim rentabilizando melhor do tempo de execução de ordenação sem necessidade de um mecanismo extra para verificar os estados.

De forma a completar o estudo dos mecanismos com no *ExecutorService*, foram analisados os consumos de memória utilizando *threadpools* de dimensão de dois, e um valor igual para o número de *tasks* a produzir, para ordenar um milhão de posições. Esta análise completa o estudo fornecendo a informação dos consumos de cada mecanismo. A informação é obtida da execução no Dual-Core, e é verificado em todas as instâncias o maior valor de consumo em toda a execução. A Tabela 14 apresenta os resultados obtidos do estudo em Gigabytes.

Tabela 14 – Representação dos consumos de memória utilizando *threadpools* no *Mergesort*

CountdownLach			**CompletionService**			**Booleano**		
Fixed	WS	Cache	Fixed	WS	Cache	Fixed	WS	Cache
2,055	1,967	1,822	1,904	1,917	1,931	1,999	1.990	2,034

Ao observar os resultados dos consumos de memória é possível verificar uma variação entre os valores de 0,013 aos 0.233 Gigabytes. Esta variação deve-se a aplicação JProfiller retornar os consumos a cada segundo desde o início do teste, o que causa a variação pois devido ao *divide-and-counquer,* o momento ilustrado pode não apresentar um instante com todas as divisões realizadas, e sem o GC ter efetuado limpeza de memória. Apesar desta variação, estes valores permitem a construção de uma base para as futuras análises e comparações entre os mecanismos implementados.

5.2.4 ***Fork/Join***

Para finalizar o estudo de ordenação *Mergesort* foi testado o último dos mecanismos de concorrência, onde foi realizado o estudo da estrutura *Fork/Join*. Neste foram criadas *task* do tipo *RecursiveTask*, em que estas retornam a sua porção da solução. Para dar início à análise foram realizados os testes do mecanismo no processador Dual-Core, os resultados da execução encontram-se ilustrados na Figura 76.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	1164	1554	295,29	1258,71	1849,29
5000	1722	2011	188,51	1822,49	2199,51
10000	2631	2754	124,75	2629,25	2878,75
50000	12589	13251	340,08	12910,92	13591,08
100000	26891	28027	796,3	27230,7	28823,3

Figura 76 – Resultados do teste *Fork/Join Mergesort* no Dual-Core

Apesar de ser possível verificar valores elevados de margem erro também, é possível confirmar que estes apenas correspondem a percentagens baixas quando comparadas com a média obtida. Com estes resultados fica completa a obtenção de consumos de tempo algoritmo *Mergesort* com os múltiplos mecanismos de concorrência, no processador Dual-Core.

Para completar o estudo foi elaborada a comparação dos múltiplos resultados escolhidos como casos bases, para este processador até o momento. A análise dos dados é possível, através da Figura 77, onde os dados recolhidos se encontram apresentados.

Tamanho	Single Thread	2 Threads	Executor Service	ForkJoin
mil	324	1304	1351	1164
5 mil	1137	1063	1366	1722
10 mil	2292	1791	1951	2631
50 mil	13881	7048	8150	12589
100 mil	25692	16256	15483	26891

Figura 77 – Comparação das medianas dos casos base *Mergesort* no Dual-Core

A Figura 77 apresenta a informação necessária para afirmar que a implementação *Single Thread*, para executar a ordenação, apresenta sem margem de dúvida o melhor resultado na dimensão

de mil posições. Nas dimensões entre cinco mil e cinquenta mil, inclusive, a utilização de duas *threads* demonstrou os melhores resultados. Com estes resultados é possível afirmar que no caso base do *ExecutorService* uma vez que é necessário gerir o estado das *tasks*, mesmo executando trabalho em simultâneo e utilizando *work-stealing*, o *overhead* gerado pela gestão de estado é demasiado elevado, quanto comparado com o de criação de *threads*. Na última dimensão o *ExecutorService* demonstra os melhores resultados. O caso base do *ExecutorService* elabora uma divisão por quatro *tasks*, o simples fato de serem realizadas mais ordenações mas de dimensões mais reduzidas pode ter impacto significativo nas durações quando comparadas com duas ordenações através de *threads*. A estrutura *Fork/Join* não apresenta resultados significativos, devido à sua necessidade de cálculo de *thresholds*, criando momentos de criação de grandes quantidades de *taks*, elaborando divisões desnecessárias, ou momentos um número reduzido de divisões.

Executando o mesmo teste na máquina correspondente ao i5 foi possível obter os seguintes resultados. O teste é semelhante ao executado no Dual-Core mas com a definição do valor da variável Cache em 3, os dados obtidos podem ser verificados na ilustração [Figura 78].

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	988	1517	537,95	979,05	2054,95
5000	1289	1375	92,86	1282,14	1467,86
10000	2009	2095	105,26	1989,74	2200,26
50000	8051	8635	279,1	8355,9	8914,1
100000	16602	18143	682,18	17460,82	18825,18

Figura 78 – Resultados do teste *Fork/Join Mergesort* no i5

Apesar da qualidade da informação apresentada na Figura 78, esta não é suficiente para fornecer as conclusões necessárias. Para ser possível elaborar conclusões relevantes foram comparadas as durações obtidas dos mecanismos no i5. Ao comparar os diferentes casos base determinados com a estrutura *Fork/Join* foi possível obter a seguinte Figura 79.

Tamanho	Single Thread	4 Threads	ExecutorService	ForkJoin
mil	259	586	934	988
5 mil	928	1022	821	1289
10 mil	1459	1749	1510	2009
50 mil	8330	6143	4585	8051
100 mil	17299	11835	8465	16602

Figura 79 – Comparação das medianas dos casos base *Mergesort* no i5

Ao verificar os dados obtidos é possível afirmar que na dimensão de mil a ordenação através da *Main Thread* continua a mais eficiente, mas nas restantes dimensões é visível uma grande rentabilidade na implementação do *ExecutorService*. Este acontecimento é oposto dos

resultados da máquina Dual-Core, a razão destes rendimentos baseia-se simplesmente no *hardware*. Uma vez que a máquina possui um maior número de cores, ignorando a velocidade, e uma vez que em ambos os casos foram elaboradas quatro *tasks*, a diferença está na eficácia do *HyperThreading*. Esta tecnologia permite que em grupos de dois, as *threads* elaborem *context-switching* até terminarem, o que não se reflete no Dual-Core onde as *threads* têm de terminar as suas tarefas. Apesar do *context-switching* ser uma operação pesada, a possibilidade de elaboração de uma maior quantidade de trabalho, em paralelo apresenta melhores resultados que a execução sequencial de várias tarefas. A utilização de *Fork/Join* continua a não demonstrar resultados nada satisfatórios devido à sua estrutura de divisão de trabalho, não ser específica o suficiente para uma implementação generalizada.

Por fim, foram obtidos os resultados da ordenação com *Fork/Join*, com o valor da variável Cache igual a 6, na máquina i7. Ao utilizar a *FoikJoinPool* com oito *threads,* foram obtidos os dados de duração de execução apresentados na Figura 80.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	376	447	87,3	359,7	534,3
5000	873	922	48,84	873,16	970,84
10000	1480	1537	56,93	1480,07	1593,93
50000	7131	7445	171,3	7273,7	7616,3
100000	14934	15888	399,29	15488,71	16287,29

Figura 80 – Resultados do teste *Fork/Join Mergesort* no i7

Os resultados obtidos foram expostos lado a lado com os restantes casos de estudo, executados na mesma máquina, de modo a possibilitar a análise de tempos dos diferentes mecanismos. Esta comparação é possível através da ilustração [Figura 81], onde todos os dados se encontram apresentados.

Tamanho	Single Thread	8 Threads	ExecutorService	ForkJoin
mil	318	601	907	376
5 mil	718	688	915	873
10 mil	1172	902	995	1480
50 mil	6912	2959	2649	7131
100 mil	14387	5290	5130	14934

Figura 81 – Comparação das medianas dos casos base *Mergesort* no i7

Mais uma vez a estrutura *Fork/Join* não conseguiu fornecer os resultados vantajosos em nenhuma das dimensões, e a versão *Single Thread* continua a ser a implementação na ordenação de mil inteiros que menos tempo necessita. Nas restantes dimensões é visível a aproximação entre a implementação com *threads* e o *ExecutorService*. Nas dimensões de cinco mil a dez mil, a utilização de *threads*, e o custo de overhead de tempo destas, demonstram ser mais velozes que a utilização de uma *threadpool* de dimensão oito, com o mesmo número de

tarefas. Nas dimensões superiores a dez mil, o *ExecutorService* torna-se o mais vantajoso, devido à possibilidade de execução dos *merge* em simultâneo com as *tasks*, uma vez que é utilizado o *CompletionService* no caso base. Ao possibilitar o *merge* durante as execuções das *task* problemas de maior dimensão apresentam melhores resultados uma vez que estes demoram mais a ser resolvidos. Uma vez que a duração da resolução do problema é maior, ao não acumular os *merges* para o final, é possível aproveitando a capacidade do CPU, utilizar *threads* livres para executar *merges* em paralelo com as *tasks*.

Com toda a informação sobre as diferentes durações dos mecanismos de concorrência, e das diferentes máquinas, é necessário para tomar uma boa decisão ter conhecimento dos consumos de memória, que acompanham os rendimentos. Para possibilitar a comparação de consumos memória, com as durações foram obtidos através do JProfiler os valores de maior ponto de consumo. Os dados obtidos foram obtidos da execução no Dual-Core, encontram-se presentes na Tabela 15, utilizando a unidade de medida Gigabytes.

Tabela 15 – Consumos de memória obtidos das implementações com *Mergesort*

Single Thread	2 Threads	ExecutorService	Fork/Join
0,935	1,927	1,917	1,7

Ao examinar os consumos de memória é evidente que os lucros obtidos nas durações pelos mecanismos de concorrência acartam grandes prejuízos nos consumos dos recursos. Apesar dos valores apresentarem apenas possíveis aproximações, é evidente que ao utilizar duas *threads* ou uma *threadpool* com essa quantia atribuída, os consumos duplicam em relação ao caso *Single Thread*. A estrutura *Fork/Join* utiliza uma disposição em forma de pirâmide com um grande número de níveis, em que os resultados são combinados de forma a criar o nível superior. Ao executar o *merge* das tarefas é possível que exista a limpeza dos dados duplicados, permitindo assim um menor consumo de memória que as restantes estruturas de concorrência.

5.3 Resultados da ordenação em *Pidgeonholesort*

Como terceira fase de testes foram estudados os mecanismos de concorrência utilizando algoritmo de ordenação *Pidgeonholesort*. Este é um algoritmo onde pouca informação sobre possíveis implementações paralelas foi encontrada. Com análise efetuada dos restantes algoritmos de ordenação, foi possível desenvolver os casos de estudo apresentados ao longo deste capítulo. Os resultados de cada mecanismo foram utilizados ao longo do estudo deste algoritmo de ordenação, elaborando comparações e conclusões sobre os consumos de recursos de tempo e memória.

