Mestrado em Engenharia Informática
Dissertação
Relatório Final

Otimização de Programas em Runtime na Plataforma Aeminium

Cristiano Filipe Matos Gonçalves
matos@student.dei.uc.pt

Orientador:
Doutor Bruno Cabral
Mestre Alcides Fonseca
Data: 3 de Julho de 2013

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Resumo

Na última década assistiu-se a um desaparecimento gradual dos processadores de um único *core*, sendo estes progressivamente substituídos por *multicores*. Este desenvolvimento tecnológico urgiu a necessidade de serem estabelecidas novas adaptações pelos programadores, uma vez que têm que deixar de "pensar" sequencial para "pensar" em paralelo. Com a finalidade de facilitar a criação de código paralelo, foram criadas várias ferramentas – compiladores, *runtimes*, *frameworks* e bibliotecas, que rentabilizam o tempo de criação de código e ajudam a evitar problemas inerentes a este tipo de programação.

O projeto Aeminum pretende a criação de uma linguagem de programação que seja concorrente por omissão. Escreve-se código sequencial e o compilador Aeminium fazendo uso de métodos automáticos gera código JAVA paralelo, já com tarefas e suas dependências, para depois ser executado pelo *runtime* do Aeminium em paralelo.

Infelizmente, alguns programas quando compilados e executados na *framework* Aeminium, apresentam uma diminuição de performance comparativamente com a sua versão sequencial. Neste estágio propôs-se validar três hipóteses relacionadas com a performance desta *framework* ao nível do *runtime*:

1. Um escalonamento mais equilibrado das tarefas em execução trará melhorias na performance;
2. Modificar o nível de paralelismo entre as tarefas em execução de acordo com a carga e capacidade do sistema trará melhorias de performance;
3. O controlo da granularidade das tarefas em execução e, consequentemente, do número de tarefas a executar trará melhorias de performance.

Para validar a primeira hipótese, foi proposto a implementação de uma política de inserção que tentasse balancear a inserção de tarefas. Conseguindo-se assim, um aumento de performance até 20%. Na segunda hipótese, foi proposto auxiliar a decisão de paralelização, com informação de várias filas de execução em vez de apenas uma. Obtendo-se aumentos de performance até 55% em algumas das políticas. Por último, foi proposto a criação de um módulo de agregação de tarefas para validar a terceira hipótese. Esta implementação atingiu aumentos de performance até 30%. Através da implementação das hipóteses apresentadas, tentou-se obter os valores óptimos da sua utilização e extracção de possíveis políticas de optimização a implementar em *runtimes*.

Palavras-Chave

Computação paralela, *runtimes* baseados em tarefas, políticas de optimização em *runtimes*, otimização no *scheduler*.

Agradecimentos

Para a conclusão deste estágio foi necessário um grande número de sugestões, comentários e críticas. Sendo estes pontos imprescindíveis para a melhoria possível do presente documento.

Assim, gostava de agradecer ao Professor Doutor Bruno Cabral, por todo o apoio prestado no decorrer do estágio, pois sem ele o estágio não teria chegado a este ponto.

Outro agradecimento especial para o Mestre Alcides Fonseca pela sua disponibilidade em me auxiliar sempre que necessitei.

Um agradecimento aos meus colegas do projecto Aeminium.

Índice

Lista de Figuras

Lista de tabelas

Glossário

Bytecode	É um código intermédio entre o código fonte e a aplicação final. Diferente do código máquina este necessita ser interpretado numa máquina virtual para ser executado
Deadlock	Quando duas ou mais *threads* estão à aguardando umas pelas outras por alguma ação ou recurso
Debug	Processo para pesquisa de erros durante a execução de um determinado programa
Multicore	Processador com várias unidades de processamento – chamada "core"
Overhead	Uso de um recurso (armazenamento ou processamento) em excesso, para se executar uma determinada tarefa. Uma das consequências é piorar o desempenho. Normalmente o seu uso tem como objetivo
Profiler	É uma forma de análise dinâmica de programas, com o intuito de medir variáveis relacionadas com o programa em execução (ex: memória, uso de determinadas funções, etc). O uso da informação coletada tem como objetivo a otimização de programas.
Race conditions	Quando várias *threads* ou processos competem pelo mesmo recurso
Runtime	Software que suporta a execução de programas
Speedup	Velocidade de um algoritmo paralelo em relação ao seu sequencial (quantas vezes o algoritmo paralelo é mais rápido que o sequencial)
Starvation	Acontece quando uma *thread* espera que um evento ocorra ou que outra *thread* liberte um lock
Thread	Fluxo de execução num processo
Throughput	Quantidade de dados transferidos ou quantidade de dados processados em um determinado espaço de tempo

Tabela 1 - Glossário

Tabela de Acrónimos

CISUC	*Center for Informatics and Systems of the University of Coimbra*
CMU	*Carnegie, Mellon, University*
DAG	*Directed, Acyclic, Graph*
I/O	*Input, Output*
JIT	*Just, In, Time*
JVM	*Java, Virtual, Machine*
MPI	*Message, Passing, Interface*

Tabela 2 - Acrónimos

Capítulo
1 Introdução

Este capítulo faz uma introdução ao contexto em que o estágio se encontra inserido, as principais motivações, objetivos e contribuições esperadas. No final é apresentada uma proposta para a estrutura deste documento.

1.1 Motivação

O mercado rumou de processadores de um *core* para *multicores*, pois já não é viável [1] melhorar a performance dos primeiros. Isto, devido às capacidades dos materiais utilizados na sua construção não resistir a temperaturas mais elevadas e não ser prático a inclusão de componentes de arrefecimento mais potentes em máquinas domésticas. Além dos problemas mencionados associam-se ainda alguns problemas relacionados com o consumo [8] e perdas de energia. Este cenário tornou-se um enorme problema para os programadores, já que estes novos processadores permitem computação paralela e os torna complexos de programar. Para que se possa tirar o máximo partido de todos os processadores, o *software* deve estar escrito de forma paralela, ou seja, as tarefas (instruções a ser executadas) que não são dependentes, podem e devem, ser executadas ao mesmo tempo, em diferentes processadores. Mas, para isso os programadores necessitam [8] de lidar com as comunicações, memória partilhada e sincronização, o que pode levantar vários problemas, criando custos mais elevados para as empresas. O atual desafio é, em suma, encontrar soluções/ferramentas que permitam facilitar este trabalho.

A *framework* Aeminium vem colmatar esta necessidade, habilitando o programador a escrever código sequencial que depois é automaticamente paralelizado. No entanto, verifica-se uma diminuição de performance de certos programas paralelizados nesta *framework* comparativamente com a sua versão sequencial. Esta situação ocorre devido à dificuldade de identificar o número mais adequado de tarefas concorrentes, a dimensão das tarefas, utilização de dependências, tipo da própria tarefa (I/O ou *CPU Bound*), balanceamento das tarefas por processador e decisão de paralelizar ou sequencializar o código em execução. Existe assim uma necessidade de obter as soluções óptimas – criação de políticas de optimização, para as características citadas e a partir destas desenvolver-se um módulo de otimização de programas para o *runtime* Aeminium que maximize a performance em função do *hardware* disponível.

1.2 Contexto do Estágio

Este estágio está integrado no projeto "Aeminium – Freeing Programmers from the Shackles of Sequentially" que tem como objetivo a implementação de uma *framework* de desenvolvimento de aplicações paralelas. O projeto Aeminium é financiado pelo programa CMU|Portugal e resulta de uma parceria entre a Universidade de Carnegie Mellon nos Estados Unidos da América, Universidade de Coimbra, Universidade da Madeira e a empresa Novabase, em Portugal. Em Coimbra os trabalhos decorrem nos laboratórios do CISUC no grupo de "Software and Systems Engineering".

1.2.1 Aeminium

A ideia principal do projeto Aeminium é a criação de uma linguagem de programação concorrente por omissão e respectiva *framework*. Esta *framework* é constituída por dois componentes principais: compilador e *runtime*. A investigação levada a cabo no CISUC mostrou não ser necessário uma linguagem de programação dedicada para se obter paralelismo implícito. Assim, a equipa Aeminium da UC desenvolveu um compilador (J2JPar) que traduz JAVA sequencial para uma versão altamente paralela compatível com o *runtime* Aeminium. A figura 1 apresenta o funcionamento geral desta *framework*.

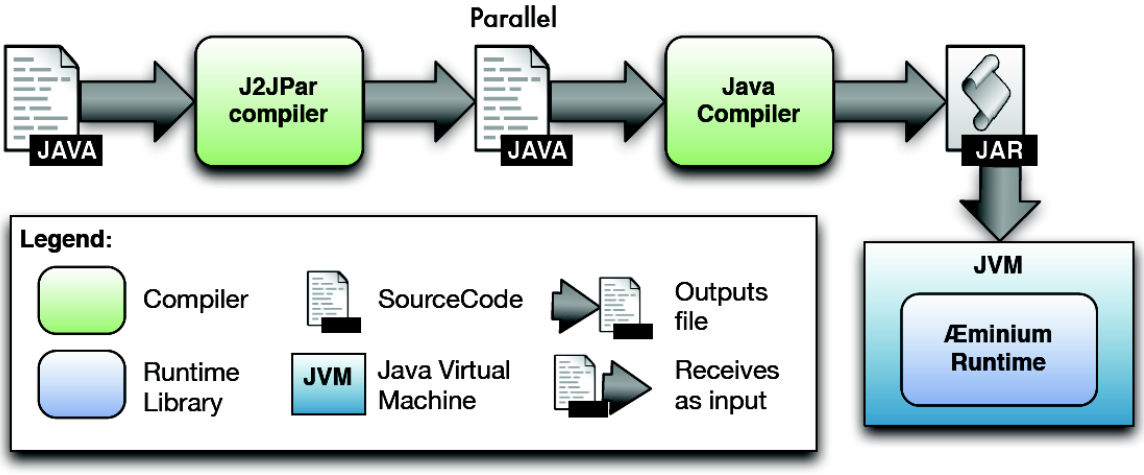

Fig. 1 - Arquitectura do Aeminium. Fonte: [23]

O funcionamento pode ser resumido a três passos:

1. O compilador J2JPar traduz o código fonte em JAVA sequencial para código JAVA paralelo, já com tarefas e suas dependências;

2. O compilador JAVA compila o código JAVA para *bytecode*, enquanto verifica conformidade com a interface da biblioteca do *runtime* Aeminium;
3. O *bytecode* JAVA é executado na *Java Virtual Machine* e utiliza através de chamadas a biblioteca do *runtime* Aeminium.

1.2.2 Limitações do Compilador Aeminium

O compilador JAVA tem como objetivo fazer a análise lexical (dividir o programa em *tokens*), *parsing* (dependências existentes no código) e semântica (detetar inconsistências).
Por sua vez, o compilador Aeminium tem como principal função a criação das tarefas e suas dependências. Um dos problemas verificado é que este cria tarefas para quase tudo o que existe no código. Por exemplo, num ciclo "for de i=0 até i<100", este pode criar 100 tarefas, uma para cada iteração, mesmo que cada iteração faça apenas um simples cálculo. Esta criação exagerada de tarefas pode levar a um decréscimo de performance, uma vez que existe *overhead* na criação das tarefas, podendo não ser compensatória a criação de uma tarefa para cada operação.

Independentemente do código a ser executado no *runtime* Aeminium ter sido criado pelo seu compilador ou por um humano, surge aqui uma oportunidade para se poder optimiza-lo. Estas oportunidades podem surgir tanto pelo número elevado ou reduzido de tarefas, dimensão das tarefas, distribuição de carga pelos *cores*, escalonamento das tarefas, utilização das dependências ou distribuição do I/O pelas tarefas.

1.3 Objetivos

Os objetivos deste estágio são:

- Compreender e descrever os problemas de performance na execução de programas paralelos no *runtime* Aeminium;
- Realizar experiências, recolher e analisar dados de forma a perceber que politicas devem ser implementadas por um otimizador de programas para o *runtime* Aeminium;
- Desenvolver um módulo de optimização de programas paralelos para o *runtime* Aeminium;
- Validar as diferentes políticas de optimização criadas.

1.4 Contribuições

As principais contribuições deste trabalho são as seguintes:

- Apresentação dos resultados de um estudo empírico sobre como o número / tamanho das tarefas, decisão de paralelização e forma de escalonamento influencia a performance dos programas paralelos organizados em DAGs;
- Desenvolvimento e validação de diferentes políticas de optimização de programas em *runtimes*, baseados nos resultados do estudo apresentado;
- Criação e publicação de um módulo de optimização de programas no *runtime* Aeminium.

1.5 Estrutura da proposta de estágio

Este documento está organizado da seguinte forma:

- Primeiro capítulo: Introdução ao contexto do trabalho apresentado, motivação, objetivos e contribuições;
- Segundo capítulo: Apresentação do estado da arte em que se insere o presente estágio;
- Terceiro capítulo: Conceitos de base sobre a plataforma Aeminium (utilização e funcionamento interno);
- Quarto capítulo: Identificação e validação das hipóteses para investigação;
- Quinto capítulo: Apresentação da optimização efetuada no escalonador, com detalhes da implementação e validação;
- Sexto capítulo: Apresentação da otimização da decisão de paralelizar, possíveis políticas e validação;
- Sétimo capítulo: Implementação do módulo de agregação de tarefas, detalhes de implementação e validação;
- Oitavo capítulo: Apresentação do plano de trabalhos e riscos associados ao estágio;
- No final são apresentadas as conclusões retiradas para cada uma das hipóteses e ideias para trabalho futuro a efectuar no *runtime* Aeminium.

Capítulo
2 Conceitos de base e estado da Arte

Neste capítulo serão referenciados trabalhos de outros autores que se relacionam com o tema deste estágio. Estes podem ser abordagens concorrentes ou trabalhos que fundamentam e validam a nossa abordagem. Inicialmente irão ser abordados conceitos gerais como concorrência, escalonamento e tipos de abordagem usadas pelas linguagens de programação para a concorrência. No final serão apresentadas plataformas para programação paralela, técnicas para se retirar paralelismo de código sequencial e possíveis optimizações em *runtime*.

2.1 Concorrência

O objetivo da concorrência é produzir sistemas mais responsivos, reduzir latência e aumentar o *throughput*. Criando aplicações mais rápidas e criando uma melhor experiência ao utilizador.

As *threads* são consideradas um bom mecanismo para explorar o paralelismo em *multicores*. No entanto lidar com recursos partilhados pode-se tornar complexo para o programador, uma vez que tem que tratar da coordenação para evitar problemas como *starvation*, *race conditions* ou *deadlocks*.

O grande desafio é, assim, obter performance com um custo/esforço de programação baixo. Compiladores e *runtimes* que auxiliem a paralelização podem ajudar a minorar o dito custo/esforço no desenvolvimento de *software* paralelo.

2.2 Escalonamento

Para a execução eficiente da computação *multithread*, um algoritmo de escalonamento deve assegurar que existem *threads* ativas suficientes para manter o CPU ocupado e tentar manter *threads* relacionadas no mesmo processador para que os problemas de comunicação sejam minimizados. Além disso deve tentar que o número de *threads* concorrentes em estado ativo se mantenha com limites razoáveis para que os requisitos de memória não sejam demasiado grandes [9].

Alguns dos paradigmas de escalonamento que aparecem para tratar do escalonamento multithread são: *work sharing* e *work stealing*.

No *work sharing* é criado uma [21] *pool* de *threads* para executar as tarefas do programa. As tarefas que são criadas pelo programa são enviadas para uma fila central, onde

as *threads* vão exercer o seu trabalho. Um dos problemas associados a este tipo de escalonamento está relacionado com o facto de que, como todas as *threads* vão exercer o seu trabalho numa fila central, esta pode-se tornar um *bottleneck*. Outro problema decorre da maneira como são tratadas as tarefas suspensas, já que quando uma tarefa suspende a *thread* que a executa também se suspende e para a compensar é criada uma nova *thread* (grande *overhead* de criação/destruição de *threads*), que pode levar a número de *threads* bastante superior ao número de cores disponíveis.

O *Work stealing* é baseado na ideia de desacoplar as tarefas das *threads*, criando para isso um conjunto fixo de *threads* – normalmente uma *thread* para cada processador, que por sua vez exercem tarefas para executar. Nesta técnica os processadores com baixa utilização, tomam a iniciativa e "roubam" *tasks* a processadores com elevado trabalho, permitindo um balanceamento dinâmico do trabalho.

Uma das diferenças entre estes algoritmos é que a migração de *threads* ocorre muito mais frequentemente no *work sharing* que no *work stealing*, uma vez que no último todos os processadores têm trabalho e caso não o tenham, vão procura-lo.

2.3 Abordagens na programação concorrente

Nas linguagens de programação existem dois tipos de abordagens usadas para a programação concorrente: implícita e explícita. De seguida são apresentadas com mais detalhe, cada uma delas.

2.3.1 Abordagem com concurrência explicita

Num sistema concorrente explícito, o programador cria [1] e gere a execução concorrente de modo ativo. Existe uma gestão manual da execução concorrente. Este tipo de abordagem é o mesmo que se pode encontrar ao escrever código JAVA para o *runtime* Aeminium. Aqui pode-se decidir as dependências entre tarefas, quando uma tarefa deve ou não ser atómica, o número de tarefas que se criam, entre outras.

2.3.2 Abordagem com concurrência implicita

No caso de um sistema implícito onde se insere a *framework* Aeminium - compilador e runtime - o utilizador não necessita de escrever o código de modo paralelo. Neste tipo de abordagem o programador escreve o código com as instruções que devem ser executadas e o sistema decide a melhor maneira de o executar mais eficientemente.

2.4 Paralelização automática

A paralelização automática de um código sequencial para código paralelo é um pouco limitada nas linguagens existentes. Algumas plataformas possuem concorrência explícita, tendo que o programador através de anotações indicar o paralelismo ao compilador e outras nos melhores casos são parcialmente implícitas, retirando paralelismo de algumas estruturas de dados tais como: *arrays*, matrizes e *loops*.

2.4.1 Linguagens paralelas por defeito

Algumas das linguagens paralelas por defeito são:

- CILK
- X10
- Chapel
- Fortress

Seguidamente será efetuada a apresentação do CILK.

CILK

O CILK é uma extensão de C/C++ para suportar tarefas paralelas. Esta extensão tem como filosofia que o programador deve concentrar-se na estrutura do programa para expor o paralelismo - identificar os elementos que podem executar com segurança em paralelo, deixando o *runtime* do CILK com a responsabilidade de escalonar a computação para correr eficientemente numa determinada plataforma. Além disso trata de assuntos relacionados com *load balancing*, sincronização e protocolos de comunicação.