5.3.1 *Single Thread*

Para iniciar o estudo foram elaborados as execuções da ordenação *Pidgeonholesort* nas três máquinas adquiridas para esta dissertação. Este estudo de ordenação foi elaborado seguindo a

estrutura de testes semelhante ao estudo *Single Thread* dos restantes algoritmos de ordenação. Os resultados estão ilustrados nas três seguintes figuras, Figura 82 para dados obtidos do processador Dual-Core, Figura 83 para os da máquina i5 e Figura 84 para os dados obtidos do CPU i7.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	31	104	32,14	71,86	136,14
5000	181	179	14,68	164,32	193,68
10000	298	276	19,48	256,52	295,48
50000	413	419	8,6	410,4	427,6
100000	823	846	30,87	815,13	876,87

Figura 82 – Resultados da execução do teste *Pidgeonholesort Single Thread* na máquina Dual-Core

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	91	112	20,24	91,76	132,24
5000	213	237	35,42	201,58	272,42
10000	76	128	90,33	37,67	218,33
50000	308	313	4,55	308,45	317,55
100000	650	728	84,26	643,74	812,26

Figura 83 – Resultados da execução do teste *Pidgeonholesort Single Thread* na máquina i5

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	49	58	11,13	46,87	69,13
5000	98	103	10,63	92,37	113,63
10000	47	47	0,25	46,75	47,25
50000	228	232	4,05	227,95	236,05
100000	460	461	2,53	458,47	463,53

Figura 84 – Resultados da execução do teste *Pidgeonholesort Single Thread* na máquina i7

Para além de tornar possível o avaliar do desempenho individual em diferentes máquinas, os dados obtidos fornecem pouco conteúdo. Ao analisar as figuras é facilmente verificado que o algoritmo de ordenação de fato é extremamente eficiente ordenando muito mais rapidamente que os restantes algoritmos de ordenação estudados. Esta velocidade pode ter impacto sobre os mecanismos de concorrência, e foi um fator foi considerado importante para o estudo. Um outro indício interessante observado foi a rapidez com que o Dual-Core ordenou nas menores dimensões, ao ordenar mais rapidamente que as restantes máquinas. Apesar de ser veloz nessa dimensão, ao verificar o valor máximo, que corresponde a uma estimativa do pior caso de ordenação, este ultrapassa os das restantes máquinas. Demonstrando que apesar da rapidez, é necessário prestar atenção à flutuação nas durações de modo a verificar a verdadeira eficiência.

5.3.2 *Threads*

Assim que foram obtidos os valores da execução do teste *Single Thread* foram elaborados os testes de ordenação utilizando diferentes quantidades de *threads*, nas três máquinas. Para iniciar o estudo do mecanismo *Thread* foram executados os testes no Dual-Core, dos quais foram possível obter os seguintes dados ilustrados na Figura 85.

Tamanho	Metricas	2	3	4	5	6	7	8
mil	Mediana	848	1287	755	812	969	1105	1266
	Média	848	1600	1203	889	1072	1199	1349
	Margem E.	117,66	223,18	262,9	92,1	79,96	66,04	65,03
5 mil	Mediana	394	568	733	840	986	1152	1346
	Média	405	604	784	882	1098	1239	1762
	Margem E.	23,53	40,49	54,4	42,26	68,32	66,55	306,17
10 mil	Mediana	492	673	805	936	1151	1242	2073
	Média	532	781	873	977	1436	1441	3139
	Margem E.	40,23	59,21	54,91	42	132,84	108,55	630,56
50 mil	Mediana	1237	1332	1471	1669	1811	1955	2387
	Média	1314	1368	1544	2200	1936	2092	2470
	Margem E.	74,39	66,04	61,23	311,49	87,55	95,9	130,31
100 mil	Mediana	1962	2147	2189	2751	2781	3203	3138
	Média	2048	2248	2333	2838	3417	3412	3374
	Margem E.	77,43	105,52	102,23	107,29	396,25	171,56	149,04

Figura 85 – Resultados da execução dos testes *Pidgeonholesort* com *threads* na máquina Dual-Core

Ao verificar as medianas obtidas, é possível verificar um aumento nestas com o incrementar do número de *threads*, assim indicando que o uso de um número de *thread* igual à quantia de processadores lógicos seria a mais opção correta. Ao observar as margens de erro na ordenação de mil inteiros com duas *threads*, é possível verificar um valor elevado, que auxilia a perceber o porque da duração de ordenação de mil posições ser mais longa que a de cinco mil. Como previsto os valores das primeiras ordenações são mais elevados. Mesmo utilizando uma amostragem de sessenta, se o algoritmo de ordenação for veloz, e se os aumentos das durações forem bastantes e com valores próximos, a variação pode não ser totalmente refletida na margem de erro. Uma vez realizada a análise inicial, foi elaborada a produção de comparações da execução ao longo de uma hora. Os resultados das comparações permitem determinar os possíveis minutos ganhos ou prejudicados, com cada quantidade de *threads*. De forma a obter os resultados foi utilizado o uso de duas *threads* como caso base. Os valores finais obtidos podem ser analisados na Figura 86.

Tamanho	2	3	4	5	6	7	8
mil		31,1	-6,6	-2,5	8,6	18,2	29,6
5 mil	Casos	26,5	51,6	67,9	90,2	115,4	145,0
10 mil	Base	22,1	38,2	54,1	80,4	91,5	192,8
50 mil		4,6	11,4	21,0	27,8	34,8	55,8
100 mil		5,7	6,9	24,1	25,0	38,0	36,0

Figura 86 – Análise da execução ao longo de uma hora do *Pidgeonholesort* com *threads* no Dual-Core

A Figura 86 apresenta resultados de prejuízo nos restantes casos quando comparados com o caso de duas *threads*, exceto na primeira dimensão. Na primeira dimensão tal não acontece devido à duração inicial com duas *threads*, assim permitindo duas situações em que a utilização de um número superior de *threads* torna-se mais eficiente. Esta eficiência é apenas por quantias de minutos tão baixas que, a utilização de apenas duas *threads* continua a ser mais eficiente como versão genérica. Uma razão da versão de duas *threads* permitir apenas nesta dimensão estes casos, é o fato de ser o primeiro teste a ser executado e como referido no capítulo 3.1, os primeiros são sempre prejudicados nas suas durações. Um outro detalhe, é o prejuízo apresentado com três *threads*, obtendo um valor desta dimensão que prova o ao impacto do sistema nas durações. Com o estudo dos resultados com múltiplas *threads*, foram comparados os resultados do caso base escolhido com os obtidos da utilização de apenas uma única *thread*. Esta análise permitiu verificar os rendimentos deste mecanismo de concorrência quando comparado com o caso mais simples de implementar. A Figura 87 apresenta os valores em questão, e os resultados das comparações efetuada apresentando em minutos as diferenças ao executar cada caso ao longo de uma hora.

Tamanho	Single Thread	2 Threads	Comparação
mil	31	848	1581,29
5 mil	181	394	70,61
10 mil	298	492	39,06
50 mil	413	1237	119,71
100 mil	823	1962	83,04

Figura 87 – Comparação das durações do *Pidgeonholesort* entre o *Single Thread* e 2 *threads* no Dual-Core

Ao verificar os resultados das comparações é possível afirmar que a implementação de *threads* só da indícios de prejuízos. O algoritmo de ordenação simplesmente é demasiado rápido, e a realização com o *overhead* de criação das *threads* simplesmente torna-se irrecuperável nestas dimensões. Ao verificar a mesma comparação nos restantes algoritmos de ordenação, *Quicksort* e *Mergesort*, é possível verificar que quanto mais rápido o algoritmo, menos lucros trás a o mecanismo *Thread*.

Efetuando os testes de ordenação com múltiplas *threads* mas agora no processador i5, foram possíveis obter as seguintes informações apresentadas na Figura 88.

Metricas	2	3	4	5	6	7	8
Mediana	400	400	499	415	485	572	452
Média	455	556	591	532	562	628	516
Margem E.	72,87	116,14	84,26	103,49	53,9	58,96	38,97
Mediana	235	345	301	327	386	430	478
Média	252	329	322	354	430	486	554
Margem E.	24,8	35,68	24,8	24,8	32,89	35,42	43,52
Mediana	227	414	408	538	595	671	703
Média	238	452	458	576	663	769	779
Margem E.	15,69	36,18	41,24	46,81	65,79	77,43	58,7
Mediana	698	802	848	847	941	1021	1031
Média	723	823	870	906	1033	1113	1164
Margem E.	39,73	35,68	43,27	54,4	67,56	68,07	93,37
Mediana	1042	1184	1186	1186	1386	1330	1408
Média	1084	1191	1226	1308	1502	1446	1510
Margem E.	49,59	48,33	55,41	77,43	88,56	70,09	71,36

Figura 88 – Resultados da execução dos testes *Pidgeonholesort* com *threads* na máquina i5

A Figura 88 prova que de fato uma vez que o algoritmo é muito eficiente quanto às durações de execução, os resultados com menor número de paralelismo apresenta os melhores resultados. Num processador como i5 em que existem dois cores que permitem a execução de quatro *threads* aplicando *context-switching,* a utilização da regra de criação de *threads* igual à quantia de processadores lógicos, apenas iria contribuir em desperdícios de tempo. Em todas as instâncias de dimensão a implementação que apresentou o melhor resultado foi a de duas *threads,* com apenas uma exceção que igualou a sua duração. Apesar de o lucro nestas dimensões estar alocado na menor quantidade de *threads*, com o aumento da dimensão, as durações nas restantes quantias de *threads* foram melhorando. Com a diminuição dos rendimentos nas menores quantidades de *threads*, existe a possibilidade de que num estudo mais alargado de dimensões, as maiores quantias de *threads* poderiam vir a ultrapassar os rendimentos da utilização de duas *threads*.

Para provar as conclusões tiradas, de uma forma simples de compreender, foi elaborado o estudo de execução ao longo de uma hora entre o caso base normal do i5 e as restantes situações. Os resultados das comparações estão apresentados em minutos na Figura 89.

Tamanho	2	3	4	5	6	7	8
mil	-11,9	-11,9	Casos	-10,1	-1,7	8,8	-5,7
5 mil	-13,2	8,8	Base	5,2	16,9	25,7	35,3
10 mil	-26,6	0,9		19,1	27,5	38,7	43,4
50 mil	-10,6	-3,3		-0,1	6,6	12,2	12,9
100 mil	-7,3	-0,1		0,0	10,1	7,3	11,2

Figura 89 – Análise da execução ao longo de uma hora do *Pidgeonholesort* com *threads* no i5

Ao avaliar os resultados da utilização de duas *threads*, é óbvio que é possível obter uma boa quantia de minutos, em vez da implementação com a utilização das regras de definição de

paralelismo mais comuns. Da Figura 89 é necessário sublinhar que, na primeira dimensão, é possível obter lucros com a utilização de cinco, seis e oito *threads*. Estes lucos apesar de existirem são uma inconstante, uma vez que o algoritmo é muito veloz qualquer diferença de microssegundo poder representar uma diferença poderosa ao longo de uma hora e uma vez. Um fator que pode levar a situações semelhantes, em que a margem não apresenta a resposta por completo ao porque deste acontecimentos, é o fator final do algoritmo em que são verificadas os *arrays* de pombos. Estes têm uma dimensão igual à diferença entre o menor e o maior valor presentes no *array* inicial, levando a *arrays* de grande ou pequena dimensão. A dimensão do *array* de pombos a percorrer causa impactos diferentes no sistema e na duração da ordenação em cada teste, uma vez que novos *arrays* iniciais são gerados para cada teste da amostragem.

Com a compreensão dos resultados obtidos, o processo final é iniciado comparando os resultados obtidos do caso base do estudo de *threads* com os resultados obtidos do estudo *Single Thread*. A comparação é realizada através do estudo de funcionamento ao longo de uma hora, apresentando em minutos as diferenças na Figura 90.

Tamanho	Single Thread	4 Threads	Comparação
mil	91	499	269,01
5 mil	213	301	24,79
10 mil	76	408	262,11
50 mil	308	848	105,19
100 mil	650	1186	49,48

Figura 90 – Comparação das durações do *Pidgeonholesort* entre o *Single Thread* e 4 *threads* no i5

A Figura 90 apresenta resultados negativos ao utilizar quatro *threads*. O algoritmo de ordenação *Pidgeonholesort* simplesmente apresenta durações de execução muito baixas. A eficiência do algoritmo cria um problema para a concorrência uma vez que esta gera o seu próprio *overhead* de tempo que, simplesmente não consegue fornecer os rendimentos necessários para se tornar rentável.

A Figura 91 apresenta os resultados obtidos do estudo do mecanismo *Thread* com o algoritmo de ordenação *Pidgeonholesort*.

Metricas	2	3	4	5	6	7	8
Mediana	280	336	423	446	487	553	643
Média	287	419	477	495	527	590	676
Margem E.	41,24	75,4	63,51	45,55	35,17	26,06	26,06
Mediana	176	247	320	398	482	577	646
Média	181	258	339	439	512	594	680
Margem E.	9,62	13,66	19,74	37,7	25,56	24,04	26,82
Mediana	214	290	365	426	529	626	657
Média	239	310	391	448	567	679	694
Margem E.	31,63	21	22,01	22,01	28,85	45,04	27,07
Mediana	503	546	578	649	714	785	877
Média	522	581	625	686	762	840	946
Margem E.	27,58	42	40,23	33,15	50,1	37,2	54,15
Mediana	753	820	871	918	996	1103	1148
Média	813	852	912	994	1077	1184	1222
Margem E.	42,26	38,21	41,24	56,68	57,69	71,36	49,85

Figura 91 – Resultados da execução dos testes *Pidgeonholesort* com *threads* na máquina i7

Mais uma vez a utilização de apenas duas *threads* apresenta os melhores resultados, com estes dados é definitivamente provado que a utilização de *threads* com algoritmos que terminem a sua execução rapidamente, não é uma implementação eficiente. A utilização de *threads* deve ser elaborada com muito cuidado, e ter em atenção que apesar dos ganhos possíveis, esta apenas apresenta rendimentos desejados se existir períodos de tempo onde se possa capitalizar.

Elaborando o mesmo estudo de funcionamento de execução de uma hora, e comparando o caso base num processador de oito processadores lógicos, com as restantes quantidades de *threads* possíveis de implementar, foi produzida a Figura 92. Esta apresenta em minutos as quantidades de tempo que diferenciam as implementações.

Tamanho	2	3	4	5	6	7	8
mil	-33,9	-28,6	-20,5	-18,4	-14,6	-8,4	
5 mil	-43,7	-37,1	-30,3	-23,0	-15,2	-6,4	
10 mil	-40,5	-33,5	-26,7	-21,1	-11,7	-2,8	Casos Base
50 mil	-25,6	-22,6	-20,5	-15,6	-11,2	-6,3	
100 mil	-20,6	-17,1	-14,5	-12,0	-7,9	-2,4	

Figura 92 – Análise da execução ao longo de uma hora do *Pidgeonholesort* com *threads* no i7

De acordo com as conclusões tiradas até o momento, a utilização de um alto nível de paralelismo não é muito eficiente. Sem experimentar é impossível dizer quando é que a utilização do caso

base vai apresentar os resultados mais positivos. A medida que os rendimentos ultrapassarem o *overhead* de criação das *threads,* as quantias de *threads* mais baixas irão perder sequencialmente para a implementação com uma quantia superior.

Para finalizar as conclusões sobre os consumos de tempo das implementações com *threads*, foram comparadas as durações de uma hora entre o caso base de oito *threads,* com a versão *Single Thread*. Os resultados desta comparação estão apresentados na Figura 93, ilustrando em minutos os prejuízos ou ganhos possíveis ao utilizar oito *threads*.

Tamanho	Sequencial	8 Threads	Comparação
mil	49	643	727,35
5 mil	98	646	335,51
10 mil	47	657	778,72
50 mil	228	877	170,79
100 mil	460	1148	89,74

Figura 93 – Comparação das durações do *Pidgeonholesort* entre o *Single Thread* e 8 *threads* no i7

Tal como nas restantes máquinas, o nível de paralelismo do caso base provou demonstrar resultados prejudiciais. Mesmo utilizando a versão de duas *threads* que fornece um lucro de 20 a 40 minutos dependendo da dimensão, esta não seria capaz de fornecer tempos mais vantajosos que a versão *Single Thread*. Assim provando que a concorrência pode ser determinante para obter rendimentos nas durações de tempo ou, a atrasos na execução de trabalho até uma situação insustentável.

Agora que todos os dados sobre as durações, utilizando as diferentes quantias de *threads* foram apresentados, foi elaborado o estudo de consumo de memória. Este apresenta a quantidade de recursos de memória em Megabytes, que a máquina utiliza de uma forma aproximada ao executar o algoritmo com as diferentes quantias de *threads*. A Tabela 16 apresenta as informações sobre os consumos, obtidas da realização dos testes análise de memória com o auxílio do JProfiler.

Tabela 16 – Representação dos consumos de memória utilizando threads no *Pidgeonholesort*

1 Thread	2Threads	3Threads	4Threads	5Threads	6Threads	7Threads	8Threads
378.493	1196,78	1271,92	1309,80	1348,31	1384,30	1423,12	1460,06

Ao criar múltiplas *threads* o aumento no consumo de memória é evidente, não só pelo *overhead* da criação destas mas também pela necessidade de executar cópias de uma porção do problema De forma semelhante ao que ocorre no algoritmo *Mergesort*, a necessidade de cópias de dados para cada uma das *threads*, torna a implementação do mecanismo *Thread* um perigo para o sistema sem a análise dos recursos disponíveis e dos consumos possíveis. Uma vez que os

resultados de tempo apresentam que a implementação mais eficiente é a *Single Thread*, e que o seu consumo de memória apresenta os mesmos resultados, esta implementação torna-se de fato muito eficiente em ambos consumos.

5.3.3 *ExecutorService*

Como análise do mecanismo *Thread* completa, foi elaborado o estudo do funcionamento das *threadpools* e dos mecanismos de verificação de estado através do *ExecutorService*. Nas três diferentes máquinas, foram criadas as *threadpools* com as dimensões iguais ao número de processadores lógicos, apesar dos resultados do estudo de *threads* apresentar que esta não é a implementação mais benéfica. Com esta escolha foi esperado obter-se resultados interessantes sobre os mecanismos, de forma a se poder elaborar uma implementação concorrente sem o problema apresentado no consumo de tempo do mecanismo *threads*.

De forma a dar início ao estudo foram elaborados os testes na máquina com o CPU Dual-Core, os resultados obtidos desta encontram-se apresentados na Figura 94.

Parametros		CountdownLach			CompletionService			Boolean		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	1350	1342	1372	1305	1545	1339	843	1107	849
	4	1386	1375	1296	1113	1562	1430	692	751	1264
	8	626	623	1770	666	671	1792	617	608	1944
5 mil	2	625	1189	540	570	879	533	573	571	553
	4	560	598	918	566	602	922	552	572	1003
	8	537	514	1320	542	578	1492	551	596	1447
10 mil	2	549	533	550	579	559	560	572	526	591
	4	547	539	711	535	585	752	579	603	900
	8	548	540	1144	547	590	1353	593	573	1474
50 mil	2	1136	1011	1168	1114	1014	1137	1119	985	1133
	4	1125	1036	1275	1866	984	2677	1301	1001	1370
	8	1169	1159	1646	1128	1138	1775	1318	1149	1988
100 mil	2	1939	1801	2026	1905	1769	2042	1919	1751	1888
	4	1843	1766	2141	1731	1724	1814	2012	1926	2152
	8	1971	1963	2414	1999	1803	2328	2988	2122	2789
Total (Threadpool):		15911	15989	20291	16166	16003	21946	16229	14841	21345
Total (Mecanismo):		52191			54115			52415		

Figura 94 – Resultados dos testes *ExecutorService* com *Pidgeonholesort* máquina Dual-Core

Ao contrário de todos os resultados obtidos com o *ExecutorService* até o momento, em que indicavam que a utilização de Booleanos como variável de estado não seria a melhor implementação, a Figura 94 indica uma situação contrária. Apesar dos resultados aparentarem ser bastante normais, a anomalia dos resultados ao utilizar booleanos com uma *threadpool* de W*orkStealing* torna o estudo um pouco mais complexo. Ao verificar os resultados através do sistema de cor implementado, é visível que as zonas de lucro, na dimensão de mil ao utilizar Booleanos, são muito mais vantajosas que as dos restantes mecanismos. Caso sejam analisados os dados, considerando as durações ao ordenar mil posições como um caso isolado, são obtidos resultados diferentes, esta análise é possível ao verificar as informações na Figura 95

	CountdownLach			CompletionService			Boolean		
	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	3362	3340	4438	3084	3778	4561	2152	2466	4057
Restante	12549	12649	15853	13082	12225	17385	14077	12375	17288

Figura 95 – Totais de durações do *ExecutorService* com filtro de dimensões

Apesar da Figura 95, apresenta resultados positivos nas restantes dimensões ao utilizar Booleanos com uma *threadpool* de W*orkStealing* mas a diferença total das durações é reduzida em mais de mais de 75%, chegando mesmo a ser ultrapassada pela implementação de uma *threadpool* de *WorkStealing* com o *CompletionService*. Utilizando apenas os dados obtidos do estudo das dimensões restantes, foi optado para o caso base o uso da *threadpool* de W*orkStealing* com o mecanismo *CompletionService*. Esta escolha foi baseada no conhecimento, que os resultados dos primeiros testes são afetados nas primeiras execuções.