Os dois comandos principais usados pelo CILK são:

- *spawn*: Indica que um procedimento pode executar em paralelo;
- *sync*: Indica que a execução não pode continuar até que os procedimentos anteriores com *spawn* estejam concluídos.

O *workload balancing*, ou seja distribuição de tarefas pelos *workers* é efectuado através do algoritmo *work stealing*. Na figura 2 é apresentado um exemplo do Fibonacci programado em CILK.

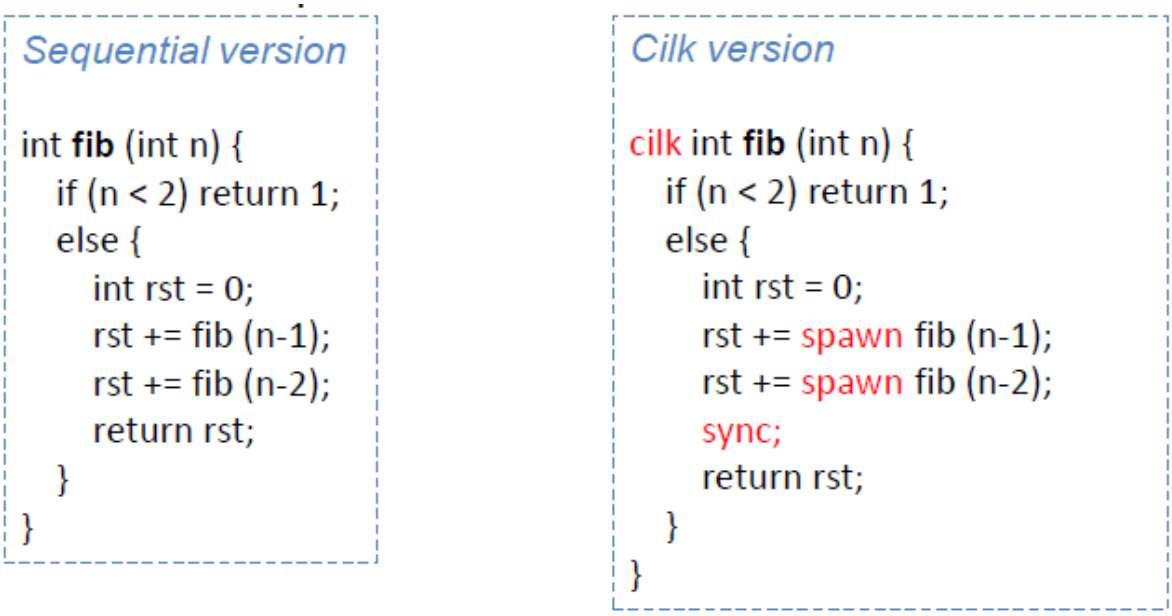

Fig. 2 - Exemplo do Fibonacci em Cilk. Fonte: [27]

O comando *cilk* indica que é um procedimento e tem que ser inserido na função *main*. O comando *spawn* indica que o procedimento filho pode ser executado em paralelo no procedimento pai e o comando *sync* faz esperar até que todos os filhos tenham completado – apenas os filhos do pai corrente. O comando *sync* não é obrigatório uma vez que o compilador insere este comando por *default* antes do *return*.

2.4.2 Linguagens sequenciais executadas em paralelo

Actualmente existem poucas linguagens implícitas. A plataforma Aeminium com a sua especificidade nesta área vem tentar colmatar esta necessidade. Algumas das plataformas existentes são listadas de seguida:

- HASKELL
- JAVAR
- SISAL
- Par4All

De seguida é efetuada uma breve apresentação da plataforma JAVAR e SISAL.

JAVAR

É um compilador escrito em C, que transforma automaticamente [26] código com paralelismo implícito em paralelismo explícito através de *multi-threading*. A detecção de paralelismo é efetuada de modo automático, mas o programador e também pode inserir anotações para identificar paralelismo. A detecção automática é efetuada pelo compilador

através da identificação de *loops* e métodos recursivos na tentativa de explorar paralelismo implícito A identificação de paralelismo efetuada pelo programador é efetuada através do uso de anotações, como por exemplo usar /*par*/ como comentário de um ciclo. O funcionamento do JAVAR (fig. 3) pode ser resumido nos seguintes passos:

- Um ficheiro com extensão .java é usado como input no compilado JAVAR. O compilador JAVAR transforma o ficheiro .java numa forma com *multi-theading*(código JAVA com *threads*). Tornando o paralelismo implícito em explícito;
- O ficheiro .java é compilado em *bytecode* por um compilador Java e executado na JVM.

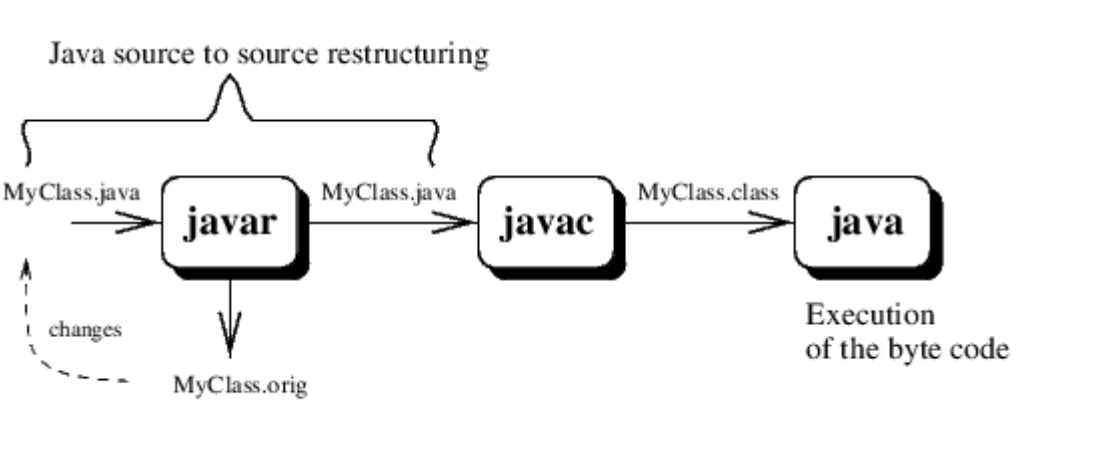

Fig. 3 - Funcionamento do JAVAR. Fonte: [26]

O JAVAR é um protótipo e não fornece um Java *front-end* completo. Um exemplo desta carência é por exemplo a falta de suporte para escape caracteres. Uma interessante característica é a criação de apresentações em HTML de programas Java.

SISAL

Sisal é uma [29] linguagem de programação funcional, combinando características das linguagens modernas com sintaxe legível e bases sólidas de matemática. Foi definida em 1983 por James McGraw na Universidade de Manchester. Uma interessante vantagem do Sisal é que o código Sisal pode ser misturado com C ou Fortran, criando aplicações híbridas. Além disso é uma boa linguagem para desenvolvimento rápido, uma vez que os programas tendem a ser mais pequenos, mais fáceis de entender e modificar.

2.5 Retirar paralelismo de código sequencial

Pode-se retirar paralelismo do código de dois modos: manual e automático. Manualmente realizado pelo programador e de forma automática pelo compilador. Sendo que quando efectuado pelo programador é requerido mais esforço, uma vez que este precisa de ter em atenção problemas associados à programação paralela anteriormente citados – *race conditions*, *deadlocks*, entre outros.

De modo automático, o compilador através da análise do código [14] pode encontrar paralelismo que nem sempre é óbvio para um programador experiente. Mas para isso, o compilador tem de explorar regiões do código através de análise para depois paraleliza-lo. A análise é a base para a paralelização automática, que visa encontrar vários tipos de dependências entre as várias partes de um programa - dependências de dados (escrita/leitura da mesma variável) e dependências de controlo (prioridade entre dois statements).

Algumas das técnicas usadas para extração automática são DOALL, DOACROSS, DOPIPE e *decoupled Software pipelining*(DSWP).

2.5.1 DOALL

Um *loop* pode-se dizer DOALL [15] se todas as suas iterações em qualquer invocação poderem ser executadas simultaneamente. O *loop* pode assim ser dividido em vários grupos e executado em paralelo. Esta é uma das paralelizações mais simples de se efetuar. Na figura 4 apresenta-se um exemplo desta paralelização.

```
int a[N], b[N], c[N];
for(i=0;i<N;i++){
  c[i] = a[i]+b[i];
}
```

(a)

```
Sequential program variables:
  int a[N], b[N], c[N], tid[M];

Parallel code:
  void doall(int *tid){
    int id = *tid;
    for(i=id*N/M;i<(id+1)*N/M;i++){
      c[i] = a[i]+b[i];
    }
  }
```

(b)

Fig. 4 - Exemplo DOALL a)Programa Original b)programa paralelizado. Fonte: [14]

2.5.2 DOACROSS

Ao contrário dos *loops* DOALL [20] que não impõem nenhuma ordem de execução entre as iterações do *loop*, os *loops* DOACROSS impõem uma ordem de execução parcial, no sentido de que algumas iterações são forçadas a esperar a execução parcial ou completa de algumas iterações anteriores. Sendo normalmente utilizados para paralelizar *loops* dependentes das iterações anteriores. Os *loops* DOACROSS fazem uso de conhecimento a-priori para realizar as sincronizações necessárias.

2.5.3 PIPELINE

Um *pipeline* é uma cadeia de elementos [14] onde um elemento anterior gera um *input* que vai ser processado pelos elementos seguintes. Neste tipo de paralelização muitas vezes é necessário *buffering* na comunicação entre as *threads*. Duas técnicas deste tipo de paralelização são apresentadas de seguida: DOPIPE e *Decoupled Software Pipelining*.

DOPIPE

O DOPIPE foi originalmente proposto para paralelizar código com recorrências. Este divide o *loop* original em várias partes e distribui-o pelas *threads*. As dependências entre as *threads* são forçadas a ser unidirecionais, isto é, não existem dependências cíclicas entre as *threads*.

Decoupled Software pipelining(DSWP)

O DSWP, tal como o DOPIPE, apenas trabalha se um *loop* poder ser dividido em várias partes que não formam dependências cíclicas. Este particiona as instruções de um *loop* ao longo de uma sequência de loops. O DSWP difere do DOPIPE por usar filas na comunicação entre as *threads*.

2.6 Otimizações em runtime

Tradicionalmente as técnicas de otimização eram centradas nas transformações do compilador para expor o paralelismo. Os problemas associados às otimizações efetuadas apenas no compilador são:

- Serem estáticas: não têm em conta as condições do *runtime*, mas um computador paralelo apresenta um ambiente bastante variável, tanto ao nível da disponibilidade de recursos como ao nível da transmissão de mensagens;
- Serem imprevisíveis: possuem necessidades computacionais imprevisíveis e padrões de comunicação que não podem ser deduzíveis a partir de uma análise estática do programa paralelo;

- Compilação ser modular: se um programa paralelo é composto por módulos compilados separadamente, o compilador não possui informação global suficiente para otimizar o programa. Como exemplo temos as decisões de *load balancing* que não podem ser feitas por um módulo isolado, uma vez que a carga sobre o processador está afetada por cálculos em todos os módulos.

Além disto as otimizações no runtime podem tratar de muitos casos onde as transformações do compilador são inadequadas, tais como:

- A compilação pode ser otimizada para a ocupação do CPU alvo (ex: agregação / desagregação de tarefas, tal como o tentado neste estágio);
- O sistema ser capaz de recolher estatísticas do programa a correr e assim rearranjar e recompilar para melhorar a performance. No entanto alguns compiladores estáticos podem usar *profile information* como *input*.

2.6.1 JIT Just-In-Time compiler

A *Java Virtual Machine* [3] usa o *Just-In-Time* (JIT) *Compiler*, para aumentar a sua performance, visto este sistema ter vantagens sobre compiladores estáticos, já que as decisões podem ser feitas *on-the-fly* e adaptar-se às características do programa em execução. A sua especificação define uma arquitetura abstrata para executar programas compilados em diferentes plataformas. O formato binário do código para a JVM é chamado de *bytecode* e armazenado em ficheiros com extensão .class.

O JIT é usado para melhorar a performance dos programas de computador e normalmente oferece melhor performance que os interpretadores. Basicamente é um tradutor em tempo de execução que converte código num formato para outro. Como por exemplo *bytecode* para código máquina em tempo de execução.

O JIT é um compilador dinâmico [2] que pode explorar as informações do perfil do *runtime*, possuindo informações do processador que se possui, o número de cores, registos disponíveis, sistema operativo, estado da memória usada atual, entre outros.

Algumas das otimizações efetuadas pelo JIT são o *inline method*, *remove redundant loads*, *copy propagation*, *eliminate dead code*.

Apesar das implementações efetuadas neste estágio diferirem bastante do tipo de optimizações efetuadas pelo JIT, fez-se um estudo do mesmo, para se conhecer os vários tipos de optimização possíveis de efectuar em *runtime*.

2.6.2 Thread-level Speculation

É uma técnica usada para explorar paralelismo no *runtime*. Executa *threads* dependentes [24], adiando a verificação se as dependências são violadas para o *runtime*. Tem com ideia básica [25] executar o acesso a determinado dado assumindo que não existe dependência, se a especulação não é segura, invoca uma acção de recuperação (*rollback*), tendo que a *thread* dependente voltar a executar.

Capítulo
3 Conhecimentos base sobre a plataforma Aeminium

Neste capítulo é apresentado o *runtime* Aeminium. Através de exemplos de código é exposto como se utiliza esta *framework* de forma manual e de seguida é feita uma descrição do funcionamento interno, com recurso a diagramas UML.

3.1 Utilização do runtime Aeminium

O *runtime* Aeminium possibilita a criação de código paralelo de forma manual, tal como algumas *frameworks* existentes (ex: Cilk). É possível criar tarefas e gerir as suas dependências. Através da análise destas dependências o *runtime* Aeminium decide quais as tarefas que podem ser executadas paralelamente. De seguida é realizada uma breve introdução aos seguintes pontos:

- Inicialização e paragem;
- Criação de tarefas;
- Utilização das dependências.

Para se utilizar o *runtime* Aeminium deve-se começar por criar uma instância, usando o seguinte código:

```
Runtime rt=Factory.getRuntime();
```

Seguidamente pode-se visualizar a inicialização e paragem do *runtime*. Entre a inicialização e paragem do *runtime* será onde se vai encontrar o código pretendido.

```
Final Runtime rt=Factory.getRuntime();
rt.init();
//código desejado escrito aqui
rt.shutdown();
```

Para se iniciar a paralelização de código é necessário a criação de *Tasks*, sendo cada *Task* constituída por um *Body*. O *Body* é a classe onde se encontram as instruções em Java que pretendemos executar. Na figura 5 é apresentada através de uma ilustração, uma tarefa com o seu *body* a depender de outra tarefa.

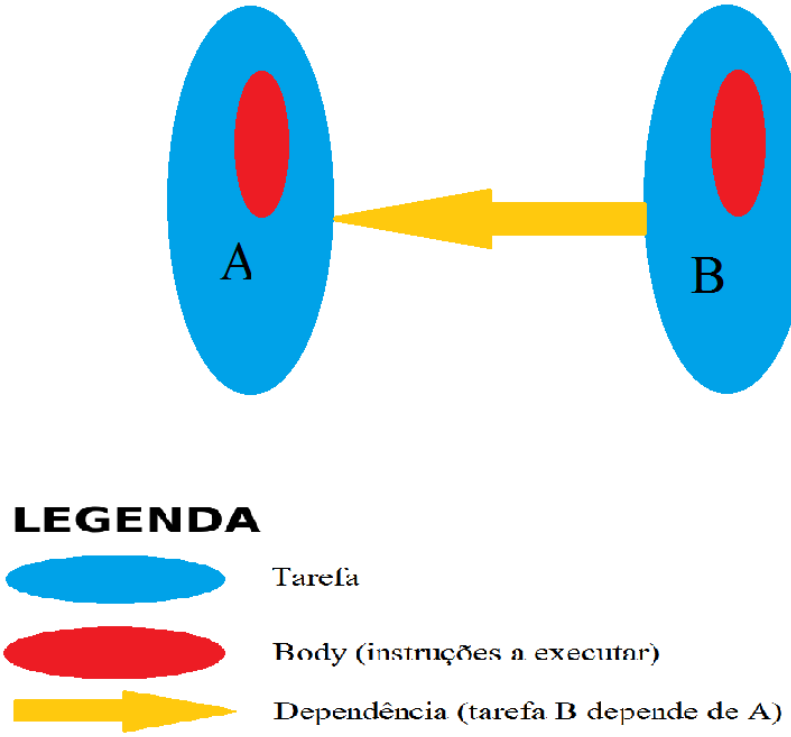

Fig. 5 - Tarefa, Body e dependência

Como se pode visualizar a tarefa possui no seu interior o *Body* e a tarefa B depende de A, isto quer dizer que apenas pode ser invocada sua execução depois da tarefa A terminar. Posteriormente será apresentado o uso destas dependências. O *Body* é passado como argumento à classe *Task*, juntamente com os *Hints*, informação que se pode transmitir ao *runtime*. Um exemplo da criação de uma *Task* é apresentado no código seguinte:

```
Task task = rt.createNonBlockingTask(new Body(){
      @Override
      Public void execute(Runtime rt, Task current){
            //código escrito aqui

},(short)Hints.NO_HINTS);
```

Apesar de no exemplo anterior, encontrar-se apenas a representação da criação de uma *NonBlockingTask*, existem vários tipos de tarefas, tais como: *BlockingTasks* e *NonBlockingTasks*. Estes tipos são usados para informar o *runtime* se uma tarefa pode ou não bloquear. Alguns exemplos dos *Hints* citados anteriormente e que foram adicionados no decorrer do estágio são:

- O tamanho das tarefas (ex: Hints.SMALL);
- Se a tarefas contêm loops (ex: Hints.LOOPS);
- A tarefa não tem dependentes (ex: Hints.NO_DEPENDENTS);
- A tarefa não possui tarefas filhas (ex: Hints.NO_CHILDREN);
- A tarefa é recursiva (ex: Hints.RECURSION);
- Caso não se pretenda transmitir nenhuma informação (ex: Hints.NO_HINTS).

Este tipo de informação é importante para que o *runtime* possa tomar as melhores opções, aquando do escalonamento das tarefas, um exemplo é o *merge* de tarefas abordado nesta investigação. Após a criação da *Task* é necessário efetuar o seu escalonamento, usando para isso a seguinte função:

```
rt.schedule(task,Runtime.NO_PARENT,Runtime.NO_DEPS);
```

Como primeiro parâmetro é passada a *Task* a escalonar, em segundo a *Task* pai e por último uma colecção de *Tasks* (dependências). De notar, que nesta função também é possível transmitir alguns *Hints*:

- Caso a tarefa não possua pai: Runtime.NO_PARENT;
- Caso não possua dependências: Runtime.NO_DEPS.