De forma a verificar os benefícios ao longo de uma hora das múltiplas implementações, foi elaborada a coleção dos resultados e calculada a diferença entre as implementações com o caso mais simples de implementar. No final de todo o tratamento de dados, foi produzida a Figura 96, onde se encontram todos os resultados.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	36,1	35,5	37,7	32,9	50,0	35,3		18,8	0,4
	4	60,2	59,2	52,4	36,5	75,4	64,0		5,1	49,6
	8	0,9	0,6	112,1	4,8	5,3	114,3		-0,9	129,0
5 mil	2	5,4	64,5	-3,5	-0,3	32,0	-4,2		-0,2	-2,1
	4	0,9	5,0	39,8	1,5	5,4	40,2		2,2	49,0
	8	-1,5	-4,0	83,7	-1,0	2,9	102,5		4,9	97,6
10 mil	2	-2,4	-4,1	-2,3	0,7	-1,4	-1,3	**Casos**	-4,8	2,0
	4	-3,3	-4,1	13,7	-4,6	0,6	17,9		2,5	33,3
	8	-4,6	-5,4	55,8	-4,7	-0,3	76,9	**Base**	-2,0	89,1
50 mil	2	0,9	-5,8	2,6	-0,3	-5,6	1,0		-7,2	0,8
	4	-8,1	-12,2	-1,2	26,1	-14,6	63,5		-13,8	3,2
	8	-6,8	-7,2	14,9	-8,6	-8,2	20,8		-7,7	30,5
100 mil	2	0,6	-3,7	3,3	-0,4	-4,7	3,8		-5,3	-1,0
	4	-5,0	-7,3	3,8	-8,4	-8,6	-5,9		-2,6	4,2
	8	-20,4	-20,6	-11,5	-19,9	-23,8	-13,3		-17,4	-4,0

Figura 96 – Comparação das durações dos mecanismos *ExecutorService* com *Pigeonholesort* no Dual-Core

Ao verificar os resultados, é possível afirmar que várias implementações apresentam um grande conjunto de resultados lucrativos. Ignorando os casos em que foram utilizadas *threadpools* do tipo *Cached*, todos os casos apresentam durações bastante baixas e interessantes. Na utilização do *ContdownLach* não existe uma diferença significativa, mas como existem dois momentos que apresentam durações superiores ao caso base, indicando assim que a utilização de utilizar uma *threadpool* do tipo *WorkStealing* pode ser a mais benéfica. O mesmo cenário é apresentado no *CompletionService* mas neste a *Fixed threadpool* apresenta mais uma situação lucrativa que a de *WorkStealing*. Na globalidade os ganhos desta situação não recuperam os prejuízos ao comparar ambas. Uma vez que a *threadpool WorkStealing* aprsenta ser outra vez a melhor implementação foi possível determinar uma parte do caso base. Para determinar que quantidade de *tasks* se deveria utilizar em conjunto com a porção do caso base escolhido, foi utilizada a Figura 96. Uma vez que a utilização de duas e quatro *tasks* na dimensão de mil garantem imensos prejuízos de tempo, foi necessário optar pela utilização de oito *task*. Com o caso base totalmente definido, foram comparados as durações deste com as obtidas do estudo das *threads*. De forma a entender o impacto dos diferentes mecanismos de concorrência em sistemas empresarias, foram determinadas as diferenças de execução ao longo de uma hora. As diferenças encontram-se ilustradas em minutos na Figura 97.

Tamanho	2 Threads	8 Tasks, TP de Workstealling com CompletionService	Comparação
mil	848	671	-12,52
5 mil	394	578	28,02
10 mil	492	590	11,95
50 mil	1237	1138	-4,80
100 mil	1962	1803	-4,86

Figura 97 – Comparação das durações dos casos base *Pidgeonholesort Thread* e *ExecutorService* do Dual-Core

Segundo os valores da Figura 97, a implementação do caso base *ExecutorService* ultrapassa a implementação de duas *threads*, em três das cinco dimensões. Ao verificar a menor dimensão o *ExecutorService* apresenta o melhor resultado, mas em ambos os mecanismos é óbvia a influência de interrupções no sistema, ao apresentar durações mais longas que a ordenação de dimensão superior. Apesar disto o *ExecutorService* apresenta um aumento muito inferior que o da utilização de *threads,* o que indica que o carregamento inicial ao utilizar *threads* é mais pesado. Nas restantes dimensões existiu uma forte presença de lucros ao utilizar *threads*, que nas dimensões superiores foram ultrapassadas pelo *ExecutorService*. Ambos apresentam resultados interessantes em dimensões diferentes, mas existem notas que é necessário compreender. Ao contrário da utilização de *threads*, em que a utilização de duas *threads* foi superior em todos os aspetos as outras múltiplas quantias, o caso base do *ExecutorService* apresenta resultados que podem ser melhorados alterando a quantia de *task*, ou os mecanismo de concorrência implementados.

De seguida foram testadas as implementações do *ExecutorService* no processador i5. No fim foi possível o conjunto de tempos ilustrados na Figura 98.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	506	704	494	498	745	493	722	673	759
	4	661	694	536	710	600	619	1054	893	573
	8	671	780	593	553	719	719	695	764	621
	16	630	577	679	487	569	653	520	676	804
	32	544	554	834	501	582	869	552	525	851
5 mil	2	171	386	252	313	356	303	288	372	335
	4	334	460	318	437	472	350	468	485	350
	8	356	554	381	500	453	392	467	674	336
	16	363	553	603	503	525	605	582	561	517
	32	621	1142	793	471	729	874	1427	1761	1128
10 mil	2	438	421	284	465	396	265	331	509	346
	4	746	525	387	517	579	353	793	628	387
	8	519	563	424	529	616	357	486	819	380
	16	576	573	547	524	596	520	1056	806	648
	32	535	596	773	603	644	762	660	940	866
50 mil	2	672	671	626	685	632	616	825	788	656
	4	899	875	722	905	844	704	1089	1015	739
	8	890	885	798	923	1039	771	990	1054	918
	16	913	882	946	944	869	917	1223	1177	1158
	32	1185	893	1166	1058	892	1213	1470	1310	1602
100 mil	2	1232	1020	975	1106	993	997	1093	1026	1006
	4	1340	1142	1085	1320	1112	1183	1366	1185	1092
	8	1346	1195	1181	1339	1338	1180	1532	1272	1346
	16	1362	1426	1413	1586	1226	1307	1623	1356	1542
	32	1448	1351	1631	1558	1284	1573	1763	1638	2101
Total (Threadpool):		18958	19422	18441	19035	18810	18595	23075	22907	21061
Total (Mecanismo):		56821			56440			67043		

Figura 98 – Resultados dos testes *ExecutorService* com *Pidgeonholesort* máquina i5

Ao contrário dos resultados obtidos no Dual-Core, o i5 apresenta subidas inesperadas na duração, muito menos significativos nas menores dimensões. Esta estabilidade no final faz com que a utilização de Booleanos perca por completo a sua força, garantindo a firmeza dos restantes mecanismos de verificação de estado. A Cached *threadpool* com o *CountdownLach* apresenta o melhor total resultados. Ao analisar as margens de erro e ao comparar, é possível afirmar que este caso foi beneficiado, em determinados momentos. Como auxílio na determinação de qual o melhor caso base, foi elaborado a comparação de durações de execução ao longo de uma hora. Para esta comparação, foi calculado o número de iterações que o caso mais simples é capaz de executar numa hora e verificar em minutos os prejuízos ou lucros para os restantes atingir esse número de execuções. No final das comparações, com os resultados obtidos foi produzida a Figura 99.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	WS	Cached	Fixed	WS	Cached	Fixed	WS	Cached
mil	2	-18,0	-1,5	-18,9	-18,6	1,9	-19,0	Casos Base	-4,1	3,1
	4	-22,4	-20,5	-29,5	-19,6	-25,8	-24,8		-9,2	-27,4
	8	-2,1	7,3	-8,8	-12,3	2,1	2,1		6,0	-6,4
	16	12,7	6,6	18,3	-3,8	5,7	15,3		18,0	32,8
	32	-0,9	0,2	30,7	-5,5	3,3	34,5		-2,9	32,5
5 mil	2	-24,4	20,4	-7,5	5,2	14,2	3,1		17,5	9,8
	4	-17,2	-1,0	-19,2	-4,0	0,5	-15,1		2,2	-15,1
	8	-14,3	11,2	-11,0	4,2	-1,8	-9,6		26,6	-16,8
	16	-22,6	-3,0	2,2	-8,1	-5,9	2,4		-2,2	-6,7
	32	-33,9	-12,0	-26,7	-40,2	-29,3	-23,3		14,0	-12,6
10 mil	2	19,4	16,3	-8,5	24,3	11,8	-12,0		32,3	2,7
	4	-3,6	-20,3	-30,7	-20,9	-16,2	-33,3		-12,5	-30,7
	8	4,1	9,5	-7,7	5,3	16,0	-15,9		41,1	-13,1
	16	-27,3	-27,4	-28,9	-30,2	-26,1	-30,5		-14,2	-23,2
	32	-11,4	-5,8	10,3	-5,2	-1,5	9,3		25,5	18,7
50 mil	2	-11,1	-11,2	-14,5	-10,2	-14,0	-15,2		-2,7	-12,3
	4	-10,5	-11,8	-20,2	-10,1	-13,5	-21,2		-4,1	-19,3
	8	-6,1	-6,4	-11,6	-4,1	3,0	-13,3		3,9	-4,4
	16	-15,2	-16,7	-13,6	-13,7	-17,4	-15,0		-2,3	-3,2
	32	-11,6	-23,6	-12,4	-16,8	-23,6	-10,5		-6,5	5,4
100 mil	2	7,6	-4,0	-6,5	0,7	-5,5	-5,3		-3,7	-4,8
	4	-1,1	-9,8	-12,3	-2,0	-11,2	-8,0		-8,0	-12,0
	8	-7,3	-13,2	-13,7	-7,6	-7,6	-13,8		-10,2	-7,3
	16	-9,6	-7,3	-7,8	-1,4	-14,7	-11,7		-9,9	-3,0
	32	-10,7	-14,0	-4,5	-7,0	-16,3	-6,5		-4,3	11,5

Figura 99 – Comparação das durações dos mecanismos *ExecutorService* com *Pigeonholesort* no i5

A *Cached threadpool* com o *CountdownLach* apresenta quatro situações de prejuízo, quando comparadas com o caso base, o mesmo número de prejuízos é presenciado na utilização do mesmo mecanismo de controlo de estados com a *threadpool* do tipo *Fixed*. Outra situação interessante de realçar, é ao utilizar a *Fixed threadpool* mas com o mecanismo *CompletionService*, esta combinação apresenta cinco situações de prejuízo. Estes três casos apresentam o maior conjunto de situações lucrativas, com a comparação dos prejuízos, foi é possível determinar qual o mecanismo que produz os melhores rendimentos, e os mais estáveis. No caso *Cached threadpool* com *CountdownLach*, ao somar os seus quatro pontos negativos é obtido um total de 59,3 minutos de prejuízo, é de frisar que este caso apresenta o melhor conjunto ao somar todas as durações de execução. Ao somar os prejuízos do *CountdownLach* com uma *Fixed threadpool* são obtidos 43,8 minutos, este caso apresenta o quarto melhor conjunto de resultados mas um conjunto de prejuízos inferior ao melhor. O terceiro caso escolhido foi o *CompletionService* com uma *Fixed threadpool*, este apresenta um total de 39,7 minutos de prejuízos, representando assim o quinto melhor conjunto de tempos. Os restantes casos foram ignorados uma vez que apresentam grandes quantidades de situações de prejuízo,

ou um resultado de soma de prejuízos muito elevado. O *Cached threadpool* com *CountdownLach* foi descartado, uma vez que apresenta tantas situações de *prejuízo* como o *CountdownLach* com *Fixed threadpool* mas um total de prejuízos maiores. Entre os dois últimos casos, foi optado pelo caso *CompletionService* com *Fixed threadpool*, uma vez que apresenta a menor média e mediana dos dois. De forma a escolher o número de *tasks* a utilizar no caso base, foram escolhidos os casos que não concebessem situações de prejuízo, e a maior quantia lucros. No final de todo o cálculo, foi optado por definir o caso base com quatro *tasks* pois apresentam uma soma de 3889 μs, inferior aos restantes casos, onde se obteve um total de 4044 μs com 16 *tasks* e com 36 *tasks* 4191 μs.

Para finalizar o estudo do *ExecutorService* e dos seus mecanismos no i5, foi elaborada a comparação entre o seu caso base e o caso escolhido do estudo do mecanismo *Thread* na mesma máquina. Os resultados desta comparação encontram-se apresentados na Figura 100.

Tamanho	4 Threads	4 Tasks, Fixed Threadpool com CompletionService	Comparação
mil	499	710	25,37
5 mil	301	437	27,11
10 mil	408	517	16,03
50 mil	848	905	4,03
100 mil	1186	1320	6,78

Figura 100 – Comparação das durações dos casos base *Pidgeonholesort Thread* e *ExecutorService* do i5

Ao observar os resultados da Figura 100, a afirmação mais óbvia é que o *ExecutorService* apenas apresentou resultados negativos. Apesar do mecanismo aparecer em desvantagem, é de realçar que o caso escolhido é apenas um de muitos, e que apresenta um conjunto de resultados generalizados. Ao analisar com grande cuidado as implementações e elaborar optar por um mecanismo diferente para cada situação é possível que o *ExecutorService* ofereça melhores resultados. Outro sinal interessante é a recuperação dos prejuízos com o aumentar da dimensão, isto indica que em dimensões superiores o mecanismo pode conseguir apresentar resultados significativamente superiores.

Para apresentar os resultados obtidos do processador i7, foi elaborada a Figura 101. Esta apresenta dados obtidos de testes aos mecanismos em questão com uma divisão de trabalho até 32 *tasks*.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	296	457	298	302	447	284	338	464	334
	4	490	659	463	461	681	406	484	658	450
	8	742	830	576	853	826	490	855	924	492
	16	701	729	583	711	757	591	777	808	571
	32	712	760	847	863	857	827	807	871	775
5 mil	2	196	304	181	242	318	233	302	309	268
	4	345	476	292	381	450	279	444	513	427
	8	674	810	431	642	809	335	864	835	486
	16	695	749	601	655	883	530	910	834	916
	32	1168	796	1066	966	837	816	1049	886	1352
10 mil	2	325	282	240	278	294	241	334	272	360
	4	422	491	344	445	446	312	519	429	525
	8	847	729	494	842	791	390	829	824	583
	16	873	724	646	741	722	466	749	813	786
	32	807	791	843	673	757	645	925	934	928
50 mil	2	461	518	467	473	505	480	486	558	553
	4	642	678	605	700	636	525	617	684	617
	8	1051	859	788	857	814	603	891	959	844
	16	1103	879	1212	907	823	687	1029	1082	984
	32	1271	915	1269	975	951	885	1261	1082	1375
100 mil	2	922	824	799	785	809	814	801	789	895
	4	1125	1015	894	869	902	812	854	1039	836
	8	1261	1183	1112	1120	1079	875	1168	1136	972
	16	1312	1208	1345	1183	1160	1040	1358	1191	1283
	32	1363	1285	1654	1252	1146	1282	1532	1417	1658
Total (Threadpool):		19804	18951	18050	18176	18700	14848	20183	20311	19270
Total (Mecanismo):		56805			51724			59764		

Figura 101 – Resultados dos testes *ExecutorService* com *Pidgeonholesort* máquina i7

Ao analisar as durações obtidas, estas indicam que a utilização da *threadpool* do tipo *Cached* com o *CompletionService* é a melhor implementação. Uma vez que o i7 é o processador mais poderoso, este tira grande vantagem da utilização uma *threadpool* dinâmica, onde apenas são requisitadas *threads* quando existe uma grande trabalho de processo. O *CompletionService* dos três mecanismos de verificação de estado é o que fornece o melhor conjunto de resultados. Ao utilizar este mecanismo com uma *threadpool*, que apenas utiliza as *threads* consoante a necessidade para executar ordenações velozes como as do *Pigeonholesort,* este vai rentabilizar. Pois ao executar as *tasks* numa *threadpool* de menor dimensão resultando em menor *context-switching* pois permite até quatro *threads* em simultâneo. Ao utilizar um menor número de recursos de paralelismo, o CPU poderá rentabilizar mais ainda devido à existência da tecnologia *Turbo Boost*. Esta permite em casos de utilização de um número menor *threads*, aumentar as velocidades de execução das *threads* ativas.

A Figura 102 apresenta os resultados da comparação da execução ao longo de uma hora, de cada situação com o caso base, indicando as diferenças em minutos.

Parametros		CountdownLach			CompletionService			Booleano		
Tamanho	Tasks	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached	Fixed	Workstealing	Cached
mil	2	-7,5	21,1	-7,1	-6,4	19,3	-9,6	Casos Base	22,4	-0,7
	4	0,7	21,7	-2,6	-2,9	24,4	-9,7		21,6	-4,2
	8	-7,9	-1,8	-19,6	-0,1	-2,0	-25,6		4,8	-25,5
	16	-5,9	-3,7	-15,0	-5,1	-1,5	-14,4		2,4	-15,9
	32	-7,1	-3,5	3,0	4,2	3,7	1,5		4,8	-2,4
5 mil	2	-21,1	0,4	-24,0	-11,9	3,2	-13,7		1,4	-6,8
	4	-13,4	4,3	-20,5	-8,5	0,8	-22,3		9,3	-2,3
	8	-13,2	-3,8	-30,1	-15,4	-3,8	-36,7		-2,0	-26,3
	16	-14,2	-10,6	-20,4	-16,8	-1,8	-25,1		-5,0	0,4
	32	6,8	-14,5	1,0	-4,7	-12,1	-13,3		-9,3	17,3
10 mil	2	-1,6	-9,3	-16,9	-10,1	-7,2	-16,7		-11,1	4,7
	4	-11,2	-3,2	-20,2	-8,6	-8,4	-23,9		-10,4	0,7
	8	1,3	-7,2	-24,2	0,9	-2,8	-31,8		-0,4	-17,8
	16	9,9	-2,0	-8,3	-0,6	-2,2	-22,7		5,1	3,0
	32	-7,7	-8,7	-5,3	-16,3	-10,9	-18,2		0,6	0,2
50 mil	2	-3,1	4,0	-2,3	-1,6	2,3	-0,7		8,9	8,3
	4	2,4	5,9	-1,2	8,1	1,8	-8,9		6,5	0,0
	8	10,8	-2,2	-6,9	-2,3	-5,2	-19,4		4,6	-3,2
	16	4,3	-8,7	10,7	-7,1	-12,0	-19,9		3,1	-2,6
	32	0,5	-16,5	0,4	-13,6	-14,8	-17,9		-8,5	5,4
100 mil	2	9,1	1,7	-0,1	-1,2	0,6	1,0		-0,9	7,0
	4	19,0	11,3	2,8	1,1	3,4	-3,0		13,0	-1,3
	8	4,8	0,8	-2,9	-2,5	-4,6	-15,1		-1,6	-10,1
	16	-2,0	-6,6	-0,6	-7,7	-8,7	-14,1		-7,4	-3,3
	32	-6,6	-9,7	4,8	-11,0	-15,1	-9,8		-4,5	4,9

Figura 102 – Comparação das durações dos mecanismos *ExecutorService* com *Pigeonholesort* no i7

Os lucros ao utilizar a *threadpool* do tipo *Cached* com o *CompletionService* são esmagadores em comparação com os restantes casos. Com estes conjuntos de lucros, foi estabelecido este caso como base de estudo mas uma vez que existem tantos casos de rendimento, foi necessário analisar com mais cuidado as durações com cada quantia de *tasks*. Para elaborar um caso completo foram calculados os totais separando os resultados pelas quantias de *tasks*, de forma a determinar o menor consumo de tempo. Os resultados dos cálculos encontram-se apresentados na Figura 103.

2	4	8	16	32
2052	2334	2693	3314	4455

Figura 103 – Totais de durações com filtro por número de tasks do caso base do *ExecutorService no i7*

Uma vez que as ordenações com a *Cached threadpool* e duas *tasks* apresentando os melhores resultados, e utilizam menos recursos, foi defenido este caso como base para as futuras comparações.

Com o caso base para o *ExecutorService* devidamente definido, foi realizada a comparação das durações de execução entre este e o caso base do mecanismo *Thread*. Ao elaborar esta comparação foi produzida a Figura 104, onde os possíveis minutos rentabilizados ou perdidos ao longo de uma hora são ilustrados.

Tamanho	8 Threads	2 Tasks numa Cached TP com CompletionService	Comparação
mil	643	284	-33,50
5 mil	646	233	-38,36
10 mil	657	241	-37,99
50 mil	877	480	-27,16
100 mil	1148	814	-17,46

Figura 104 – Comparação das durações dos casos base *Pidgeonholesort Thread* e *ExecutorService* do i7

A Figura 104 apresenta uma abismal diferença nas durações entre os mecanismos, demonstrando benefícios entre os 15 e os 40 minutos nas cinco dimensões. Apesar da *Cached threadpool* com *CompletionService* apresentar grandes benefícios, há que evidenciar que os lucros diminuem com o aumento da dimensão quase constantemente. Este decréscimo pode levar à possibilidade de ser uma implementação inferior em dimensões superiores. Mesmo assim o *ExecutorService* utilizando outro conjunto de mecanismos pode eventualmente continuar a apresentar rendimentos.

Para terminar o estudo dos mecanismos com *ExecutorService* foi elaborado uma análise sobre o consumo de memória no Dual-Core, utilizando duas *tasks* em cada implementação. Os resultados obtidos estão apresentados na Tabela 17, ilustrando os valores de consumo em Megabytes.

Tabela 17 – Representação dos consumos de memória utilizando threadpools no *Pidgeonholesort*

CountdownLach			**CompletionService**			**Booleano**		
Fixed	**WS**	**Cache**	**Fixed**	**WS**	**Cache**	**Fixed**	**WS**	**Cache**
936,65	936,33	823,63	899,08	936,33	936,34	823,32	787,00	786,69

Os consumos obtidos são valores aproximados de 936 Megabytes, com algumas exceções que são justificáveis uma vez que a ordenação é efetuada rapidamente e poucos dados são possíveis de obter, mesmo com um *array* de cem milhões de elementos a ordenar. Uma vez que o Jprofiler apresenta os consumos a cada segundo, determinadas implementações podem não apresentar o consumo máximo. Estes consumos quando comparados com os da utilização de *threads*, apresentam valores inferiores à utilização de múltiplas *threads* mas um aumento, quando comparados com o consumo de ordenação obtido do estudo *Single Thread*.