Um exemplo de como se pode utilizar o *runtime* Aeminium para paralelizar código sequencial é apresentado de seguida. Neste exemplo é utilizado o código para cálculo dos números de Fibonacci. No primeiro bloco, o código original e no segundo o código paralelo.

```
public static void main(String[] args) {
        int fib = 20;
        long val = seqFib(fib);
        System.out.println("F(" + fib + ") = " + val);
}

public static long seqFib(long n) {
    if (n <= 2)
        return 1;
    else
        return (seqFib(n - 1) + seqFib(n - 2));
}
```

Como se pode visualizar, o código sequencial é constituído pelo uso de chamadas recursivas à função seqFib(long n).

```
public static class FibBody implements Body {
    public volatile long value;

    public FibBody(long n) {
        this.value = n;
    }

    @Override
    public void execute(Runtime rt, Task current) {
        if (value <= 2) {
            value = 1;
        } else {
            FibBody b1 = new FibBody(value - 1);
            Task t1 = rt.createNonBlockingTask(b1, Hints.RECURSION);
            rt.schedule(t1, Runtime.NO_PARENT, Runtime.NO_DEPS);

            FibBody b2 = new FibBody(value - 2);
            Task t2 = rt.createNonBlockingTask(b2, Hints.RECURSION);
            rt.schedule(t2, Runtime.NO_PARENT, Runtime.NO_DEPS);

            t1.getResult();
            t2.getResult();
            value = b1.value + b2.value;
        }
    }
}

public static void main(String[] args) {
    Runtime rt = Factory.getRuntime();
    rt.init();

    int fib = 20;
    FibBody body = new AeFibonacci.FibBody(fib);
    Task t1 = rt.createNonBlockingTask(body, Runtime.NO_HINTS);

    rt.schedule(t1, Runtime.NO_PARENT, Runtime.NO_DEPS);
    rt.shutdown();
    System.out.println("F(" + fib + ") = " + body.value);
}
```

Para o caso paralelo é criada a classe *FibBody* que implementa a classe *Body*. No execute é inserido o código para o cálculo dos números, criando para isso duas *Tasks*. Uma para calcular o *value* -1 e outra, o *value* -2, no final o *value* do próprio *Body* é igual á soma dos dois. Tal como no código sequencial se o *value* for igual ou menor a 2 então o *value* do próprio *Body* é 1. A função getResult() é utilizada para esperar a finalização da execução das tarefas e então só depois se soma os valores resultantes.

Através dos exemplos dá para perceber que a utilização do *runtime* Aeminium é bastante simples. A grande dificuldade é entender o que se pode paralelizar.

3.2 Funcionamento interno

De seguida será feita uma descrição do funcionamento interno do *runtime* Aeminium. O funcionamento é apresentado com recurso a diagramas UML (estados, sequências e classes) de modo a auxiliar a visualização das interacções internas do *runtime*.

3.2.1 Estados das tarefas

Durante o tempo de vida de uma tarefa existem 6 estados pelos quais esta pode transitar:

- UNSCHEDULED
- WAITING_FOR_DEPENDENCIES
- WAITING_IN_QUEUE
- RUNNING
- WAITING_FOR_CHILDREN
- COMPLETED

Através da ilustração seguinte são apresentados os estados existentes durante a vida de uma tarefa e condições para as transições ocorrerem.

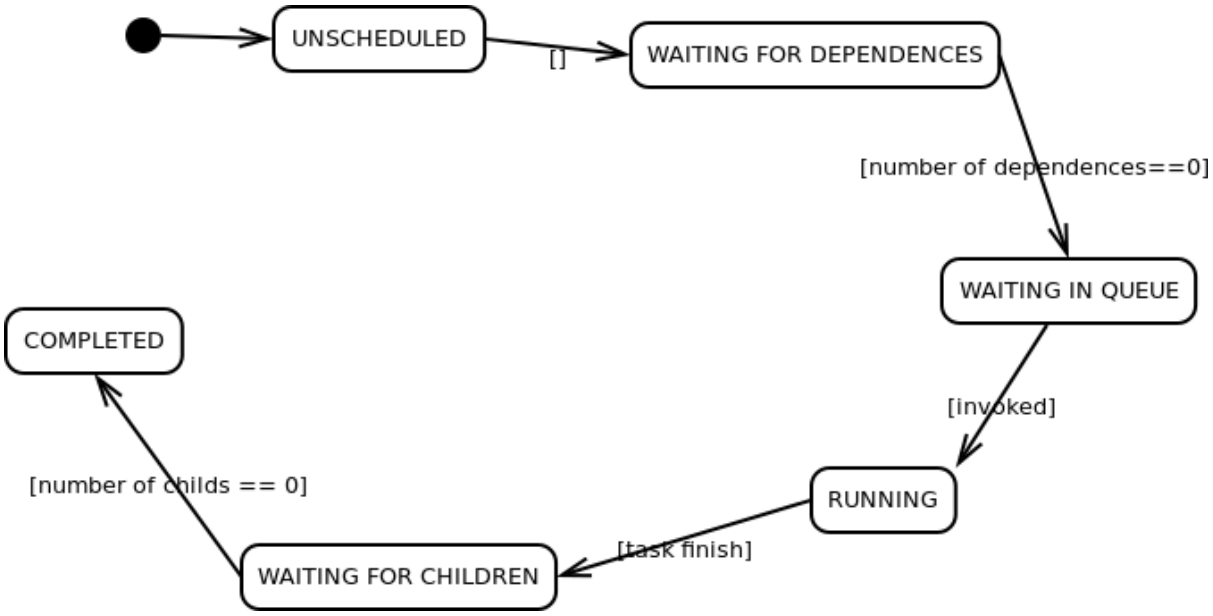

Fig. 6 – Estados das tarefas

O estado UNSCHEDULED é o estado inicial da tarefa ao entrar no grafo. Assim que esta dá entrada, transita para o estado WAITING_FOR_DEPENDENCIES e permanece neste estado até se verificar que as suas dependências terminaram de executar ou a tarefa não possui

dependências. Caso as condições anteriores se verifiquem, esta transita para o estado WAITING_IN_QUEUE, mantendo-se neste estado até que algum *worker* retire-a da *waiting queue* e inicie a sua execução, transitando imediatamente para o estado RUNNING. Logo que a tarefa termine de executar esta transita para o estado WAITING_FOR_CHILDREN e esperado até que todas as suas dependências terminem. Finalmente transita com o estado COMPLETED e dá-se finalizada a vida da tarefa.

3.2.2 Arquitectura do Aeminium Runtime

O *runtime* Aeminium (fig. 7) é constituído por três importantes módulos. O Grafo, o *Scheduler* e os *Workers* com as suas *waiting queues*.

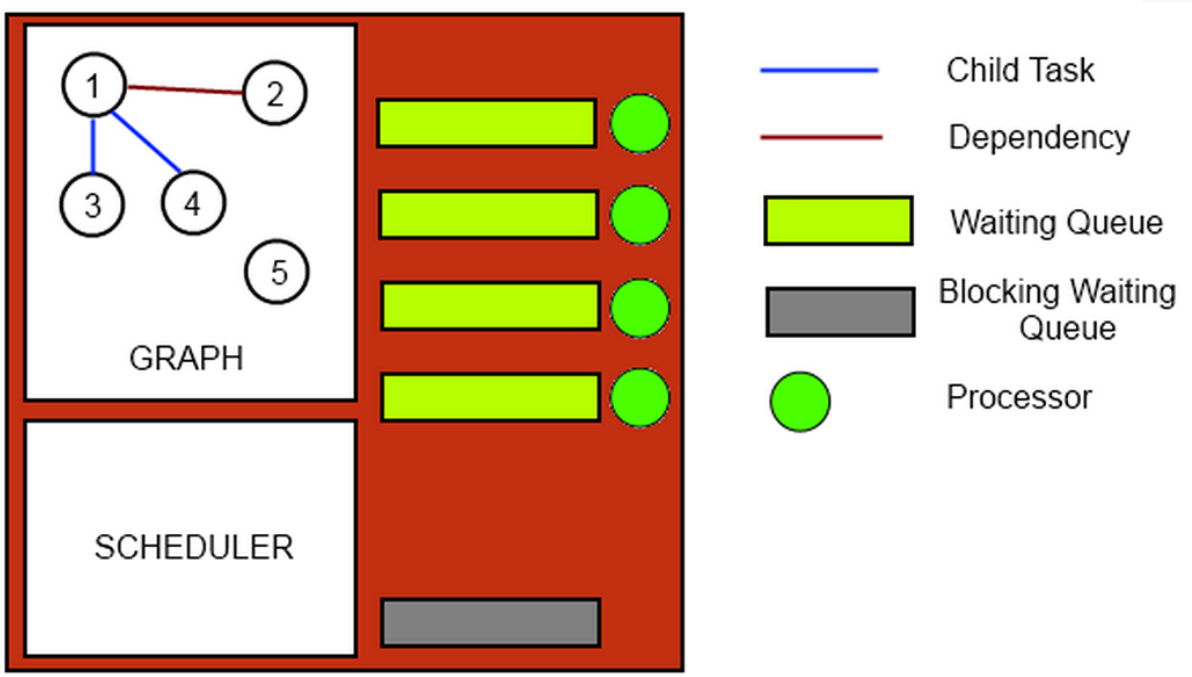

Fig. 7 - Arquitectura do Aeminium Runtime

Ao mesmo tempo que o código é executado, as tarefas são inseridas no Grafo e as dependências são verificadas. Caso, as tarefas sejam consideradas *ready* para executar, são enviadas para o *scheduler*, que tem como função inseri-las nas filas de execução de um *worker*. Relativamente aos *workers* é importante salientar que para cada um existe a respectiva WAITING QUEUE e caso não possuam tarefas nas suas filas de execução, este recorre ao *workstealing* para adquirir tarefas das filas dos *workers* vizinhos.

3.2.3 Sequência de eventos no escalonamento de tarefas

Para facilitar a compreensão da sequência de mensagens durante o tempo de vida de uma tarefa, recorreu-se um diagrama de sequências. Os diagramas com a sequência de mensagens

utilizada antes do *schedule* e após o *schedule* (fig. 24) (fig. 25) da tarefa são apresentados nos anexos deste documento. Antes do *schedule* da tarefa a troca de mensagens pode ser resumida aos seguintes passos:

1. Adição da tarefa ao *ImplicitGraph*;
2. Se a tarefa possuir pai, adiciona a tarefa à lista de filhos do pai;
3. Muda o estado da tarefa de UNSCHEDULE para WAITING_FOR_DEPENDENCES;
4. Caso a tarefa possua dependências, percorre cada uma das dependências e adiciona a tarefa à lista de dependentes. Ao mesmo tempo vai contando o número total de dependentes das suas dependências;
5. Se o número de dependentes for igual a zero, muda o estado da tarefa para WAITING_IN_QUEUE e faz o *schedule* da tarefa.

Após o *schedule* os passos principais são:

1. Caso a *thread* actual seja do tipo *WorkStealingThread*, a tarefa é adicionada à *taskQueue* da *WorkStealingThread*;
2. Caso seja a Main thread, a tarefa é adicionada à submisionQueue e é efectuado um *signalWork*() para acordar uma *WorkStealingThread* e executar as tarefas WAITING_IN_QUEUE;
3. A *WorkStealingThread* verifica se existe alguma tarefa na sua fila *taskQueue* e caso exista executa-a;
4. Caso não exista, tenta fazer *stealing* às outras filas de execução. Se encontrar uma tarefa,então executa-a, senão "adormece" a *thread*;
5. No final da execução de uma tarefa caso o número de filhos seja igual a zero, muda o estado da tarefa para COMPLETED, em caso negativo deixa o estado da tarefa em WAITING_FOR_CHILDRENS.

3.2.4 Classes fundamentais

O *runtime* Aeminium é constituído por mais de 50 classes, para facilitar a interpretação, retiraram-se as classes com menos importância para a optimização, gerando assim um diagrama de classes simplificado. A criação do referido diagrama foi efectuada com recurso ao *plugin* ObjectAId UML Explorer [28] para o Eclipse, através da utilização desta ferramenta é possível efectuar reverse *engineer* ao código existente e gerar vários tipos de

diagramas UML. O resultado obtido (fig. 26) pode visualizar-se nos anexos. De modo resumido as principais classes são as seguintes:

- *Runtime* (*ImplicitWorkStealingRuntime)*: É base do *runtime*, possui as funções para iniciar, parar, criar tarefas e fazer o *schedule*;
- Grafo (*ImplicitGraph)*: Onde se verificam as dependências;
- Tarefa (*ImplicitTask)*: A tarefa com os seus atributos;
- *Scheduler* (*BlockingWorkStealingScheduler)*: Onde se faz o escalonamento e distribuição de tarefas para as filas;
- *Worker* (*WorkStealingThread)*: Onde se executam as tarefas.

Capítulo
4 Validação do problema e identificação das questões de investigação

Neste capítulo é apresentada o *setup* experimental utilizado nas experiências, identificação e validação de possíveis hipóteses para investigação. Através da realização de experiências tentou-se identificar hipóteses que nos guiassem a políticas de optimização a implementar no *runtime* Aeminium. Preliminarmente realizaram-se experiências com diferenças no número e tamanho das tarefas na tentativa de perceber se estas características tem influência na performance do *runtime* e no final implementou-se um *profiler* para recolher dados durante a execução de vários programas, tentando assim encontrar padrões e anomalias na sua execução.

4.1 Setup experimental

Nesta secção é apresentado o setup experimental usado no decorrer das experiências. É apresentada as especificações de *hardware* da máquina Ingrid, uma introdução ao Jprofiler e o código utilizado para fazer medições do tempo de execução dos programas.

4.1.1 Hardware

Para executar as experiências foi usada a máquina Ingrid localizada no Departamento de Engenharia Informática da Universidade de Coimbra. As principais características são:

- PROCESSADOR: 2 processadores Intel X5660 2.8GHz (6 *cores* cada, com *hiper-threading*, formando um total de 24 *threads*);
- RAM: 24 GB com 50GB de espaço *swap*;
- DISCO: 2 x 1TB (7200 RPM);
- SISTEMA OPERATIVO: Ubuntu 12.10.

4.1.2 Ferramenta de estudo do comportamento - Jprofiler

Para a análise do comportamento das aplicações de teste em tempo de execução. Com esta, é possível visualizar dados tais como a memória RAM consumida em tempo real, *bottlenecks*, utilização de CPU, número de cores usados, *garbage collection*, a percentagem de tempo que as funções ocupam de processamento e outros aspetos importantes relativos ao comportamento do *runtime*. Para obter informação [19] sobre os métodos em execução, esta insere uma pequena sequência de instruções em *bytecode* no início e fim de cada método, de

modo a gravar o tempo despendido em cada um. Esta abordagem é conhecida como *bytecode injection*. Este tipo de ferramenta pode criar um "*observer effect*", só pelo simples facto de estar a monitorizar o comportamento do programa em execução, podendo desativar algumas otimizações, tais como *dead code elimination* e *inline method* que são efetuadas quando o programa está a correr sem *profiling*.

4.1.3 Medição do tempo de execução

Uma maneira bastante simples de medir o tempo de execução de um programa é usando o *system clock* e registar esse tempo num ficheiro. O Java possui o método System.currentTimeMillis() que retorna a diferença em milissegundos entre o tempo atual e a meia noite de 1 de Janeiro de 1970 UTC. Um exemplo do uso deste método é apresentado de seguida.

```
Long initialTime=System.currentTimeMillis();
//Parte do código a ser medida a performance...
Long finalTime=System.currentTimeMillis();
System.out.println("Time cost= "+(finalTime-initialTime)/1000);
```

Não se usou o System.*nanotime()*, uma vez que não era necessária uma definição tão alta em programas com tempo de execução superiores a vários segundos, como os utilizados no decorrer das experiências. Os dados obtidos, inicialmente eram retirados para sete execuções, sendo que aumentamos esse valor para trinta execuções, uma vez que só a partir deste valor se retira valores estatisticamente relevantes.

4.2 Número e dimensão das tarefas

Nesta secção são apresentados os resultados que se obtiveram ao utilizar programas com diferenças no número e tamanho das tarefas. Para tal decidiu-se recorrer a criação manual de programas básicos de paralelização de *loops*, baseando-se na técnica DOALL mencionada no livro "*Fundamentals of multicore Software development*" [14]. Este programa mais simples, permitiu-nos elevar o tamanho do ciclo para valores muito elevados. Ao código apresentado no livro adicionou-se um ciclo interno com uma operação de modo a podermos aumentar ou diminuir o tamanho de cada iteração e assim perceber se a dimensão das tarefas possui alguma influência. Além disso recorreu-se á execução do programa de multiplicação de matrizes com código gerado manualmente e outro com código gerado pelo compilador. A

grande diferença é que o código gerado pelo compilador cria muito mais tarefas e de menor dimensão que o mesmo programa com código gerado manualmente.

Na figura 8 são apresentados os tempos de execução médios obtidos em sete execuções para um ciclo externo com 12582912 iterações e um ciclo interno com 5000 iterações.

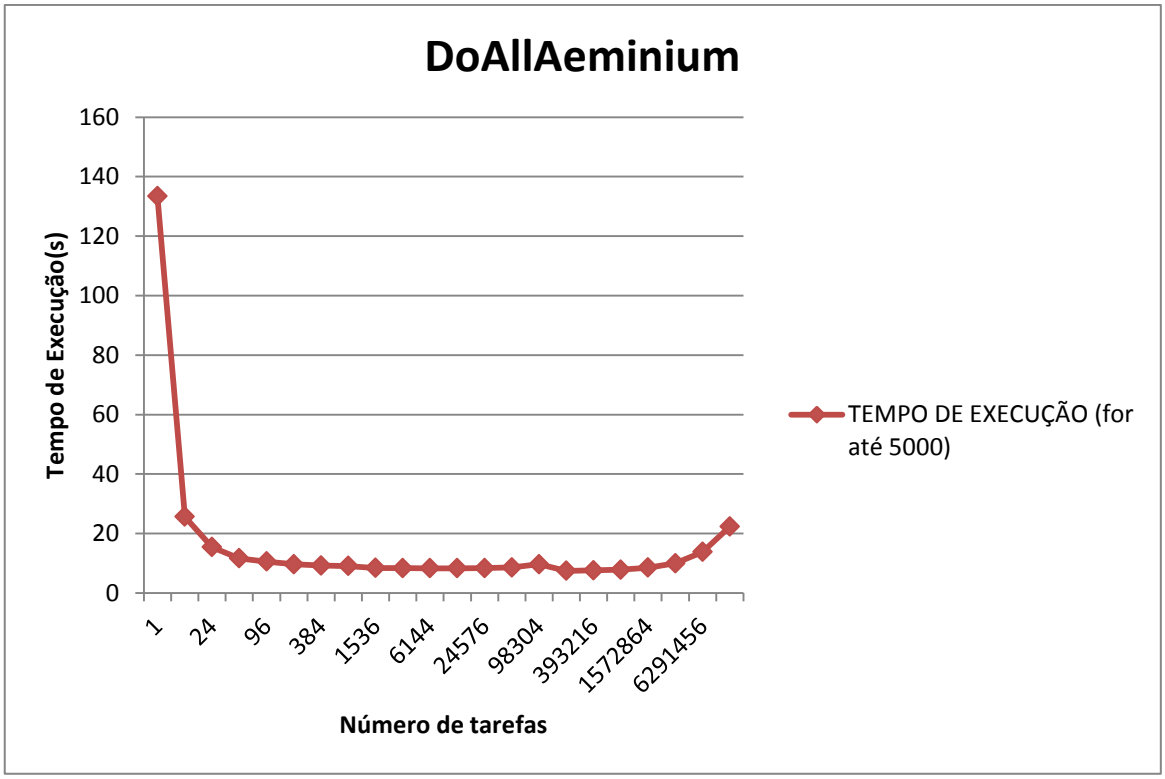

Fig. 8 - Tempo de execução DOALL com for até 5000

Como se constata na figura anterior a partir de um elevado número de tarefas a performance tem um decréscimo de performance evidenciado. Na figura 9 são apresentados os resultados obtidos a título de comparação para um ciclo interno de 5000,10000 e 15000 iterações respectivamente.

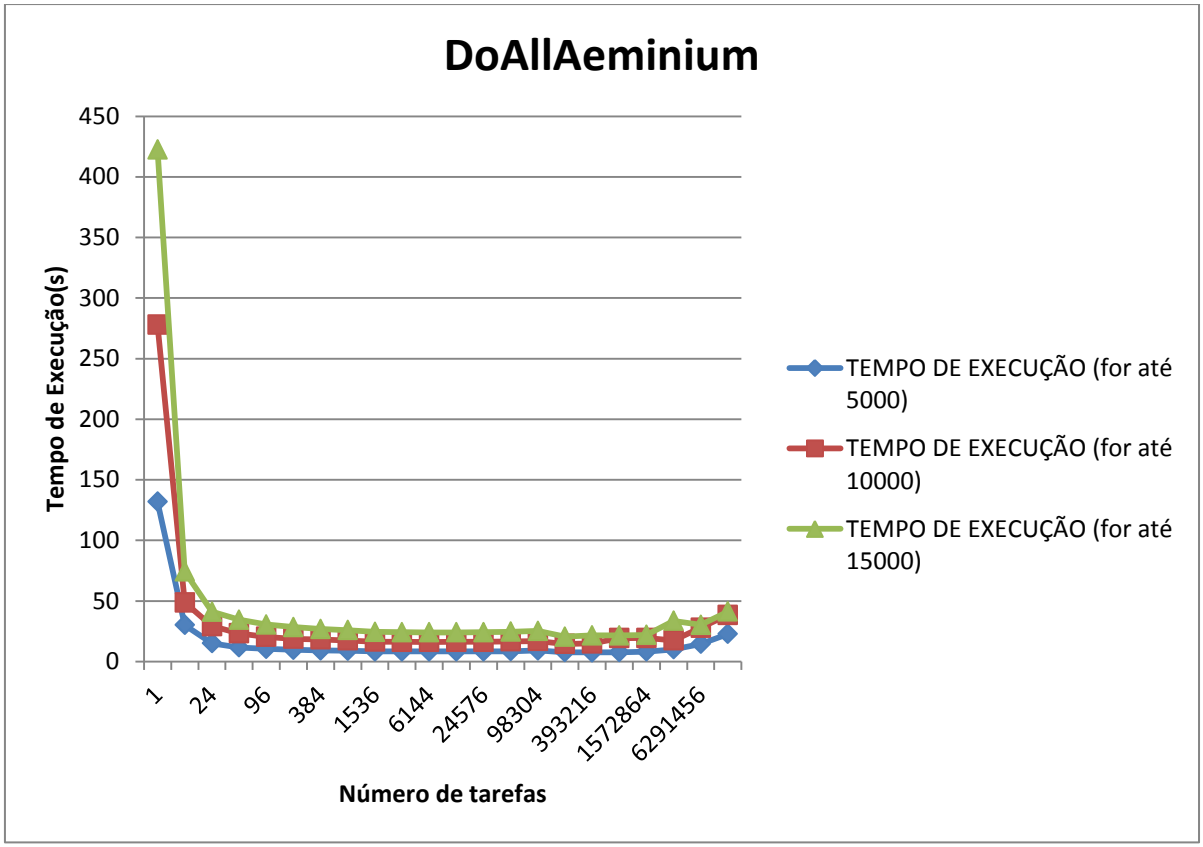

Fig. 9 - Tempo de execução DOALL com for até 5000, 10000 e 15000

Como se pode verificar a variação da dimensão da tarefa não infligiu nenhuma alteração ao comportamento do programa. No final foi efetuada uma experiência (fig. 10) com o programa *MatrixMultiplication*(multiplicação de matrizes) escrito de dois modos:

- Manualmente;
- Pelo compilador.

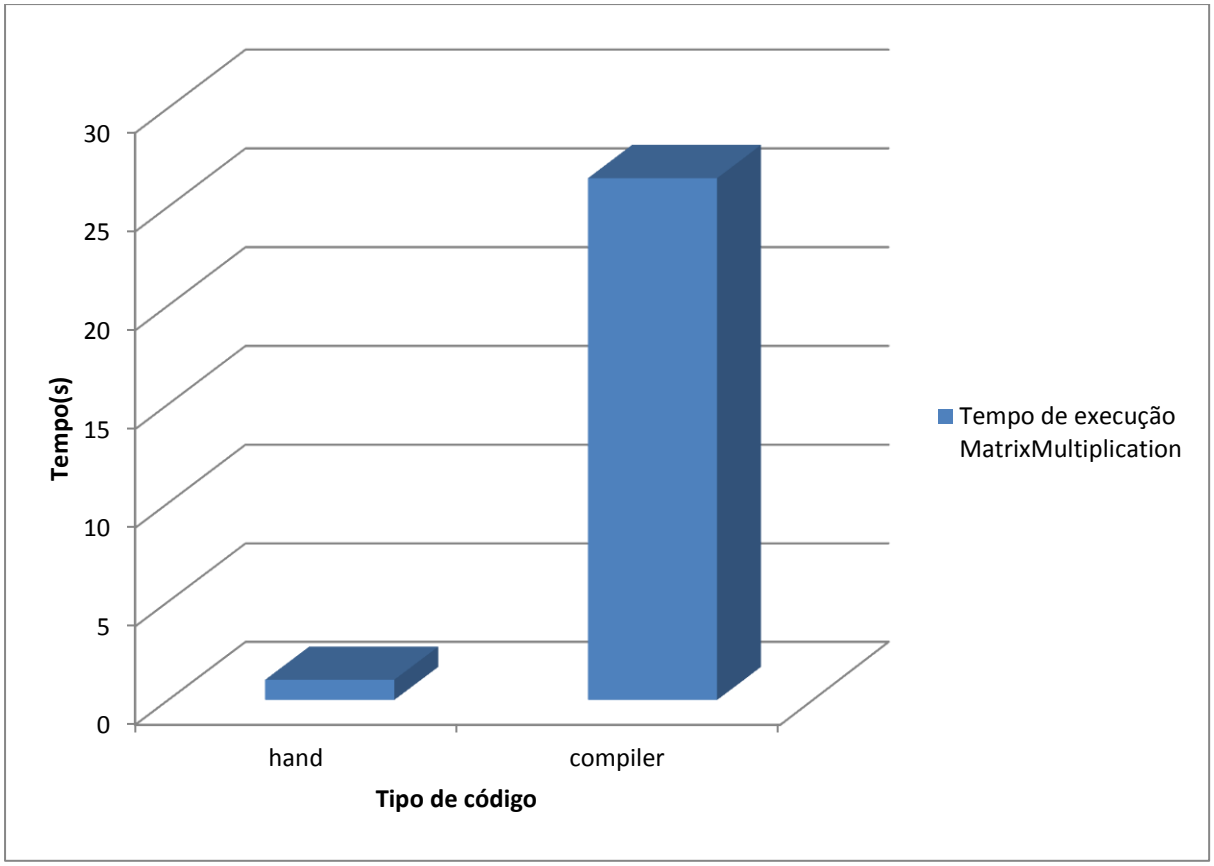

Fig. 10 - Código manual vs código do compilador

Através da análise do gráfico pode-se verificar que o código gerado manualmente possui um tempo de execução inferior, isto provavelmente associado às características do código. O código do compilador geralmente possui mais tarefas e dependências que o código escrito manualmente o que inflige um decréscimo de performance

4.3 Implementação de um profiler

Inicialmente pôs-se em dúvida se o número de *steals* não estaria demasiado elevado e se a inserção de tarefas nas filas de execução estava a ser efectuada de modo balanceado, isto é, a inserção de tarefas efectuada de modo a que o seu número em fila seja uniforme por todas as filas. Para verificar possíveis anomalias ou padrões de execução, foi implementado um *profiler* capaz de recolher diversas características durante a execução de programas e com estas encontrar possíveis optimizações. Na figura 29 em anexo é apresentado uma parte do output gerado pelo *profiler* implementado. A informação de profile é recolhida durante a execução dos programas e pode ser usada para encontrar padrões de execução, tipo de tarefas mais utilizadas, sequências de execução mais comuns, distribuição do trabalho pelos *cores*, etc. Com a implementação do profiler pretendia-se responder às seguintes perguntas:

- O número de steals pode estar demasiado elevado?

- Existe algum padrão de sequência na execução?
- Será benéfico pré-escalonar essa sequência?
- Existe algum desequilíbrio?

As características implementadas foram as seguintes:

- Total de tarefas executadas;
- Distribuição das tarefas pelos *cores*;
- Número de *steals* efetuados por cada *worker*;
- Sequência de execução das tarefas pelos *cores*;
- Sequência de execução do tipo de tarefas pelos *cores*;
- Total de tarefas por tipo;
- Número de dependentes por tipo de tarefa;
- Id dos dependentes por tipo de tarefa;
- Id do tipo de tarefa dos dependentes por tipo de tarefa;
- Número de tarefas por filas (*taskQueue*, *stealQueue*, *executed*);
- Total de tempo de espera das tarefas nas filas;

Para recolher dados do *profiler* foram executados programas dentro das seguintes categorias:

- Geometria – ClosestPair, Multiplicação de matrizes;
- Cálculo - PI, Fibonacci, fft;
- Otimização combinatória – Knapsack;
- Processamento de imagem - Histogram Equalization;
- Previsão – Blackscholes;
- Sequências – Mergesort;
- Strings – Longest Common subsequence (LCS);
- Grafos – BFS, Kdtree.

Seguidamente serão apresentados os resultados obtidos durante a execução do Knapsack para algumas da características que se efetuou *profiling*. Este programa foi escolhido para exposição do que ocorre, visto ser um dos que tem resultados mais significativos e representar a realidade de grande parte dos programas. O total de tarefas executadas por cada *worker* (fig. 11) numa execução escolhida ao acaso é apresentado no gráfico seguinte.

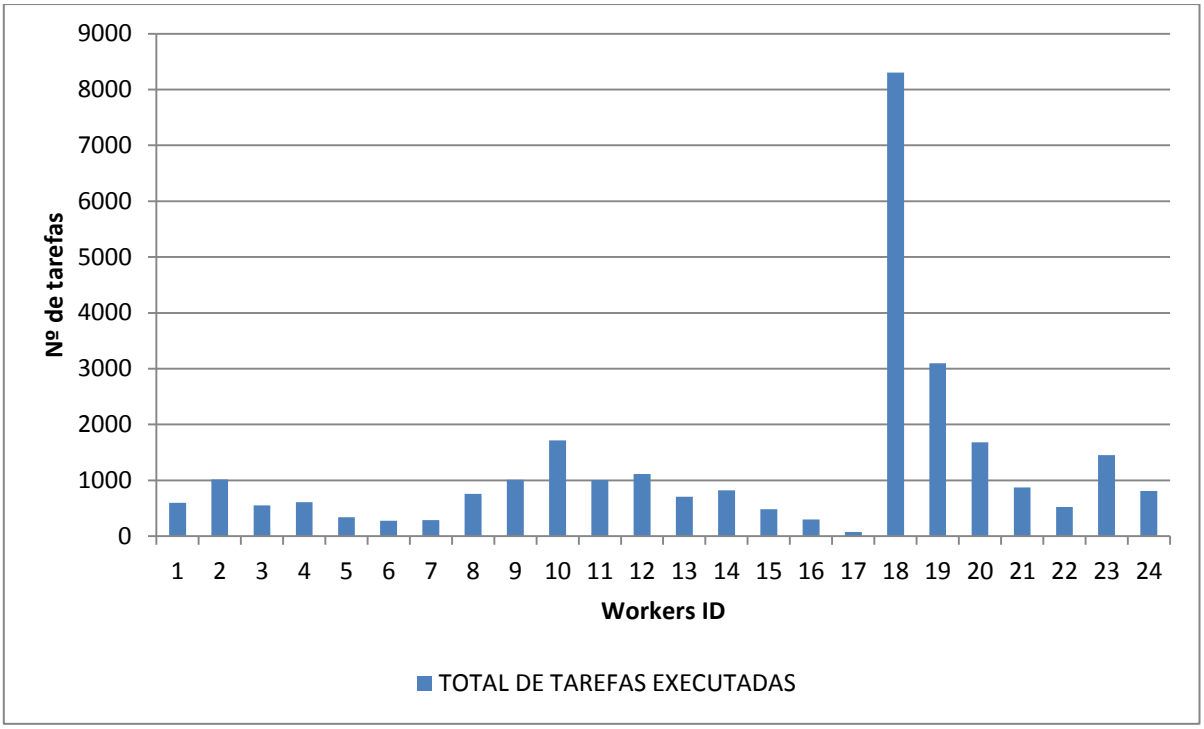

Fig. 11 - Total de tarefas executadas por worker

Como se pode verificar a distribuição de trabalhos não se encontra equilibrada pelos *workers*. Existindo alguns *workers* a executar menos de 100 tarefas, como é o caso do *worker* nº 16 e noutros casos como o do worker nº 18, a executar mais de 8000 tarefas. O número de *steals* executados por cada *worker* (fig. 12) é apresentado no gráfico seguinte.

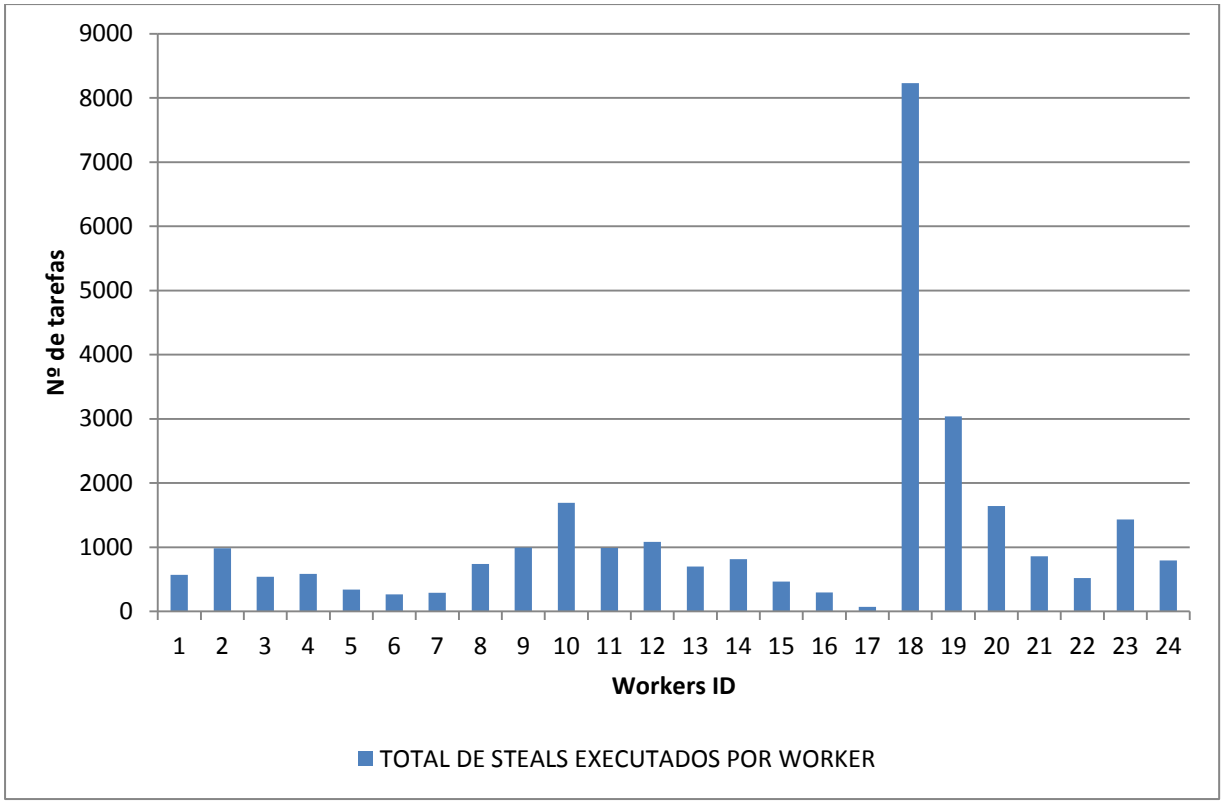

Fig. 12 - Total de steals executados por worker

Como se pode visualizar o número de *steals* dos *workers* nº 18 e nº 19 sobressaem comparativamente com os restantes *workers*. Tal como o ocorrido no total de tarefas executadas por worker, o número de *steals* continua ser desiquilibrado. O total de tarefas inseridas pelo *scheduler* em cada *worker* (fig. 13) continua a apresentar a tendência das experiências anteriores.