5.3.4 ***Fork/Join***

O último grupo de testes com o algoritmo de ordenação *Pidgeonholesort* foi elaborado utilizando *ForkJoinPools,* e criando *tasks* de acordo com o *threshold* definido em cada CPU. Neste estudo o *threshold* é definido unicamente pelo valor da cache do CPU, sem a existência da atribuição de 45% da dimensão do *array* ao valor de *threshold*, no caso deste ser bastante elevado.

Para iniciar o estudo do mecanismo, foi executado um estudo semelhante a versão *Single Thread*, analisando a amostragem, e calculando os valores necessários na máquina Dual-Core. Os resultados desta análise estão evidenciados na Figura 105.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	574	1619	1499,23	119,77	3118,23
5000	915	974	77,93	896,07	1051,93
10000	1005	1233	170,8	1062,2	1403,8
50000	2555	2884	172,06	2711,94	3056,06
100000	6913	7893	406,63	7486,37	8299,63

Figura 105 – Resultados do teste *Fork/Join Pidgeonholesort* no Dual-Core

Ao analisar os valores, é observável uma margem de erro gigantesca na dimensão de mil posições, este valor foi comprovado várias vezes. Este acontecimento é justificável pelo conjunto de pontos, o primeiro sendo a pequena duração de execução, e o segundo sendo o atraso inicial dos primeiros testes. A junção destes dois, com o funcionamento do mecanismo pode ser a causa da margem de erro obtida. De forma a concluir se estes resultados são capazes de ser lucrativos, foi elaborada a comparação dos casos base elaborados com o algoritmo de ordenação *Pidgeonholesort*. Os valores de duração de cada mecanismo encontram-se devidamente identificados na Figura 106.

Tamanho	Single Thread	2 Threads	Executor Service	ForkJoin
mil	31	848	671	574
5 mil	181	394	578	915
10 mil	298	492	590	1005
50 mil	413	1237	1138	2555
100 mil	823	1962	1803	6913

Figura 106 – Comparação das medianas dos casos base *Pidgeonholesort* no Dual-Core

Ao verificar a comparação elaborada, é possível afirmar que as implementações de concorrência elaboradas para o algoritmo *Pidgeonholesort* simplesmente fornecem rentabilizações. Apesar de deste fato é de evidenciar que a implementação de uma ForkJoinPool, a pesar de ter um bom início na dimensão mais pequena, o aumento desta simplesmente torna a implementação desinteressante. Estes pontos provam que alguns algoritmos simplesmente não rentabilizam da concorrência, sem a definição mais cuidada possível dos *thresholds*, e com as validações necessárias.

De forma idêntica ao Dual-Core, foram obtidos os resultados expostos na Figura 107, ilustrando as durações ao executar uma *ForkJoinPool* com ordenação *Pidgeonholesort* no CPU i5, onde a Cache existente é igual a 3 MB.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	418	483	102,23	380,77	585,23
5000	409	463	37,2	425,8	500,2
10000	499	662	167,26	494,74	829,26
50000	1212	1272	84,77	1187,23	1356,77
100000	1697	2106	197,11	1908,89	2303,11

Figura 107 – Resultados do teste *Fork/Join Pidgeonholesort* no i5

Mais uma vez existe um valor elevado na margem de erro na primeira dimensão, mas com um menor impacto que no Dual-Core, provocando com que o valor da mediana na primeira dimensão, seja superior ao da dimensão superior. Apesar deste valor poder apresentar uma anomalia, foi elabora a comparação de durações, entre os resultados obtidos das implementações com mecanismos de concorrência no i5. As durações obtidas encontram-se apresentadas na Figura 108.

Tamanho	Single Thread	4 Threads	ExecutorService	ForkJoin
mil	91	499	710	418
5 mil	213	301	437	409
10 mil	78	408	517	499
50 mil	308	848	905	1212
100 mil	650	1186	1320	1697

Figura 108 – Comparação das medianas dos casos base *Pidgeonholesort* no i5

A *Single Thread* é a implementação que fornece melhores resultados. Ao contrário dos resultados obtidos do Dual-Core, os resultados nas três primeiras dimensões ao utilizar a estrutura *Fork/Join* provam ser bastante eficientes, ficando par-a-par com restantes implementações concorrentes. Este ponto prova que o *thresdhold* definido no i5 é mais adequado do que o definido no Dual-Core, indicando que uma possível causa dos resultados do Dual-Core serem tão inadequados é o valor do *thresdhold* ser demasiado baixo.

A Figura 109, apresenta as durações obtidas ao efetuar os testes de execução da *threadpool ForkJoinPool* para a execução ordenações do tipo *Pidgeonholesort* no processador i7.

Tamanho:	Mediama	Media:	Margem	ValorMin:	ValorMax
1000	281	316	74,65	241,35	390,65
5000	288	322	39,22	282,78	361,22
10000	193	212	13,92	198,08	225,92
50000	655	758	89,07	668,93	847,07
100000	1308	1464	84,51	1379,49	1548,51

Figura 109 – Resultados do teste *Fork/Join Pidgeonholesort* no i7

As durações no CPU i7 apresentam margens de erro muito baixas, o que permite uma abordagem de comparação com os restantes mecanismos muito mais simples. Apesar das margens de erro demonstrarem valores aceitáveis, existe uma diminuição nos tempos de ordenação nas dimensões intermedias. Este fato leva a considerar que os resultados obtidos nas menores dimensões são mais elevados pois o *threshold* é demasiado alto, gerando um número de divisões em excesso, atrasando a ordenação do *array*. Com a coleção de resultados completa, foi elaborada a produção da Figura 110, onde estão ilustrados todos os casos base produzidos.

Tamanho	Single Thread	8 Threads	ExecutorService	ForkJoin
mil	49	643	284	281
5 mil	98	646	233	288
10 mil	47	657	241	193
50 mil	228	877	480	655
100 mil	460	1148	814	1308

Figura 110 – Comparação das medianas dos casos base *Pidgeonholesort* no i7

No i7, a utilização da estrutura Fork/Join demonstrou um aumento no rendimento devido à amplificação do atributo de cache, no cálculo de *threshold*. Mesmo com este aumento, esta implementação continuou incapaz de competir com as durações de execução da versão *Single Thread*. Sendo um algoritmo de ordenação que, realiza a sua função em durações muito pequenas, este apresenta melhores rendimentos nas menores divisões de tarefas, sendo estas as versões sem concorrência. Uma vez que o *ExecutorService* utiliza o caso escolhido de duas *tasks* com uma *threadpool* do tipo *Cached*, a resolução do problema utilizará uma *threadpool* com duas *threads*, sendo este o valor mínimo de paralelismo. A implementação de 8 *threads* implementa os piores resultados, uma vez que elabora oito divisões sobre o problema a resolver. No final a estrutura *Fork/Join* não demonstra os melhores resultados, pois em maiores dimensões sem o aumento da variável cache implica que existirá um aumento da criação de tarefas.

Para avaliar os diferentes consumos de memória foram comparados e avaliados os consumos de cada caso base no Dual-Core. Os resultados de cada análise encontram-se apresentados em Megabytes na Tabela 18.

Tabela 18 – Consumos de memória obtidos das implementações com *Pidgeonholesort*

Single Thread	**2 Threads**	**ExecutorService**	**Fork/Join**
378,493	1196,78	936,34	3389,76

Ao verificar os diferentes consumos, é possível afirmar que nenhuma versão concorrente consegue-se aproximar devidamente do consumo da *Single Thread*, devido à necessidade de criação de um novo *array* com a porção da solução para cada divisão. A versão *ForkJoinPool* consegue atingir um bastante elevado, devido ao grande número de divisões realizadas até a porção do problema ser igual ou menor que o *threshold*. Como comprovado ao longo do estudo

no Dual-Core o *threshold* é muito baixo, necessitando assim de múltiplas tarefas e de criação de um array para cada. No *ExecutorService* com uma *threadpool* de *WorkStealing* com *CompletionService* e apenas duas *tasks*, são apresentados consumos inferiores à implementação de duas *threads*. Este decréscimo de consumo ocorre devido à redução do *overhead* gerado, e a incapacidade de obter o resultado de consumo totalmente preciso. Uma vez que o algoritmo realiza um aumento no consumo como no estudo do *Mergesort*, a melhor abordagem é a execução de *profiling* da máquina, para verificar o melhor ponto de consumo tempo e memória.

5.4 Resultados da ordenação *JavaSort*

Como fase final de estudo, foram analisados os mecanismos de ordenação já existentes na API do Java, que correspondem com os objetivos dos restantes mecanismos. Estes forneceram informações sobre os diferentes mecanismos e permitiram uma perspetiva sobre as durações, quando os algoritmos são trabalhados com grande profundidade. Os métodos da API foram considerados pela sua facilidade de acesso e pela utilidade na comparação de implementações mais aprofundadas. Os métodos escolhidos foram executados sobre as máquinas adquiridas e utilizadas nos testes previamente estudados.

Para começar foram realizados os testes sobre os métodos *ArraySort()*, *ArraysparallelSort()*, *IntStreamSort()* e *ParallelIntStreamSort()* no Dual-Core. Os resultados obtidos destes testes encontram-se ilustrados na Figura 111.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	112	111	277	255
5 mil	757	666	578	1118
10 mil	742	1597	849	866
50 mil	4389	4405	4816	4614
100 mil	9437	9478	10381	9808

Figura 111 – Resultados dos testes JavaSort no Dual-Core

Uma vez que o Dual-Core permite que duas *threads* executem em simultâneo, o esperado dos resultados seria que o *ArraysParallelSort* apresenta-se valores mais benéficos que o ArraySort mas tal não acontece. Uma vez que durante o funcionamento do sistema podem ocorrer imensos *interrupts* por outros processos, é normal que num sistema que permita uma quantia tão baixa de *threads* não demonstre diferenças significantes. Apesar da ocorrência de *interrupts* ambas versões paralelas apresentam rendimentos sobre as implementações não concorrentes.

A Figura 112 apresenta a comparação das execuções ao longo de uma hora, entre os casos bases e as suas versões concorrentes.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	Casos Base	-0,5	Casos Base	-4,8
5 mil		-7,2		56,1
10 mil		69,1		1,2
50 mil		0,2		-2,5
100 mil		0,3		-3,3

Figura 112 – Comparações dos casos base com as suas versões paralelas da API do Java no Dual-Core

Ao comparar os casos elaborados, é possível verificar que existem situações de rentabilidade ao utilizar a versão paralela, mas também existem situações de prejuízos muito baixos, e extremamente elevados. Esta diferença é notável na máquina atual devido às suas limitações de *hardware*, uma vez que o número de *threads* em execução simultânea é muito baixa e existem um grande número de processos a competir por estas *threads*. A realçar, uma vez que não se encontram visíveis, são os valores de margem obtidos na execução das duas versões *IntStream*, estas apresentam valores mais elevados e menos aproximados que nos casos *ArraySort*. No final ambos os casos apresentam resultados interessantes nas versões concorrentes mas como os restantes algoritmos elaborados, apresentam falhas na execução em máquinas com recursos mais escassos.