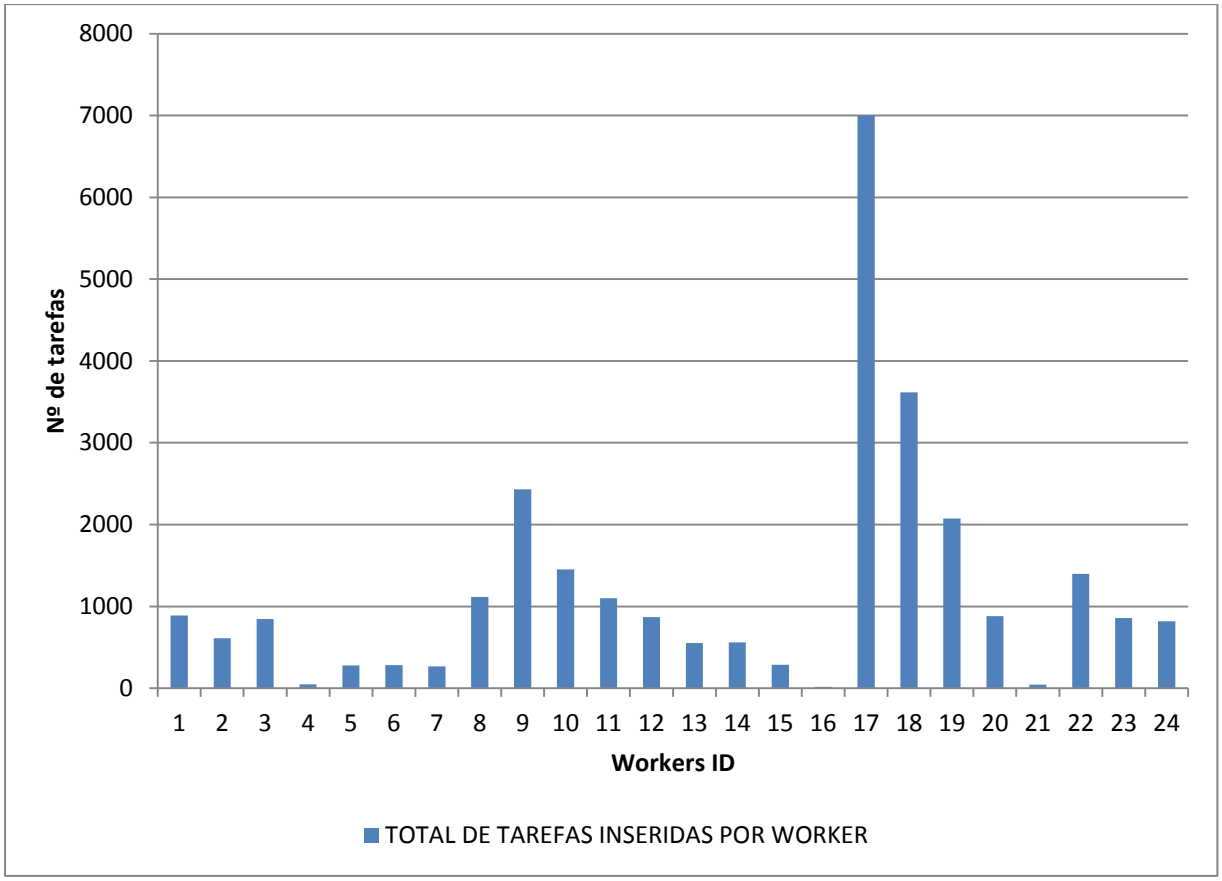

Fig. 13 - Total de tarefas inseridas por worker

Como se pode observar, existem alguns *workers* com cerca de 7000 tarefas inseridas pelo *scheduler* e outros com menos de 100 tarefas inseridas pelo *scheduler*.

Para finalizar, apresenta-se o total de tarefas pela sua origem (fig. 14). As tarefas podem provir de três locais, estes são:

- *taskQueue* - fila do próprio *worker*;
- *stealQueue* – fila de outro *worker*, tarefas adquiridas através de *steal*;
- *executed* – quando invocada a execução da tarefa sem que esta passe por uma fila pertencente a um *worker*. Esta situação pode ocorrer se a fila atingir o seu máximo.

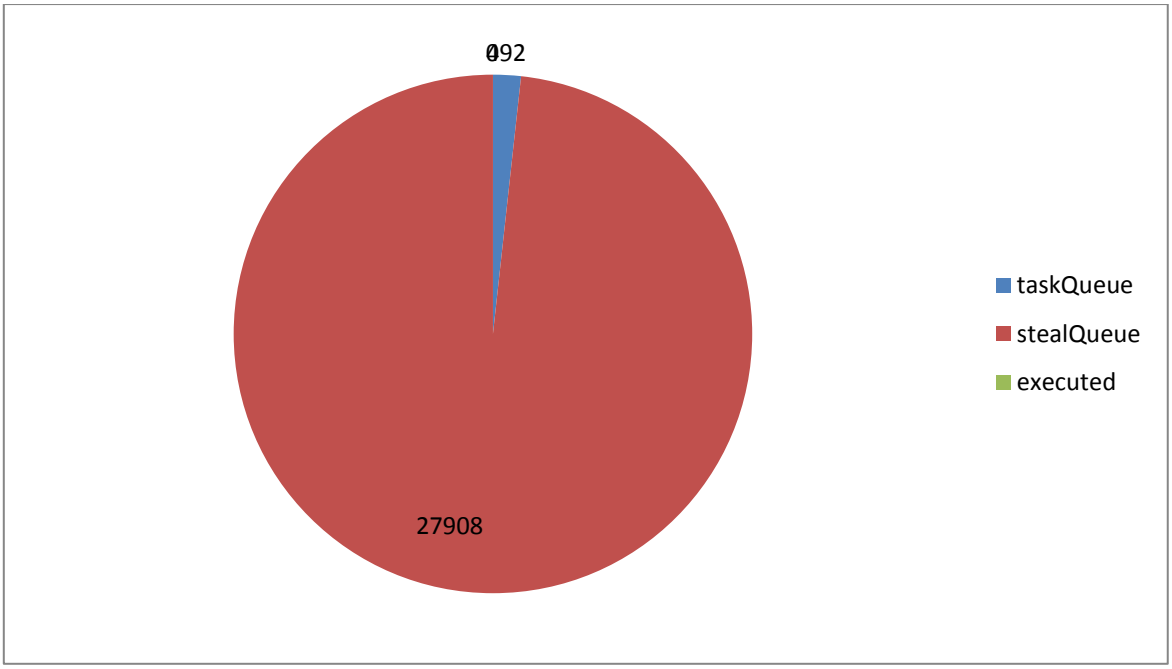

Fig. 14 - Total de tarefas por origem

Através da observação do gráfico anterior pode-se notar que a maioria das tarefas executadas provém da stealQueue. O número de tarefas provenientes da origem *executed* é nulo, que pode-se dizer ser normal.

Um ponto a notar é que o *worker* nº 17 é onde são mais inseridas tarefas - cerca de 7000, no entanto não executa nem 100 tarefas.

Com a implementação do *scheduler* conseguiu-se responder à primeira pergunta inicialmente formada. Mostrando que existia desequilíbrio na inserção de tarefas pelo *scheduler*.

4.4 Hipóteses a investigar

Através da realização das experiências anteriores foram definidas três hipóteses para investigação:

1. Um escalonamento mais equilibrado das tarefas em execução trará melhorias na performance;
2. Modificar o nível de paralelismo entre as tarefas em execução de acordo com a carga e capacidade do sistema trará melhorias de performance;
3. O controlo da granularidade das tarefas em execução e, consequentemente, do número de tarefas a executar trará melhorias de performance.

A primeira hipótese foi retirada da visualização do número de tarefas inseridas em cada *worker*. Uma vez que esta inserção não é balanceada, esta pode ser a causa do elevado número de *steals*, que podem causar um decréscimo na performance devido ao seu peso computacional. Como hipótese temos que, se efectuarmos um escalonamento mais equilibrado, podemos aumentar o *throughput*, diminuir os *steals* e melhorar a performance. Para isso, implementa-se uma optimização no *scheduler* que em vez de inserir tarefas nas filas de execução de modo *random*, irá basear-se no número de tarefas em fila para decidir em qual inserir a nova tarefa.

A segunda hipótese foi criada a partir da visualização da função que devolve a decisão de paralelizar ou sequencializar o código em execução. A decisão é baseada num *threshold* fixo relacionado com o tamanho da fila. Enquanto o tamanho da fila não atinge o *threshold* definido, esta devolve sempre a decisão de paralelizar e quando alcançado o *threshold* devolve sequencializar o código até ao final da execução do programa. Além desta visualização, acrescenta-se o fato de existirem desequilíbrios no total de tarefas executadas por *worker*. Este desequilíbrio pode levar a que o tamanho das filas dos *workers* que executam mais trabalho, alcancem valores elevados, originando a decisão de sequencializar mesmo que existam filas completamente vazias. Um excerto do código da função paralelize() é apresentado de seguida.

```
if ( ((WorkStealingThread)thread).getTaskQueue().size() >
parallelizeThreshold ) {
    return false;
} else {
    return true;
}
```

Uma hipótese a testar é se utilizar a informação do tamanho de todas a filas para auxiliar a decisão de paralelização pode otimizar a decisão e aumentar a performance.

Finalmente, a terceira hipótese foi retirada da observação dos resultados da primeira experiência. Através do uso de código escrito manualmente e código escrito pelo compilador nota-se que a grande diferença é o número de *Tasks* geradas pelo compilador comparado com o código escrito manualmente, sendo no primeiro caso muito mais elevado. Como se pode visualizar o código do compilador é mais lento comparado com o código manual. O número / dimensão das tarefas deve ser um ponto importante a investigar, pois para tarefas pequenas o *overhead* de criação e escalonamento de uma nova tarefa pode não compensar relativamente ao tempo que esta demora a executar. Uma possível implementação poderá ser a acoplação de

várias tarefas em uma só ou desagregação de uma tarefa em várias (fig. 15), fazendo assim variar o número e dimensão das tarefas.

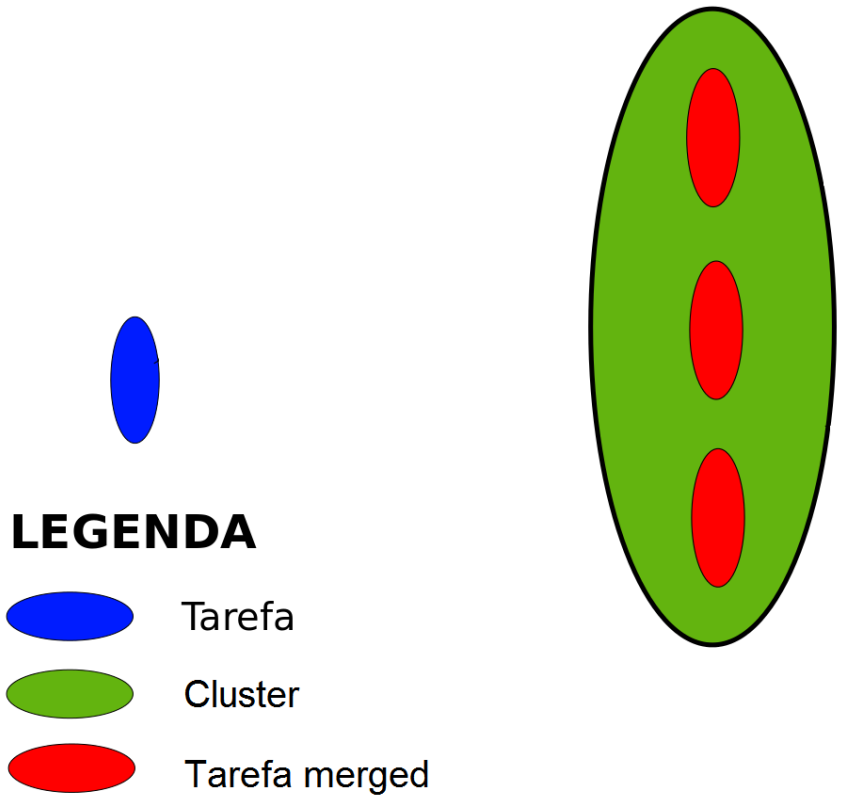

Fig. 15 - *Cluster* de tarefas

Em vez de se possuir várias tarefas, temos uma tarefa que engloba várias no seu interior. Esta solução poderá ajudar a diminuir o número de verificações e diminuir o *overhead* de escalonamento, sendo por isso uma hipótese a verificar se proporciona vantagens.

Capítulo
5 Otimização no escalonador

Neste capítulo é apresentada a primeira hipótese a investigar, abordagem utilizada e resultados obtidos.

5.1 Hipótese

Como primeira hipótese de investigação temos:

- Um escalonamento mais equilibrado das tarefas em execução trará melhorias na performance.

5.2 Abordagem

Visto detetar-se que a inserção de tarefas nas filas não era realizada de modo balanceado, implementou-se uma optimização no *scheduler* do *runtime* Aeminium. A optimização consistiu no varrimento das filas a inserir, na tentativa de encontrar uma fila vazia. Caso seja encontrada, adiciona-se a nova tarefa e envia-se um sinal para o *worker* iniciar imediatamente a sua execução. No caso de não se encontrar nenhuma fila vazia, a tarefa é inserida na fila do *worker* actual, tal como acontece na implementação original. Um exemplo do código inserido na função *scheduleTask*() da classe *BlockWorkStealingScheduler* é apresentado de seguida. Primeiro o código original e a seguir a optimização implementada.

```
/************* ORIGINAL ******************/

Thread thread = Thread.currentThread();

/************* OPTIMIZER *****************/
Thread thread = null;
int i;
for (i = 0; i < maxParallelism; i++) {
    if (numberOfTaskInWorkerQueue[i] < 1) {
        thread = threads[i];
        break;
    }
}
if (thread == null) {
    thread = Thread.currentThread();
}
/*****************************************/
```

Como se pode observar, a utilização da *thread* corrente é apenas para o caso de não se encontrar nenhuma fila vazia. Para elucidar, o objecto *thread* observado no código é o *worker* que contém a fila na qual se pretende inserir a tarefa. Para se recolher o número de tarefas em fila no momento da inserção, foi adicionado um contador a cada fila –

numberOfTasksInWorkerQueue[], que é actualizado sempre que se adicionam ou retiram tarefas. Para esta hipótese existiam várias abordagens possíveis, tais como:

- Encontrar a fila com menos tarefas para inserir a nova tarefa;
- Encontrar a fila com menor peso total de tarefas para inserir a nova tarefa.

Contudo, verificou-se que estas abordagens trariam demasiado *overhead*, uma vez que era necessário a cada inserção fazer o varrimento de todas as filas e assim encontrar a fila com menos tarefas ou menor peso em fila. Para clarificar, o peso total seria a soma dos pesos de todas as tarefas em fila. Para conhecer os pesos das tarefas, recorria-se aos *Hints* anteriormente expostos. Transferindo a informação relativa ao tamanho das tarefas do compilador para o *runtime*. Na política implementada, o varrimento total aconteceria apenas no pior dos casos - "Se todas as filas estiverem vazias, para quê procurar a que tem menos tarefas?", o que criaria menor peso computacional comparativamente com as duas abordagens sugeridas em cima.

5.3 Resultados

Nesta secção são apresentados os resultados obtidos para a validação da primeira hipótese proposta. Os resultados são apresentados em gráficos referentes aos tempos de execução e *speedup*. Foram utilizados os seguintes programas escritos manualmente [https://github.com/AEminium/AeminiumBenchmarks]:

- Knapsack: Resolve o problema de Knapsack através de um algoritmo genético;
- Mergesort: Programa para ordenar *arrays*;
- Blackscholes: Programa para previsões de ativos da bolsa;
- Doall: Programa com ciclo sem dependências entre as iterações;
- Histogram: Programa para dar contraste em imagens;
- Pi: Programa para cálculo do pi usando o método de montecarlo;
- Kdtree: Programa para organização de pontos no espaço;
- MatrixMultiplication(matrix): Programa para multiplicação de matrizes.

O tempo de execução médio de trinta execuções para os programas anteriores com e sem optimização (fig. 16) mostram que a optimização do *scheduler* obteve um efeito positivo na performance do *runtime*.

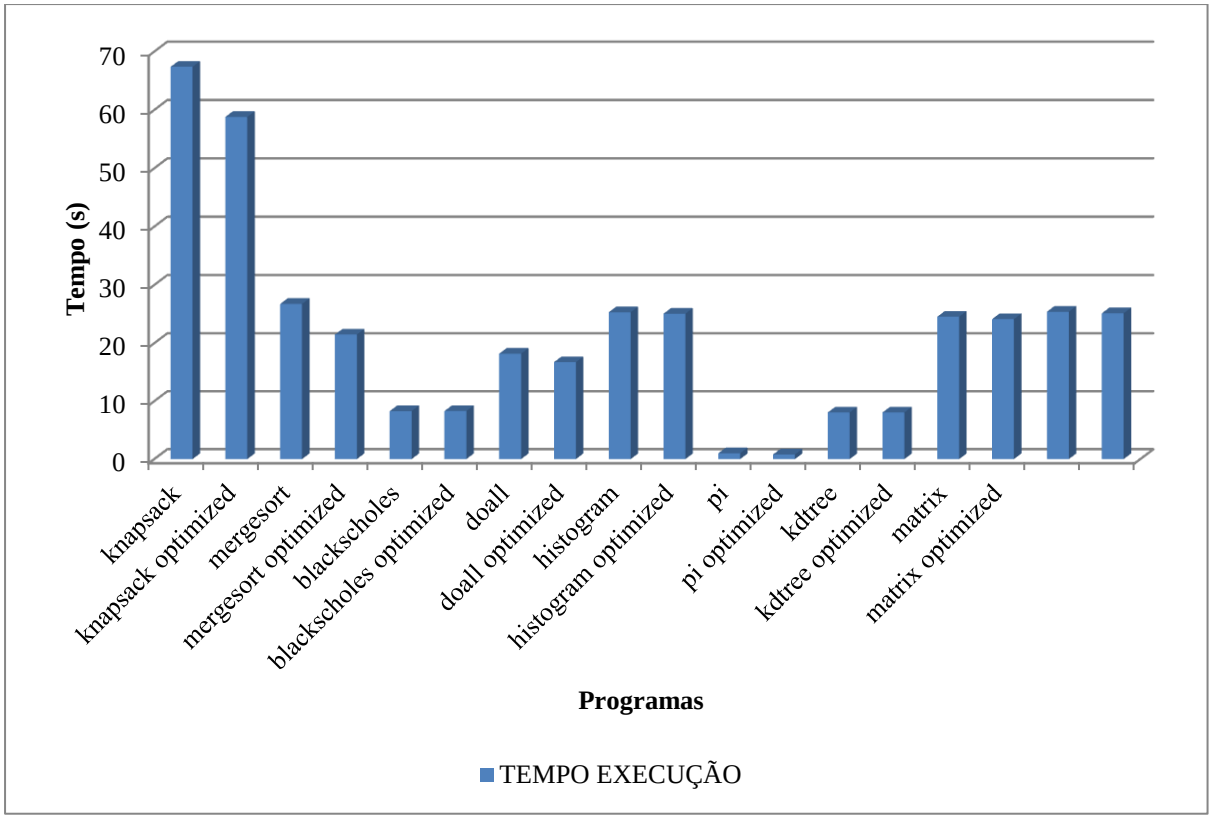

Fig. 16 - Tempo de execução com e sem otimização no scheduler

No eixo horizontal apresenta-se os programas em pares (original e optimizado) e no eixo vertical, o tempo de execução médio em segundos. O tempo de execução de alguns programas após a implementação da optimização, mostra resultados bastante interessantes, como é o caso do Knapsack, Mergesorte e pi com melhorias de performance entre os 13% (knapsack) e 20% (mergesort e pi). Nos restantes casos foi menos evidente ou quase nula, sendo que em nenhum dos casos a performance diminuiu. Através do gráfico do *speedup* (fig. 17) pode-se observar que tanto o mergesort como o pi obtiveram um *speedup* a rondar 1.25.

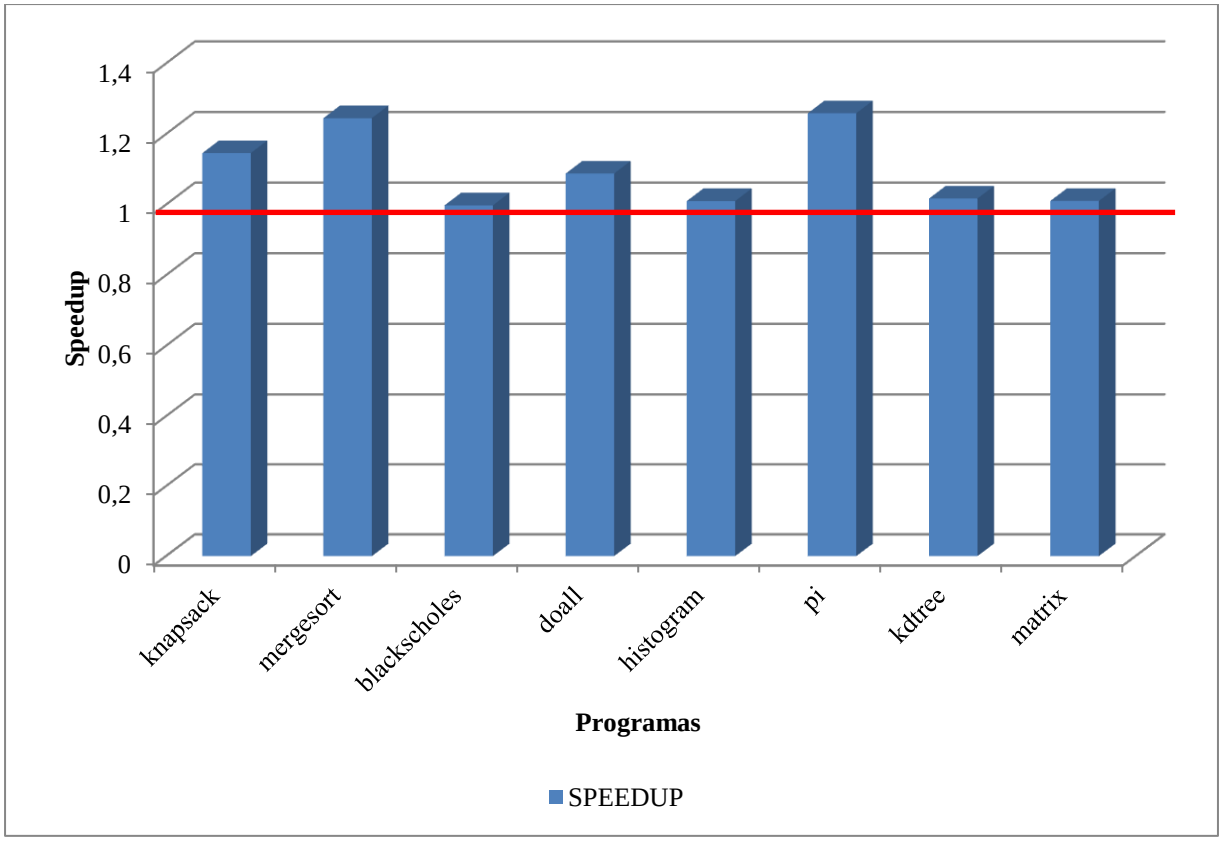

Fig. 17 - Speedup com e sem otimização no scheduler

No eixo horizontal são apresentados os programas experienciados e no eixo vertical o *speedup* obtido na optimização implementada em relação à versão sem optimização. Pode-se observar com detalhe que nenhum dos programas apresentou *speedup* inferior a um.

Capítulo
6 Otimização da decisão de paralelizar

Neste capítulo é apresentada a segunda hipótese a investigar, abordagem utilizada e resultados obtidos.

6.1 Hipótese

Como segunda hipótese de investigação temos:

- Modificar o nível de paralelismo entre as tarefas em execução de acordo com a carga e capacidade do sistema trará melhorias de performance.

6.2 Abordagem

A função paralelize() tem a seu cargo informar se o *runtime* deve paralelizar ou sequencializar o código. Clarificando, esta função tem como função decidir se é utilizado o código sequencial ou o código paralelo. O exemplo da utilização desta função no programa kdtree é exposto de seguida.

```
public void createSubTrees(final Point[] points, final int depth, Runtime
rt, Task parent) {
    int median = (points.length - 1) / 2;
    if (rt.parallelize())
        createSubTreesInPar(points, depth, rt, parent, median);
    else
        createSubTreesInSeq(points, depth, rt, parent, median);
}
```

Como se pode observar, caso a função devolva *true,* é chamado o código paralelo (*createSubTreesInPar*()), senão é chamado o seu código sequencial (*createSubTreesInSeq*()). Durante as primeiras experiências foi posto em dúvida se esta decisão não estaria demasiado simplista. Esta função baseava-se somente no tamanho de uma fila para tomar a decisão, o que em certos casos poderia levar a que esse *threshold* fosse atingido, mesmo existissem várias filas sem qualquer trabalho. Esta abordagem simplista poderia levar na maioria das vezes a opções inválidas de sequencialização. Como tentativa de resolução deste problema propôs-se as seguintes implementações:

1. Adicionar a verificação de todas as filas estarem ocupadas como condição para se sequencializar;
2. Adicionar a verificação de metade das filas estarem ocupadas como condição para se sequencializar;

3. Retirar o *threshold* e utilizar apenas a verificação de todas as filas estarem ocupadas como condição para se sequencializar.

Um exemplo da alteração efetuada na função paralelize() para o caso da segunda implementação é apresentado de seguida.

```
if (((WorkStealingThread) thread).getTaskQueue().size() >
parallelizeThreshold
        && scheduler.checkParallelize() == false) {
    return false;
} else {
    return true;
}
```

Como se pode visualizar foi adicionada a função checkParallelize() que retorna um *boolean*. Esta função retorna *false* caso existam menos de metade das filas vazias, de outro modo retorna *true*.

6.3 Resultados

Os tempos de execução médios obtidos de trinta execuções nas quatro implementações para verificar a segunda hipótese são apresentadas para os programas mergesort (fig. 18) e kdtree (fig. 19). Foram escolhidos para a validação desta hipótese os programas citados anteriormente, visto que do conjunto de programas existentes no benchmark [https://github.com/AEminium/AeminiumBenchmarks], eram os que recorriam à utilização da função paralelize() ou facilmente se modificavam para recorrer a essa utilização. Nas implementações com threshold foi utilizado o valor 3 (configuração default do runtime), isto quer dizer que só ocorre a sequencialização se o tamanho da fila for superior a este valor. De seguida são apresentados os resultados para o programa mergesort.

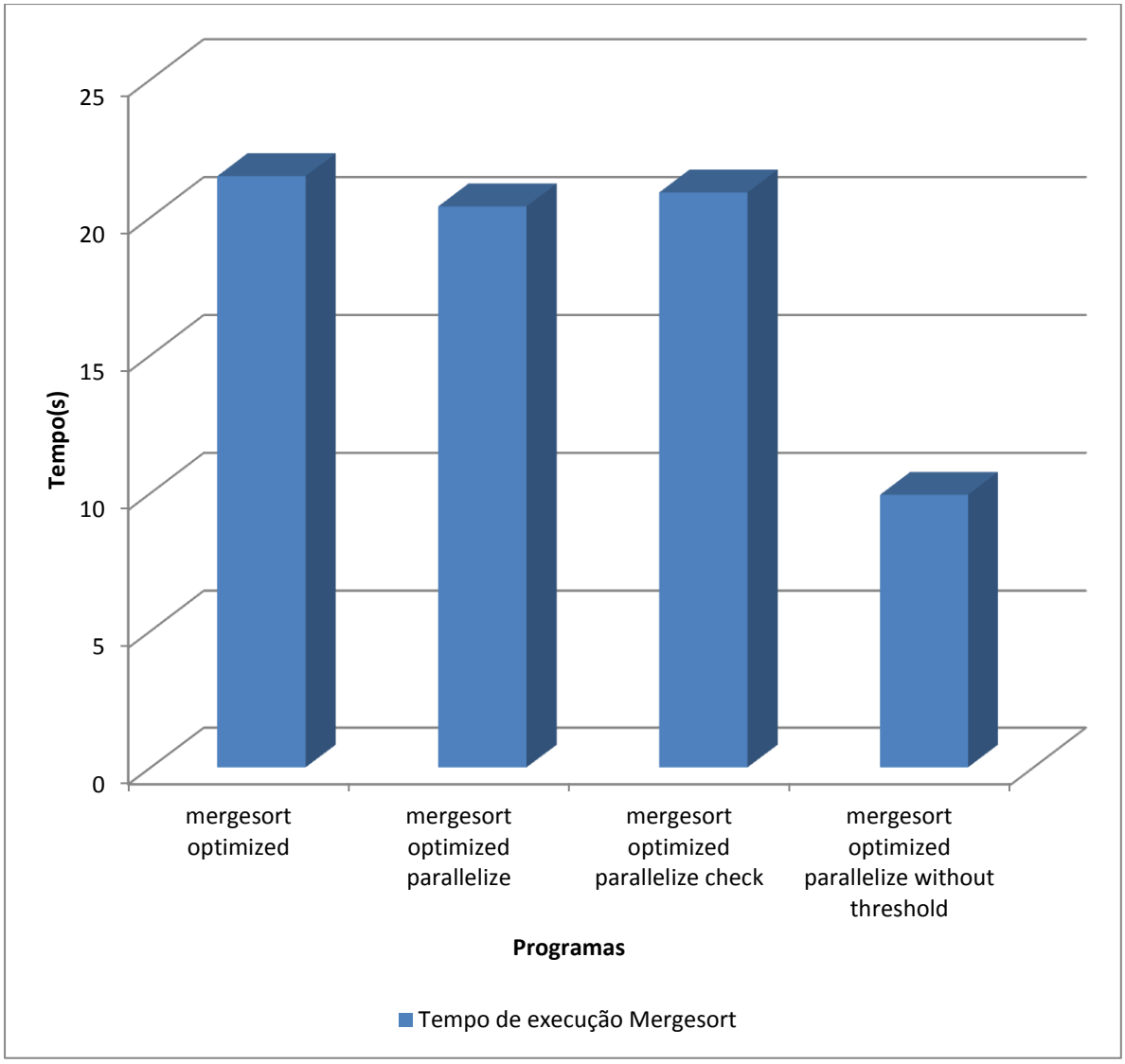

Fig. 18 - Mergesort com várias decisões de paralelizar

Como se pode observar existem quatro barras, cada uma referente a uma implementação (eixo horizontal):

- *mergesort otimized*: versão do *runtime* com a optimização do *scheduler*;
- *mergesort otimized paralelize*: função paralelize com verificação se todas as filas estão ocupadas e *threshold* ==3;
- *mergesort otimized paralelize check*: função paralelize com verificação se pelo menos metade das filas estão ocupadas e *threshold* ==3;
- *mergesort otimized paralelize without threshold*: função paralelize com verificação se todas as filas estão ocupadas e remoção do *threshold* das condições para paralelizar;

No eixo vertical é apresentado o tempo de execução médio em segundos. No mergesort todas as implementações obtiveram melhores resultados que a versão com o paralelize() implementada actualmente no *runtime*. Tendo melhorias de performance entre 3% a 55%. Seguidamente são apresentados os resultados para o programa kdtree.

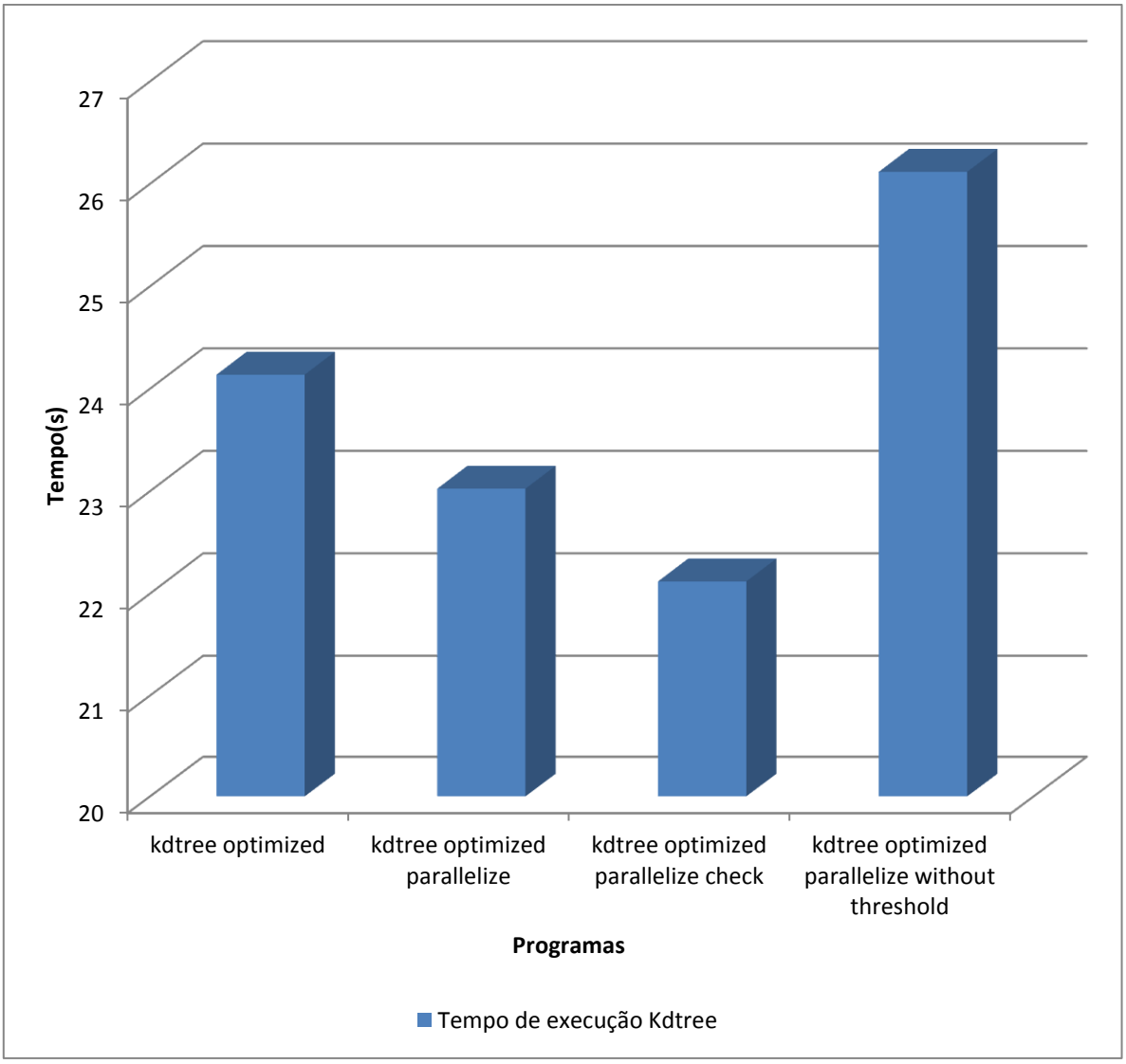

Fig. 19 - Kdtree com várias decisões de paralelizar

Novamente, são apresentadas as implementações experienciadas no eixo horizontal e o tempo de execução médio no eixo vertical. No kdtree pode-se verificar, que ao contrário do mergesort, a versão sem *threshold* é a que obtém piores resultados. No entanto, as outras duas implementações continuam a obter melhor *performance* que a implementação original, conseguindo melhorias até 9%. A implementação do *threshold* no kdtree poderá estar a influenciar a performance, na medida que protege a decisão de sequencializar não só quando este possui todas as filas ocupadas, mas também com um tamanho superior a três. A

sequencialização antecipada poderá levar à falta de tarefas, originando que alguns *workers* fiquem sem trabalho, diminuindo a paralelização e posteriormente a *performance*.

Capítulo
7 Módulo para efetuar agregação de tarefas

Neste capítulo é apresentada a terceira hipótese a investigar, abordagem utilizada e resultados obtidos.

7.1 Hipótese

Como terceira hipótese de investigação temos:

- O controlo da granularidade das tarefas em execução e, consequentemente, do número de tarefas a executar trará melhorias de performance.

7.2 Abordagem

O número e dimensão das tarefas poderão gozar de alguma influência na performance do *runtime* Aeminium. Como hipótese a este problema podemos fazer *merge* e *unmerge* de tarefas, diminuindo o número de tarefas escalonadas e por consequente o número de verificações. A arquitectura pensada para o módulo de efectuar *merge* e *unmerge* de tarefas é apresentada na figura 20.