Ao executar os mesmos testes sobre os mecanismos de ordenação presentes na API na máquina i5, foram retirados os valores de medianas representando as durações na Figura 113.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	125	85	271	298
5 mil	737	506	979	744
10 mil	547	780	608	967
50 mil	3250	2093	3476	2278
100 mil	6950	3859	7221	4027

Figura 113 – Resultados dos testes JavaSort no i5

Ao comparar com os resultados obtidos no Dual-Core, a diferença entre as versões normais e concorrentes encontra-se mais acentuada. Com a exceção da ordenação de dez mil posições, as versões paralelas apresentam sempre os melhores tempos de duração, está é justificável uma vez que nas versões paralelas os valores de margem de erro obtidos são muito superiores que os das versões base. As margens de erro na dimensão de dez mil apresentam uma subida de 446,6 do *ArraySort* para a sua versão paralela, e uma subida de 175,1 do *IntStreamSort* para o *ParallelIntStream*. A robustez da máquina ao possibilitar o funcionamento de quatro *threads* em simultâneo possibilita estas rentabilizações mais acentuadas. Para melhor entender a profundidade dos possíveis ganhos foi elaborada a comparação de execução de uma hora entre os casos base e as suas versões paralelas, os dados obtidos no final dos cálculos encontram-se apresentados na Figura 114.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	Casos Base	-19,2	Casos Base	6,0
5 mil		-18,8		-14,4
10 mil		25,6		35,4
50 mil		-21,4		-20,7
100 mil		-26,7		-26,5

Figura 114 – Comparações dos casos base com as suas versões paralelas da API do Java no i5

No i5 os lucros encontram-se muito mais acentuados em comparação com os resultados do Dual-Core, com lucros bastante constantes independentemente do aumento da dimensão. A API do Java encontra-se fortemente trabalhada, e com verificações para um enorme número de possibilidades, isto permite apesar de causar do *overhead* gerado em cada verificação que no final seja possível obter os resultados satisfatórios. Os valores de margem de erro merecem ser referidos uma vez que apresentam valores muito mais baixos em grande número de situações, chegando apresentar margens inferiores a 1% da média em questão.

Da execução dos mesmos testes mas agora no i7, foi possível obter as seguintes durações ilustradas na Figura 115.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	72	70	228	237
5 mil	528	416	473	644
10 mil	511	452	589	529
50 mil	3061	1027	3239	1085
100 mil	6594	1949	7027	1880

Figura 115 – Resultados dos testes JavaSort no i7

Os valores da Figura 115 apresentam as durações mais baixas das três máquinas, devido à velocidade e quantidade de *threads* possíveis de executar sincronamente no i7. Com a análise é verificado que, os resultados melhoram abundantemente em todas as dimensões entre os casos do *ArraySort* e a sua versão paralela. Nos casos *IntStream* o mesmo não ocorre, nas dimensões de mil e cinco mil, a razão do porque desta ocorrência não foi possível justificar-se apenas com os valores de margem de erro. Na primeira dimensão, como verificado ao longo da tese, as margens de erro causam impacto nas durações, isto é verificado nas médias, pois no *IntStream* foi obtida uma de média *de* 1122, e na versão paralela foi obtida uma média de 1471.Este impacto na média reflete a principal razão de na dimensão de mil a versão paralela os resultados não serem lucrativos. Na dimensão de 5 mil o mesmo não é verificado mas ao observar a média obtida no *IntStream* de 545, e a mediana de 473 foi observada uma diferença substancial entre os valores. Apesar de ser um caso comum a mediana apresentar um valor mais reduzido que a média, neste caso a diferença apresenta a possível causa da versão paralela não ser rentável nesta dimensão. A versão paralela apresenta uma média de 668 e uma mediana de 644, uma diferença muito reduzida quando comparada com a versão *IntStream*. Ao verificar estes valores, foi determinada que os resultados da versão paralela foram mais prejudiciais que a versão base, porque o *ParallelIntStream* apresentou valore mais constantes, fornecendo uma mediana mais

próxima de um caso comum do que a versão base. No final é possível concluir que os mesmos mecanismos mas em máquinas com elevadas capacidades de paralelismo obtenham um rendimento final gigantesco, devido ao bom desenvolvimento do código.

No final foi elaborada a comparação de resultados de execução ao longo de uma hora, de forma a apresentar os minutos ganhos ou perdidos, entre os casos bases e as suas versões paralelas. Esta análise é possível através dos valores ilustrados na Figura 116.

Tamanho	ArraySort	ArraysParallel	IntStreamSort	ParallelIntStream
mil	Casos Base	-1,7	Casos Base	2,4
5 mil		-12,7		21,7
10 mil		-6,9		-6,1
50 mil		-39,9		-39,9
100 mil		-42,3		-43,9

Figura 116 – Comparações dos casos base com as suas versões paralelas da API do Java no i7

Os lucros são evidentes nas implementações paralelas, fazendo com que o uso das versões paralelas seja uma vantagem nos processadores com grandes capacidades de sincronismo de *threads*, como o i7. Mesmo o uso do *ParallelIntStream* com um total de 24,1 minutos de prejuízo, no final esse prejuízo seria completamente ignorado pelos lucros nas dimensões superiores.

Para dar por completo a totalidade do estudo foi elaborada análise de consumo de memória dos mecanismos. A Tabela 19 apresenta o valor aproximado de consumo máximo em Megabytes, ao realizar uma execução dos mecanismos.

Tabela 19 – Consumos de memória obtidos das implementações com *JavaSort*

ArraySort	ArraysparallelSort	IntStreamSort	ParallelIntStreamSort
376,91	377,85	376,91	751,04

Ao analisar os consumos das ordenações *ArraySort* é verificado que o consumo extra é o *overhead* gerado pela criação da nova *thread*. Este *overhead* é demostrando uma diferença de 0,94 megabytes, muito idêntico aos resultados obtidos do estudo de consumo de memória no algoritmo *Quicksort*. A mesma diferença de consumos já não é verificada com o uso de *IntStream*, nesta é verificada um aumento para aproximadamente o dobro. Com este especto de consumo o *IntStream* apesar de demonstrar uma boa coleção de durações, pode ser considerado uma implementação indesejável. Uma vez que o *ParallelIntStreamSort* representa o uso de duas *threads*, no Dual-Core, e o consumo duplicou com a introdução da nova *thread*, o consumo de memória demonstra depender do número de processadores lógicos presentes na máquina. Com esta visão dos consumos, é necessário que os *software developers* realizem o devido *profillling* da máquina e do algoritmo, de forma a obter um bom balanço entre o consumo de tempo e memória.

6 Conclusões

AS análises elaboradas ao longo desta dissertação tiveram como objetivo construir uma base de informação sobre que mecanismos de concorrência, de forma aos programadores que utilizem a API do Java conseguiam obter uma maior eficiência dos seus algoritmos. Cada algoritmo de ordenação permitiu analisar o mesmo conjunto de mecanismos de concorrência, e como estes afetaram as suas implementações mais comuns. Ao avaliar os consumos de tempo e de memória foi possível verificar o impacto que as ferramentas, fornecida pela biblioteca *java.util.concurrent*, realizaram nos algoritmos de ordenação. Através das comparações dentro de cada mecanismo foi possível verificar como os resultados de cada algoritmo ordenação foram modificando, e o porque de tal acontecer.

Uma vez que os mecanismos de concorrência foram avaliados de acordo com o impacto com o algoritmo, a avaliação do mecanismo não foi realizada de uma forma genérica mas sim como este teve impacto na situação. Mesmo ao verificar todos os resultados obtidos lado a lado, foram elaboradas conclusões sobre cada mecanismo, tendo em conta o que ocorreu em todos os algoritmos de ordenação, e como o *hardware* afetou o mecanismo.

6.1 *Threads*

As conclusões mais óbvias de retirar das análises do mecanismo, são que este é capaz de ser rentável, se a duração base de execução do algoritmo permitir que o *overhead* de tempo com a criação de paralelismo seja compensado, e que o impacto no consumo de memória depende da estrutura do algoritmo. O *overhead* tem impacto na duração de execução, este para ser negado necessita de conseguir, através da ordenação paralela, recuperar uma quantia de tempo superior à duração de criação das *threads*. Ao verificar todos os resultados obtidos foi analisado que, a quantia exata de tempo que é necessário para criar o objeto e adquirir a *thread* do sistema não é fácil de determinar. O *overhead* varia consoante a necessidade do algoritmo, e da velocidade do processador. A quantia tempo que um algoritmo necessita de execução, numa única *thread,* para ser rentável com paralelismo, é impossível ser identificada com 100% certeza. Para determinar com algum grau de certeza os passos mais indicados seriam, compreensão dos

recursos de *hardware* disponíveis, estudo da necessidade de partilha de dados e por fim criação se necessário de sistema de comunicação de forma a saber o seu estado, ou para que elas possam comunicar entre si. Depois de executados estes passos, é necessário analisar como o algoritmo se comporta na máquina, utilizando uma única *thread*. Com a temporização de execução base é possível determinar um ponto de partida. Nas análises realizadas no Dual-Core, nenhuma ordenação base com uma duração inferior a 1100 microssegundos conseguiu demonstrar rentabilizações com paralelismo, esta análise com o valor do caso base permite identificar a extensão dos lucros ou prejuízos. No caso do algoritmo em média resolver problemas que demorem menos dessa duração em sistemas como o Dual-Core é facilmente identificado o erro ao implementar paralelismo para resolver o problema. A implementação de mecanismos de concorrência para resolver problemas é errada sem uma análise aprofundada. A afirmação de regras como a de implementação de um número de *threads* igual ao número lógicos do sistema, também esta provada ser errada. Ignorando o fato da implantação numa única *thread* poder ser mais rentável, máquinas como o i7 utilizado na dissertação contem um número elevado de *threads* disponíveis, e a utilização de todas para resolver um problema pode levar a durações superiores que ao utilizar quantias menores de *threads*.

A criação de múltiplas *threads* é um risco, não só por consumir memória e por ser possível atrasar o objetivo do algoritmo mas também por consumir *threads* ao sistema. Qualquer programador necessita de entender que *threads* são recursos e que estas apesar de demonstrarem capacidades de reutilização e permitirem *context-switching*, estas existem em quantias limitadas. A utilização do mecanismo *Thread* apesar de não ter demonstrado resultados interessantes sobre os restantes mecanismos concorrência, este demonstra simplicidade e a capacidade de obter rendimentos sobre implementações mais casuais.