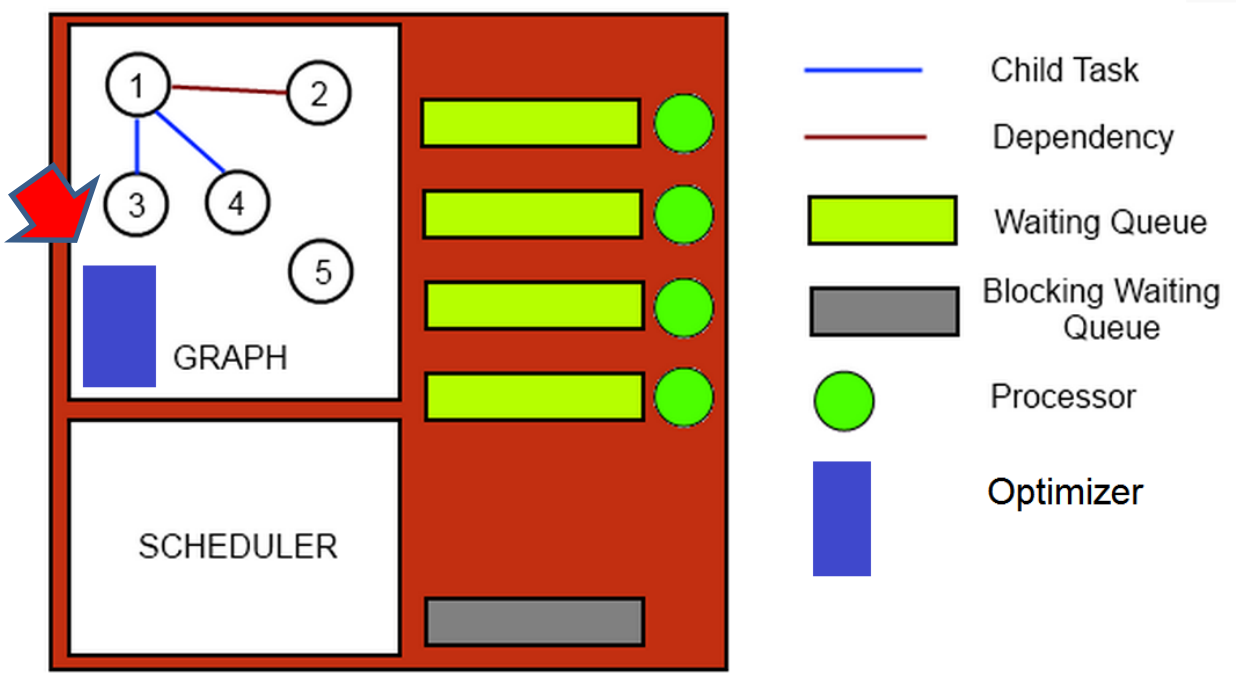

Fig. 20 - Arquitetura do runtime com otimizador

Como se pode observar, o módulo de optimização (*Optimizer*) foi adicionado ao Grafo. Inicialmente pensou-se fazer o *merge* nas filas de execução. Clarificando, a tarefa já se encontraria na fila de execução à espera de ser executada e caso verificasse certas condições

era efetuado o *merge* com uma nova tarefa. Em anexo (fig. 27) (fig. 28) é possível visualizar o diagrama de sequências desta possível implementação. Depois de analisada, esta implementação provou ter possíveis defeitos, tal como:

- Bloquear a execução de um *worker*: Durante o *merge* seria necessário efectuar *lock* à tarefa e caso fosse a única tarefa em fila o *worker* teria que aguardar que o *merge* finalizasse.

O funcionamento do módulo implementado consistiu em atrasar o escalonamento de uma tarefa se esta não possuísse tarefas filhas (informação recebida através dos *Hints*). Esta condição é necessária já que as tarefas filhas são lançadas em execução e perderíamos o controlo das dependências, caso efectuássemos um *merge* de duas tarefas que fossem dependentes uma da outra. Seguidamente, uma nova tarefa poderia efectuar o *merge* com a tarefa em espera caso existissem as seguintes condições:

- A tarefa em espera é a dependência da nova tarefa e possui apenas um dependente que é a nova tarefa;
- A nova tarefa não tem dependências.

Para implementar o *merge* das tarefas poderíamos recorrer às seguintes abordagens:

- *Lazy tasking merging*: ao adicionar uma tarefa ao *runtime*, verifica-se se existe possibilidade de a adicionar a uma já existente - cria-se um novo body, que é o conjunto dos *bodies* das duas e substitui-se o *body* da que se encontra à espera de executar por este novo *body*;
- *Body appending*: permitir adicionar mais *bodies* às tarefas já existentes, ter assim um *array* de *bodies* na tarefa.

Para esta implementação foi escolhida a primeira opção- *Lazy tasking merging*, pois não era necessário modificar a estrutura da *Task*. O exemplo da função *merge* de tarefas sem dependências é apresentado de seguida.

```
private boolean mergeTaskNoDependences(ImplicitTask taskParent,
        ImplicitTask itask) {

        Body bodyMerged = mergeBodies(taskParent.body, itask.body);
        itask.clusterTask = taskParent;
        itask.taskFinishedMerge(rt);

        taskParent.body = bodyMerged;
        taskParent.addMergedTask(itask);

        taskParent.updateTaskSize(itask);

        mergeTotal++;
        return true;
}
```

Como se pode observar, foi implementada uma função *mergeBodies*, que recebe como parâmetros os *bodies* das duas tarefas a efectuar *merge* e retorna um novo *body* que é a acoplação dos dois *bodies* fornecidos. Após o *merge* informa-se o *runtime* que a *task* acoplada terminou e adiciona-se essa *task* à lista de *tasks merged* no *cluster*. No final é efetuado um *update* ao tamanho do *cluster* através da função *updateTaskSize()*.

Após o *merge*, a tarefa *cluster* aguarda o seu escalonamento até que o seu tamanho alcance o valor estipulado (valor mínimo para efetuar *merge*). Esta situação pode ser visualizada no código seguinte.

```
if (mergeTaskNoDependences(tempTask, itask) == true) {
    if (tempTask.getTaskSize() > MERGE_NUMBER) {
        rt.scheduler.scheduleTask(tempTask);
    } else {
        tasksReadyToSchedule.add(tempTask);
    }
    return;
}
```

No caso apresentado a variável MERGE_NUMBER é o valor mínimo para efectuar *merge* e a lista *tasksReadyToSchedule* é onde a tarefa aguarda pelo seu escalonamento.

7.3 Resultados

Para efectuar a validação do módulo para efectuar *merge* de tarefas foi utilizado código gerado tanto de modo manual como pelo compilador. Foram utilizados os seguintes programas escritos de modo manual:

- Blackscholes;

- AeForPi (pi);
- MatrixMultiplication;
- Histogram;
- Knapsack.

Pelo compilador J2JPar:

- Doall.

A utilização dos programas existentes no *benchmark* foi limitada aos programas citados anteriormente, uma vez que estes eram os únicos que verificavam as condições para efectuar *merge* de tarefas. Foram feitas experiências com os programas escritos manualmente (fig. 21), retirando-se o tempo médio de trinta execuções, variando o valor mínimo para efectuar *merge* entre os valores 0 e 48.

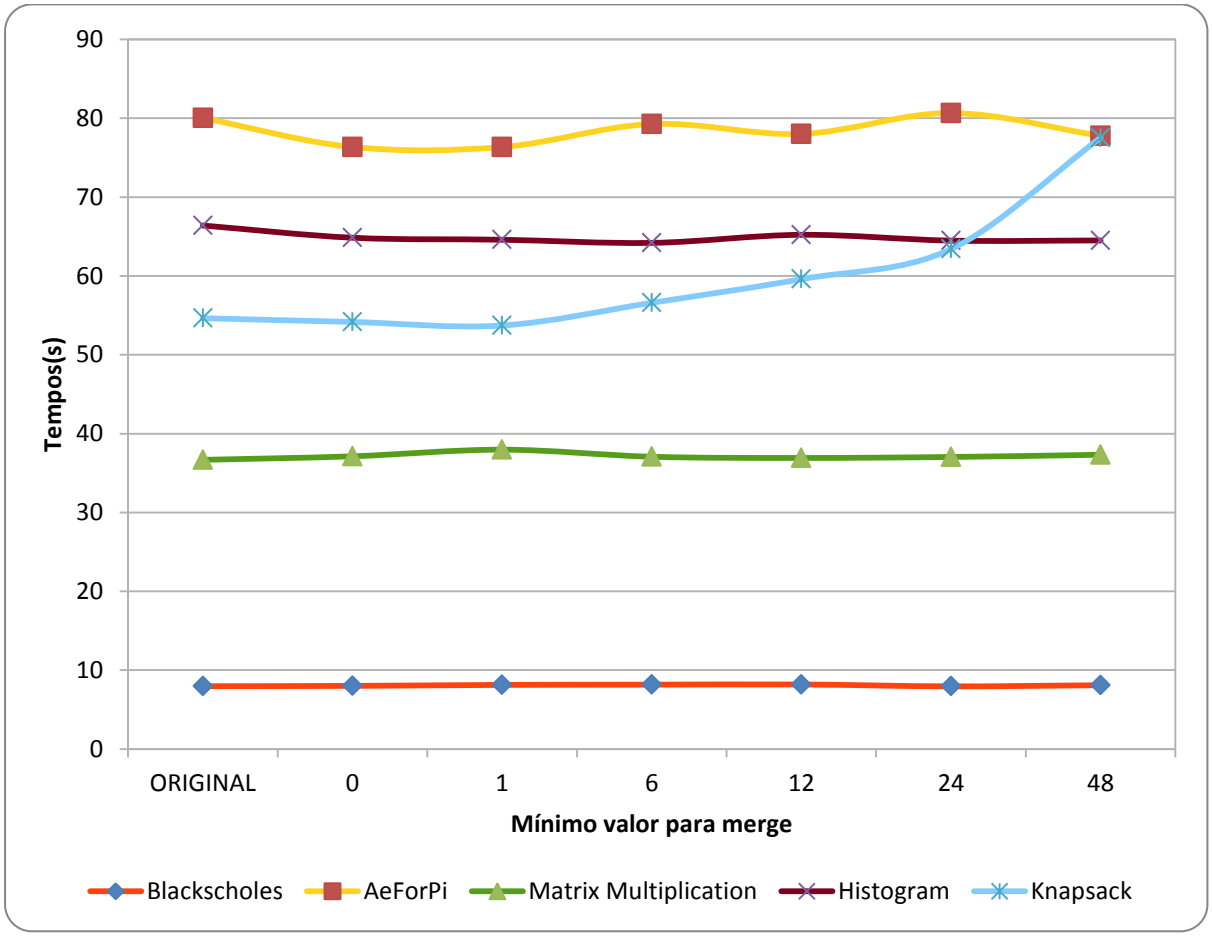

Fig. 21 - Variação do valor mínimo para efetuar merge em código manual

Com se observa, a variação do valor mínimo para efectuar *merge* não inflige qualquer modificação relevante na performance, sendo que para valores elevados a tendência é degradar-se, como é o caso do programa Knapsack. Por sua vez, as experiências com um programa gerado pelo compilador (fig. 22) apresentaram resultados positivos.

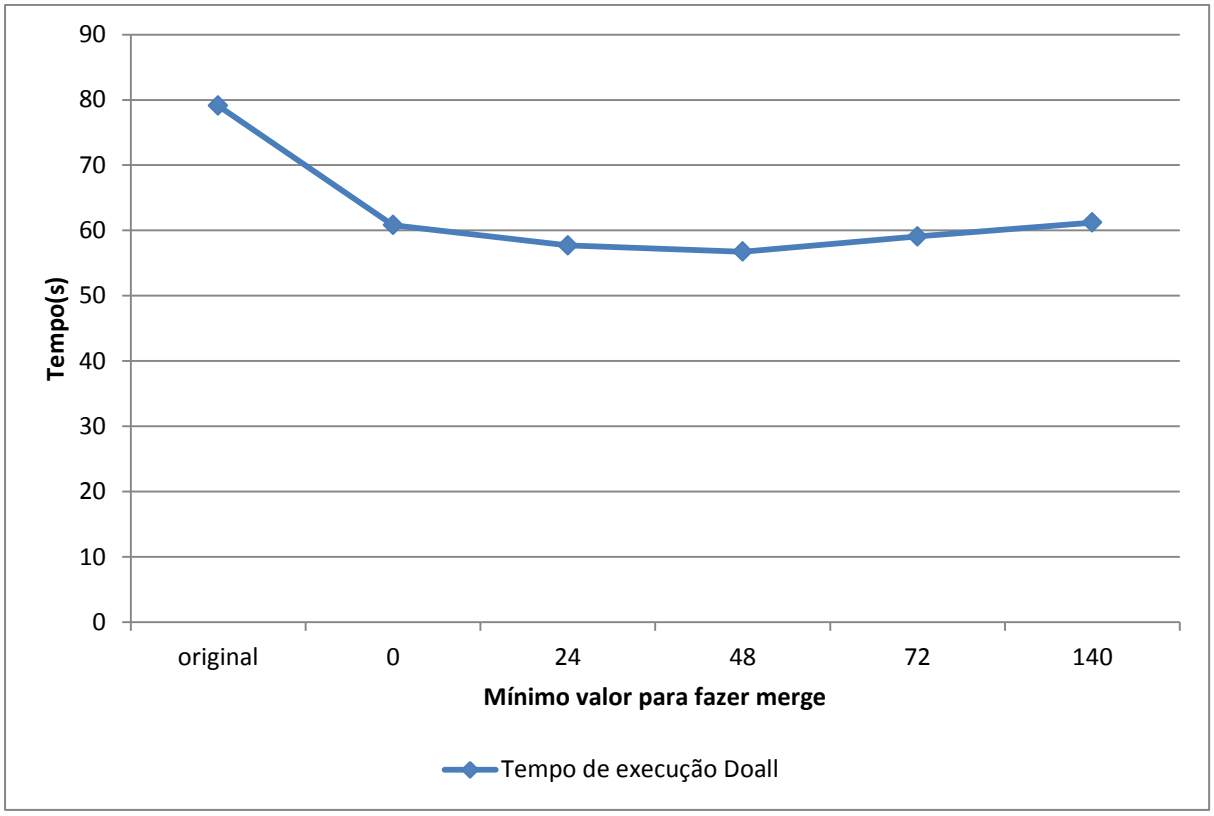

Fig. 22 - Variação do valor mínimo para efetuar merge em código gerado pelo compilador

Como se pode observar, existe um aumento de performance até *clusters* com 24 a 48 tarefas, sendo que a partir desse valor a performance começa a diminuir. O módulo de agregação de tarefas mostrou-se vantajoso para o programa Doall, conseguindo melhorias na ordem dos 30%, com desvio padrão entre 1.79 (valor min 0) e 13.77 (valor min 140). As razões para esta melhoria poderão ser o resultado da diminuição do número de tarefas e por consequência o número de escalonamentos.

Capítulo
8 Plano de trabalhos e implicações

Neste capítulo é descrito o planeamento para a elaboração deste estágio através de um diagrama de *Gantt*, uma breve descrição do funcionamento do estágio e a apresentação da equipa Aeminium Coimbra.

8.1 Planeamento e metodologias

A metodologia usada para o desenvolvimento do projeto foi baseado em princípios dos métodos ágeis. Marcando-se reuniões semanais onde se analisa o estado das metas, reestruturação das metas e delineação de novas metas. Para a gestão do projeto foi usado o Redmine, onde se marcavam as novas tarefas, *deadlines* e estado de progresso das mesmas.

8.1.1 Planeamento anual

Durante o primeiro semestre houve uma maior incidência no estudo relativo ao estado da arte do tema a investigar, ou seja, da programação paralela e otimizações ao nível do *runtime*, através de uma consulta bibliográfica intensiva. Ao mesmo tempo, foi um período dedicado a uma ambientação/adaptação da *framework* Aeminium, onde foi possível a realização de várias experiências na tentativa de identificar possíveis questões relacionadas com a performance do *runtime* Aeminium. O segundo semestre foi dedicado à implementação das hipóteses de resolução das questões geradas e validação com programas reais. A figura seguinte apresenta o planeamento inicialmente previsto. Sendo na realidade, adicionada mais uma hipótese de optimização à planificação inicialmente proposta.

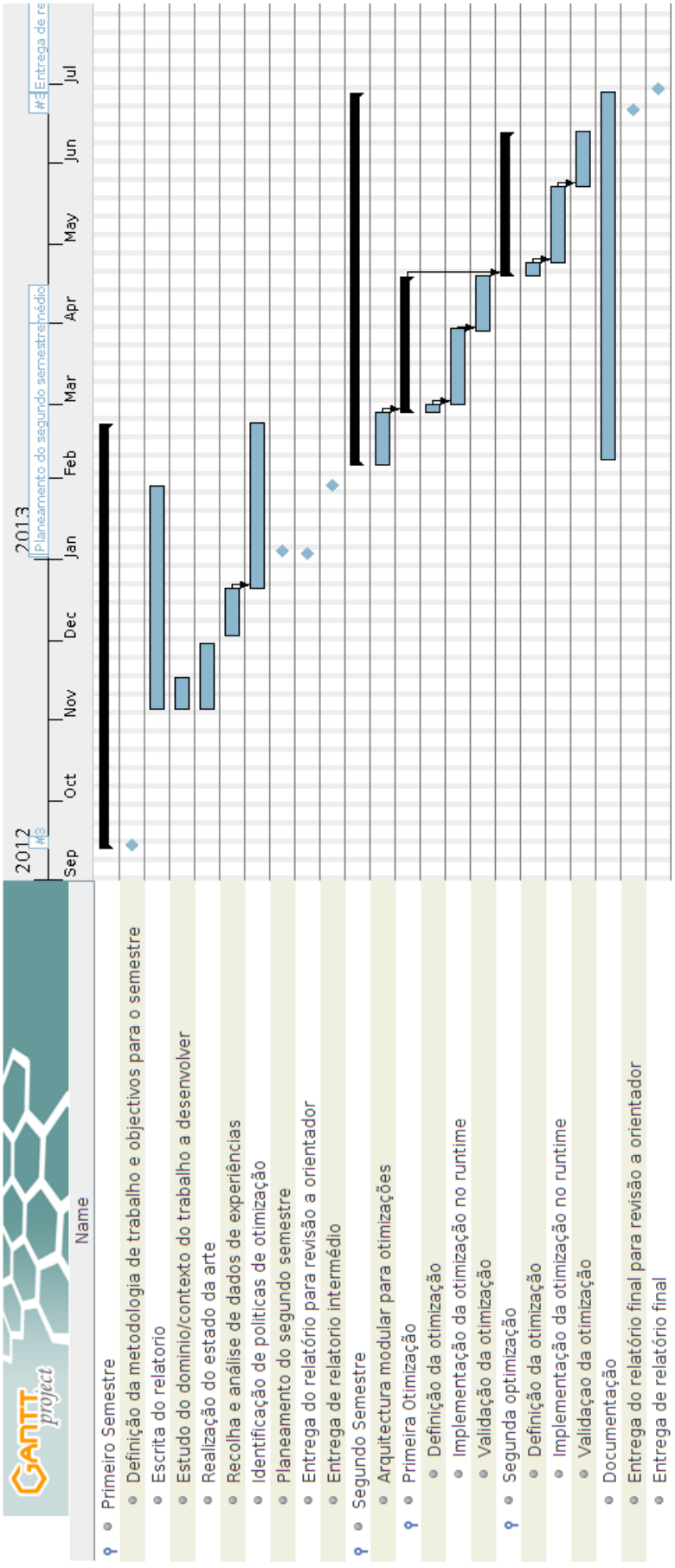

Fig. 23 - Diagrama de Gantt anual

8.2 Equipa

A equipa liderada pelo Professor Doutor Bruno Cabral é constituída pelos seguintes elementos:

- Mestre Alcides Fonseca;
- Eng.º João Rafael – Compilador Aeminium;
- Eng.º Marcelo Sousa – Tratamento de excepções;
- Eng.º Pedro Costa – Compilador Aeminium;
- Eng.º Cristiano Gonçalves – *runtime* Aeminium.