6.2 *ExecutorService*

O estudo do mecanismo *ExecutorService* permitiu a análise de várias ferramentas. Devido aos resultados obtidos serem um conjunto de pares entre os instrumentos de verificação de estado de *tasks* e *threadpools*, as conclusões foram separadas referenciando cada um destes. A primeira conclusão obtida é que, os diferentes mecanismos tiveram impacto no consumo de memória mas este foi principalmente causado pelas necessidades do algoritmo. Uma vez que o sistema verificação das *tasks* utilizando booleanos foi gerado unicamente por ser uma implementação considerada casual, esta foi ignorada das conclusões finais.

6.2.1 *Threadpools*

Ao todo foram testadas três *threadpools* com o *ExecutorService*, cada uma forneceu um melhor desemprenho sobre determinadas circunstâncias, permitindo assim obter conclusões interessantes sobre cada *threadpool*. A *Fixed threadpool* foi utilizada em 27 casos de estudo, para compreender as vantagens desta, foram comparados os totais da soma de todas durações em cada respetiva implementação, com as mesmas implementações utilizando diferentes *threadpools*. Ao verificar o desempenho em cada situação foram retiradas algumas conclusões.

A utilização da *threadpool* do tipo *Fixed* apresenta os seus melhores resultados quando é implementada num sistema em que as durações das *task* são aproximadas, e que essa duração não seja demasiado curta. Ao verificar os resultados das *tasks* utilizadas nas ordenações foi verificado, que esta *threadpool* é capaz de apresentar resultados vantajosos independentemente do CPU, e que esta consegue por vezes ser a implementação mais vantajosa. A ordenação que permitiu à *Fixed threadpool* sobressair sobre as restantes, foi a *Mergesort*, uma vez que a divisão de trabalho é efetuada quase perfeitamente balanceada, permitindo com que o *work-stealing* não pode-se rentabilizar um grande número de vezes. Apesar do *Pidgeonholesort* executar também uma divisão bastante balanceada, as *tasks* simplesmente são executadas muito rapidamente, permitindo com a utilização de *threadpools* mais dinâmicas tirar maior proveito.

A *threadpool* de *WorkStealing* ao ser analisada apresentou rendimentos superiores nos casos em que a divisão de trabalho não é equilibrada. Ao verificar as durações em cada implementação com esta *threadpool* foi verificado que os melhores momentos foram onde o algoritmo de ordenação era o *Quicksort*. Ao executar um conjunto de *tasks* em que as suas durações diferem, permite a que uma *thread* que termine rapidamente as suas *tasks* "roube" trabalho à *queue* de outras, apressando assim a obtenção do resultado final. A realização de *tasks* de durações curtas neste tipo de *threadpool* não apresentou diferenças significativas, uma vez que estas são realizadas a uma velocidade que torna o *work-stealing* dispensável.

Ao analisar as durações das implementações com a *Cached threadpool*, foi verificado que esta rentabiliza do *hardware* da máquina, ao contrário das restantes *threadpools* onde não são evidenciadas diferenças significativas. A *Cached threadpool* no Dual-Core apresentou em oito das nove comparações o pior resultado, e em nenhuma foi capaz de apresentar o melhor. Isto deve-se uma vez que esta *threadpool* implementa um nível de paralelismo consoante a quantia de trabalho, e aumenta este nível se for considerado necessário. Uma vez que para o nível mais baixo de paralelismo são necessárias duas *threads*, e Dual-Core só permite a execução de duas em simultâneo, esta *threadpool* nunca consegue a poupança de recursos. Nas restantes máquinas com um maior número de processadores lógicos é possível que ocorram situações em que a reutilização de *threads* seja o procedimento mais eficiente. Uma situação que prova esta afirmação encontra-se visível nos resultados da ordenação com *Pidgeonholesort*, uma vez que as suas *tasks* são executadas rapidamente, a utilização de um número menor de *threads* permite benefícios da tecnologia *Turbo Boost,* tornando a implementação desta *threadpool* bastante eficiente.

6.2.2 CountdownLach

O *CountdownLach* é mecanismo de contagem de objetos de concorrência em execução, através deste é possível controlar a quantia de *task* ainda por terminar, fornecendo assim um sistema de verificação de estado. Uma vez que este mecanismo não permite determinar as conclusões das *task* assim que terminem, algoritmos que necessitem de executar operações sobre os resultados das *tasks* serão prejudicados, pois necessitaram de aguardar que todas as *tasks* terminem. Esta conclusão é evidente ao verificar os resultados da utilização do mecanismo *CountdownLach* com os algoritmos de ordenação *Pidgeonholesort* e *Mergesort*. Uma vez que

ambos algoritmos necessitam de agregar os resultados das múltiplas *tasks*, o *CountdownLach* fornece durações mais extensas que ao utilizar o *CompletionService*, e no i7 é possível verificar piores durações que ao utilizar booleanos para verificar os estados das *tasks*. No Dual-Core, uma vez que este fornece um baixo nível de paralelismo, o *CountdownLach* consegue fornecer resultados bastante positivos e até os mais lucrativos. Uma vez que o processador Dual-Core só permite a execução simultânea de duas *threads*, a atribuição de mais trabalho sobre o sistema como o *merge,* e a soma dos "pombos", irão apenas atrasar a execução. Assim ao executar *tasks* em processadores mais limitados no número de *threads* disponíveis, ou algoritmos sem necessidade de operações sobre os resultados das *tasks*, torna-se mais rentável a utilização do *CountdownLach*.

6.2.3 CompletionService

O *CompletionService* funciona de forma oposta ao *CountdownLach*, não necessitando da passagem da variável de estado para as *tasks*, e ao permitir adquirir o estado das *tasks* muito facilmente. Com este funcionamento foi possível obter os melhores rendimentos do *CompletionService* em situações que o algoritmo execute em máquinas com um nível de paralelismo elevado. Ao permitir a execução das *tasks* em simultâneo com a resolução de operações sobre os resultados destas, uma grande quantidade de tempo pode ser rentabilizada. Esta conclusão é evidenciada nas ordenações com *Mergesort* e *Pidgeonholesort* nos processadores i5 e i7, onde o nível de paralelismo permite que a *Main Thread* consiga executar operações de *merge* ou de soma de "pombos". Assim de forma a se obter os melhores resultados com o *CompletionService* é favorável a utilização deste em algoritmos que necessitem de operações sobre resultados de *tasks*, ou em máquinas que permitam um elevado nível de paralelismo.

6.3 Fork/Join

O estudo realizado sobre o *Fork/Join* permitiu a obtenção de conclusões sobre os procedimentos de como rentabilizar a implantação do mecanismo *ForkJoinPool* e das *tasks* associadas. As implementações com estes mecanismos foram verificadas não serem as mais simples, devido à definição do *threshold*. A boa definição do valor de paragem de criação de novas *tasks* é indispensável. A definição deste pode ser executada de forma eficiente realizando o *profiling* de várias iterações e modificando o *threshold* até se obter um conjunto de resultados benéficos. Este tipo de verificação é um processo muito demorado mas necessário, pois a definição incorreta, ou a implementação de um valor genérico para todos tipos de problemas pode levar a atrasos sérios na execução do algoritmo. Esta conclusão foi baseada nos resultados apresentados ao longo dos capítulos 5.1.4, 5.2.4 e 5.3.4. Para a realização da dissertação foi desenvolvida uma fórmula, de modo a ser possível calcular o *threshold* genericamente ao longo dos testes com diferentes dimensões, baseando o cálculo na dimensão do problema e no *hardware* existente. Apesar desta implementação de *threshold* não ter oferecido os melhores resultados, esta forneceu informação suficiente, que ao executar um *profiling* mais aprofundado poderia tornar a implementação rentável. A utilização da fórmula levou à obtenção de durações superiores ao caso *Single Thread* o que indica um progresso na implementação do mecanismo

de concorrência, e em determinados casos consegui ser o melhor caso mas devido à máquina que executa. Apesar do *threshold* ser relevante na implementação do mecanismo, é necessário compreender que a *ForkJoinPool* é uma *threadpool* que implementa *work-stealing*. Isto faz com que a implementação deste mecanismo realize as suas operações de forma mais benéfica, nas situações pretendidas para as *threadpools* do tipo *WorkStealing*.

6.4 Trabalho futuro

É possível dar continuidade ou até melhorar a qualidade do estudo apresentado na dissertação, aprofundando a investigação ao executar os testes produzidos em máquinas com processadores que incorporem uma capacidade de *threads* ainda não testada. Computadores com processadores de quatro núcleos físicos seriam capaz de criar uma ponte de conexão entre os múltiplos resultados de cada processador, ao possibilitar a aquisição de resultados em processadores com dois e quatro núcleos físicos, com e sem a capacidade de *Hyper-Threading*. Outra melhoria possível de implementar seria a realização do estudo num SO que permitisse o *affinity* de um processo com maior facilidade, permitindo uma dedicação superior do CPU na resolução dos testes. A implementação de outras ferramentas, existentes na extensa API do Java 8, permitiriam a obtenção de mais conclusões que auxiliaria a programação de algoritmos com necessidades de concorrência. A execução de testes num ambiente completamente empresarial, com a utilização de um algoritmo com uma única função específica, com múltiplas implementações concorrentes, seria um caso ideal para obtenção de conclusões.

Referências

[Brian Goetz et al., 2006]	Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holme.s, Doug Lea. Java Concurrency in Practice, 2006
[Cay S. Horstmann, 2014]	Cay S. Horstmann. Java SE 8 for the Really Impatient: A Short Course on the Basics (Java Series), 2014
[Cláudio Maia, et al., 2011]	Cláudio Maia, Luís Nogueira, Luís Miguel Pinho. Combining RTSJ with Fork/Join: A Priority-based Model, 2011
[Clay Breshears, 2009]	Clay Breshears. The Art of Concurrency: A Thread Monkey Guide to Writting Parallel Applications, 2009. ISBN 0-596-52153-7
[Douglas Lea, 1999]	Douglas Lea. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition, 1999. ISBN 0-201-31009-0
[Intel Xeon Phi, 2015]	http://ark.intel.com/products/75800/Intel-Xeon-Phi-Coprocessor-7120X-16GB-1_238-GHz-61-core [último acesso: Set 2015]
[JavaSE 8, 2015]	http://docs.oracle.com/javase/8/ [último acesso: Abr 2015]
[Javier Fernández González, 2012]	Javier Fernández González. Java 7 Concurrency Cookbook, 2012
[Petter Andersen Busterud, 2012]	Petter Andersen Busterud. Investigating Different Concurrency Mechanisms in Java, 2012. Master thesis
[Richard H. Carver and Kuo-Chung Tai, 2005]	Richard H. Carver, Kuo-Chung Tai. Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs, 2005. ISBN: 978-0-471-74416-0

Anexos

Anexo A	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/A.png
Anexo B	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/B.png
Anexo C	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/C.png
Anexo D	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/D.png
Anexo E	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/E.png
Anexo F	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/F.png
Anexo G	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/G.png
Anexo H	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/H.png
Anexo I	https://myisepipp-my.sharepoint.com/personal/1090554_isep_ipp_pt/Documents/Shared%20with%20Everyone/I.png