8.3 Riscos

Este projecto, sendo um estágio de investigação podia sofrer de vários riscos que poderiam influenciar a concretização dos seus objectivos. Como se partia do princípio do uso de experiências, para obter-se os resultados a partir dos quais se criavam as políticas a implementar no módulo de optimização do *runtime* Aeminium. Caso as experiências não fornecessem dados favoráveis poderíamos entrar numa procura sem fim definido. Por outro lado, mesmo chegando a políticas viáveis, poderia ainda existir alguma dificuldade de transpor as regras para um módulo capaz de optimizar o código em tempo de execução. Depois de implementadas as políticas, existiu a necessidade de mais programas de teste, no entanto seria impraticável passar o estágio a reescrever código sequencial para código paralelo. Ao nível do código gerado pelo compilador, devido às suas atuais limitações, tais como:

- Não se pode usar variáveis globais;
- Não se pode usar várias classes no mesmo ficheiro;
- Inicialmente não suportava ciclos for, este suporte foi adicionado recentemente;
- Não se pode usar variáveis na declaração do for;
- Não faz tratamento de exceções.

É quase impossível compilar um programa de uso geral, sem que se tenha de fazer grandes alterações no código a compilar. Esta situação, levou a uma escassez de experiências com código gerado pelo compilador.

Capítulo 9 Conclusão

Neste capítulo são expostas as conclusões relativas ao estágio – "*Runtime Optimization of Programs for the Aeminium Platform"*. Este estágio tinha como objectivos compreender os problemas de performance aquando da execução de programas paralelos no *runtime* Aeminium. Através de uma primeira série de experiências e da implementação de um *profiler* foram retiradas três hipótese para investigação. Uma relacionada com o escalonamento de tarefas, outra sobre a decisão de paralelização e a última em relação ao número e tamanho de tarefas.

Para a primeira hipótese foi proposta a utilização de um escalonamento mais equilibrado para melhorar a performance. A implementação de um escalonamento em varrimento de modo a detetar uma fila vazia para inserir a nova tarefa, provou ser melhor opção comparativamente com a implementação de escalonamento aleatória utilizada no *runtime* Aeminium. A implementação utilizada atingiu aumentos de performance até 20% nalguns casos, sendo que em nenhum dos casos apresentou decréscimo de performance. Uma das possíveis razões para esta melhoria é a activação imediata de *workers* que estavam parados - *parked*. Um *worker* sem trabalho e com a sua fila vazia "acorda" por períodos de tempo consecutivos para tentar "roubar" trabalho aos *workers* vizinhos. Caso a inserção seja efetuada directamente nestes *workers*, é possível que se antecipe a sua activação e como consequência melhora-se a performance. Além disso a implementação utilizada pelo *runtime* Aeminium apresentava um elevado número de *steals*. Recorrendo ao *profiler* implementado, verificou-se que após a optimização, o número de tarefas inseridas por *worker* era efetuada de modo mais distribuído e o número de *steals* diminuía.

Na segunda hipótese foi sugerida a utilização da informação do tamanho de todas as filas para auxiliar a decisão de paralelização. Foram utilizados os programas mergesort e kdtree, obtendo-se resultados interessantes, mas de modo diferente para cada um dos programas. No kdtree a melhor performance foi atingida com a verificação da condição: pelo menos metade das filas se encontrar vazias de modo a se optar pela sequencialização. Com esta implementação atingiu-se melhorias de performances até 9%. Já no mergesort a máxima performance é alcançada com a remoção do *threshold* referente ao tamanho da fila e utilizando apenas a verificação da condição: todas as filas possuem tarefas para a decisão de sequencializar. Nesta implementação obtiveram-se melhorias de performance até 55%. Estes resultados apresentam-nos a relevância da utilização da informação relativa ao estado de

todas as filas e não apenas de uma. No entanto a utilização do *threashold* mostrou resultados completamente opostos para os dois programas, sendo um importante ponto a investigar.

Por último, propôs-se a implementação de um módulo para efectuar *merge* de tarefas. Foram efetuadas experiências variando o número mínimo de tarefas no *Cluster* com código gerado manualmente e com código gerado pelo compilador. Os resultados para o código gerado manualmente não mostraram resultados relevantes, no entanto para o código gerado pelo compilador, estes mostraram-se positivos, alcançando melhorias de performance até 30%. Esta heterogeneidade nos resultados poderá ser explicada pelo modo como o código é escrito. Visto que o código gerado pelo compilador tem tendência a possuir muito mais tarefas e dependências que o mesmo código escrito manualmente.

Capítulo 10 Trabalho futuro

No que respeita a trabalho futuro que possa ser realizado no Aeminium Runtime, recomendo algumas hipóteses que podem vir a dar resultados positivos. De seguida exponho algumas dessas hipóteses.

O modo como se efetua o *schedule* das tarefas possui uma razoável influência na performance do runtime Aeminium. A abordagem utilizada nesta investigação baseou-se apenas no uso da informação relativa à detecção de uma fila sem qualquer tarefa, com um varrimento da primeira fila em diante. Uma vez que o algoritmo utilizado é uma espécie de "*brute force*", seria interessante estudar um algoritmo de detecção de filas vazias mais complexo que o utilizado.

Relativamente à optimização da decisão de paralelização ou sequencialização de código, além da informação relativa ao estado de cada fila, seria interessante utilizar informação do sistema para auxiliar a decisão. O sistema possui imensas variáveis que nos podem fornecer indicações acerca da carga a que este está sujeito. Com esta abordagem poderíamos, por exemplo, no caso de alguns processadores estarem com uma taxa de utilização baixa, optar pela decisão de paralelização. A utilização de um *threashold* referente ao tamanho da fila é também um importante ponto a investigar, visto os resultados mostrarem-se bastante heterogéneos. Tendo a natureza dos programas alguma influência na condição e valor a utilizar.

A implementação de um módulo para efectuar *merge* de tarefas obteve melhorias de performance significativas. Mas, devido às limitações impostas nas condições de efectuar *merge*, as experiências em código gerado pelo compilador ficou um pouco limitada. Sendo uma possível implementação, a remoção da condição da tarefa não possuir filhos. Sem esta condição o número de *merges* iria aumentar substancialmente, podendo advir um aumento de performance. A remoção desta condição pode tornar-se complexa devido á gestão de dependências que esta necessitará. Razão pela qual, no tempo disponível neste estágio não foi possível implementar.

Referências

[1] Sven Stork, "Concurrent Programming via Access Permissions", Departamento de Engª Informática, Faculdade de ciências e tecnologias, Universidade de Coimbra,25 de Setembro 2009.

[2] Toshio Suganuma, "Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler", 2007.

[3] Stefan Hepp ,"Optimizing Java ByteCode for Embedded Systems", Instituto de Informática, Universidade Técnica de Viena, 24 de Fevereiro de 2009.

[4] Website. proguard.sourcefourge.net

[5] Raja Vallé-Rai, "Soot – a Java Bytecode Optimization Framework", Sable Research Group, School of Computer Science; McGill University.

[6] Website. www.cs.purdue.edu/homes/hosting/bloat

[7] Thomas Fahringer, "Estimating and optimizing performance form Parallel Programs", Institute for Software Technology and Parallel Systems, University of Vienna, November 1995.

[8] Alcides Fonseca," AeminiumGPU – A CPU_GPU Hybrid Runtime for the Aeminium Language", Departamento de Engª Informática, Faculdade de ciências e Tecnologia, Universidade de Coimbra, 31 de Agosto de 2011.

[9] Rober D. Blufmofe, Charles E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing", The University of Texas at Austin, MIT Laboratory for Computer Science.

[10] Brice Dobry, "Work Stealing in Multiprogrammed Environments", Department of Computer & Information Sciences, University of Delaware.

[11] docs.oracle.com/cd/E13150_01/jrockit_jvm/geninfo/diagnos/underst_jit.html

[12] Kazuaki Ishizaki, "Effectivness of Cross-Platform Optimizations for a Java Just-in-Time Compiler", IBM Research, Tokyo Research Laboratory, Japan, October 2003.

[13] Anderson Faustino da Silva, Vitor Santos Costa, "Our Experiences with Optimizations in Sun`s Java Just-in-Time Compilers", Universidade Federal do Rio de Janeiro, 2006.

[14] Vitor Pankratius, Ali_Reza,Adl-Tabatabai, Walter Tichy, "Fundamentals of multicore Software development", Chapman & Hall/CRC, 20012.

[15] Easwaran Raman, Guilherme Ottoni, Aurun Raman, Matthew J. Bridges, Davidl. August, "Parallel-Stage Decoupled Software Pipelining", Departments of Computer Science and Electrical Engineering, Princeton University, April, 2008.

[16] Keith D. Cooper and Li Xu,"Static Removal of Redundant Loads", Department of Computer Science, Rice University, Houston, Texas, USA.

[17] Nurudeen A.Lameed and Laurie Hendren, “A Modular Approach to On-Stack Replacement in LLVM”, Sable Technical Report NO. sable-2012-01-rev2, mcgill, November 22, 2012.

[18] Andrew S. Tanenbaum, Hans van Staveren, Johan w. Stevenson, “Using Peephole Optimization on Intermediate Code”, Vrije Universiteit, Amsterdam, The Netherlands, January, 1982.

[19] Elena Machkasova, Kevin Arhelger, Fernado Trinciante, “The Observer Effect of Profilling on Dynamic Java Optimizations”, University of Minnesota, Morris, October 2009.

[20] Joel H. Saltz, Ravi Mirchandaney, “The Preprocessed Doacross Loop”, Institute for computer applications in science and engineering, NASA Langley, Reseach center, Hamptom, Virginia 23665, May, 1990.

[21] Raghavan Raman, “Compiler Support for Work-Stealing Parallel Runtime Systems”, Rice University, Houston, Texas, May 2009.

[22] Manuel Mohr, “Aeminium Compilation Theory in the Context of the Plaid Language”, Department of Informatics, Karlsruhe of technology, February 24, 2011.

[23] “The Aeminium Infrastructure Design”, The Aeminium Project, April 12, 2010.

[24] Yuan-Fu Sheiue, Wei-Cheng Chen, Ching-Huan Lee, Peng-Sheng Chen, “A practical Software-based Programming Model for Thread-level Speculation”, Department of Computer Science and Information Engineering , National Chung Cheng University, Chia-Yi, Taiwan, May 3, 2012.

[25] J. Gregory Steffan and Todd C. Mowry, “The Potencial for Using Thread-level Data Speculation to facilitate Automatic Parallelization”, Department of Computer Science, Carnegie Mellon University.

[26] Aart J.C., Juan E. Villacis and Dennis B. Gannon, “Javar Manual (version 1.3BETA), Computer science department, Indiana University, Lindley Hall 215, Bloomington, Indiana 47405-4101, USA.

[27] Feng Liu, “Introduction to Cilk Programming”, COS597C.

[28] ObjectAid UML Explorer for Eclipse . http://www.objectaid.com. [acedido em 27/02/2013]

[29] SISAL programing language, http://www2.cmp.uea.ac.uk. [acedido em 21/05/2013]

Anexos

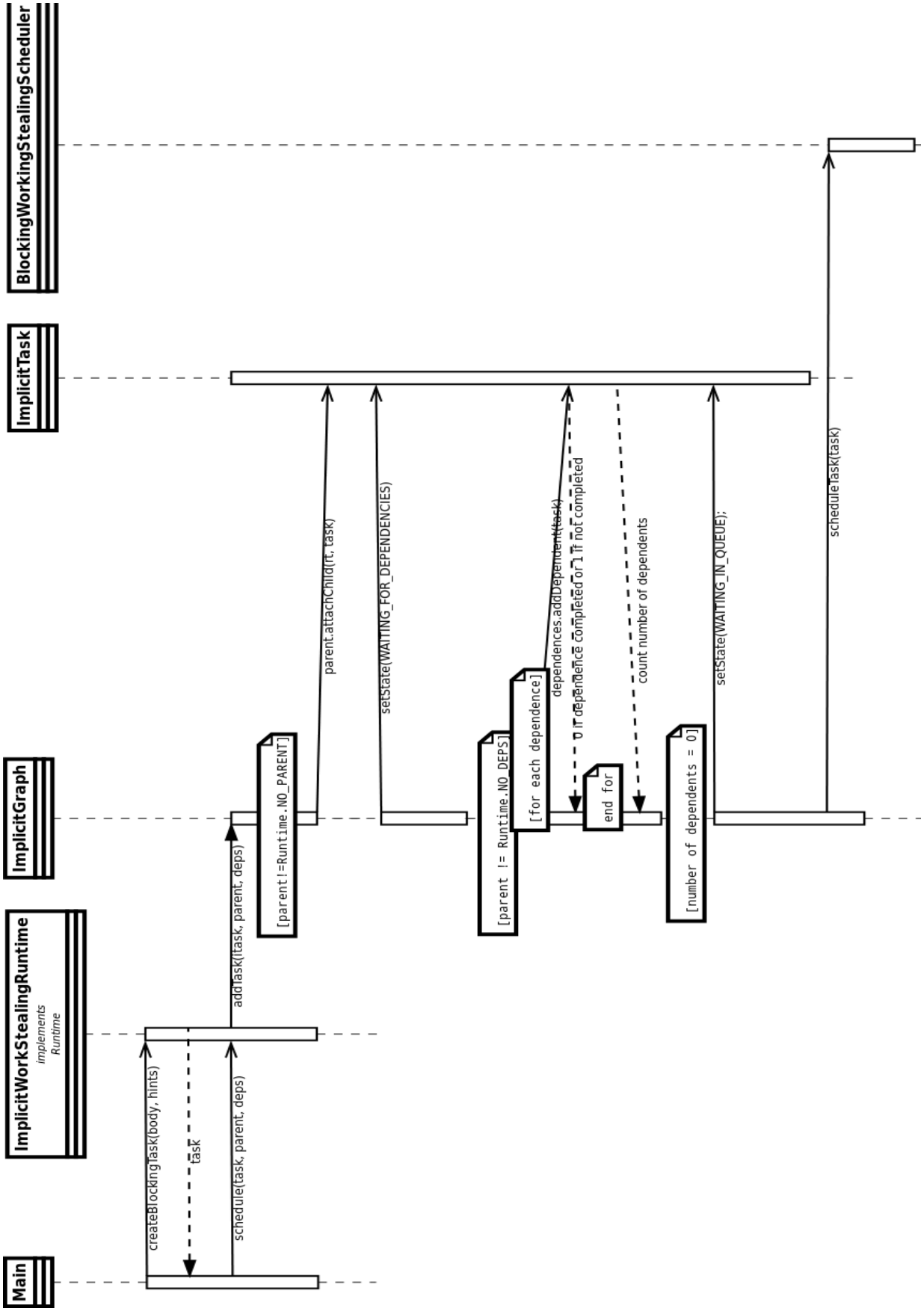

Fig. 24 - Diagrama de sequência antes do escalonamento da tarefa

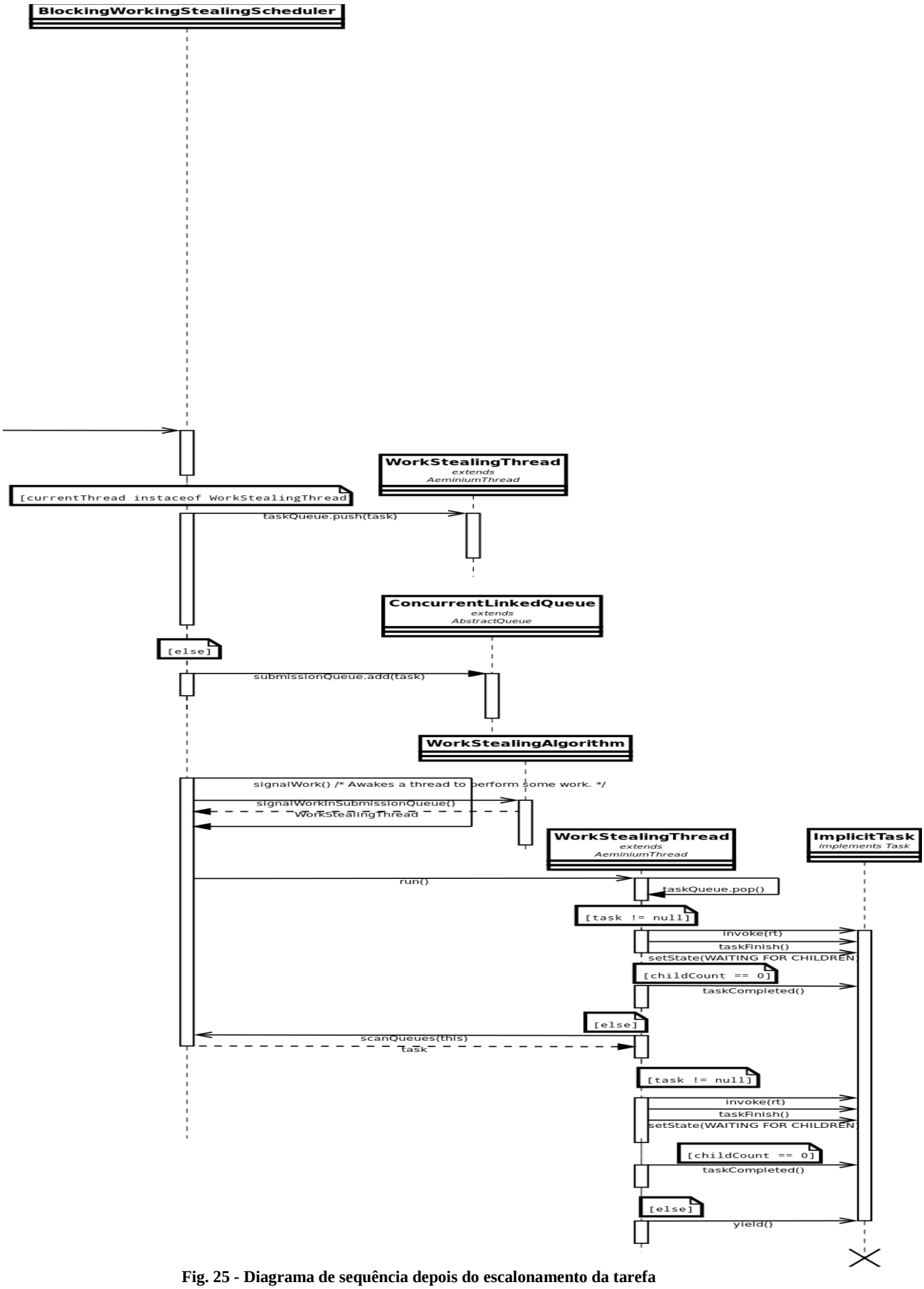

Fig. 25 - Diagrama de sequência depois do escalonamento da tarefa

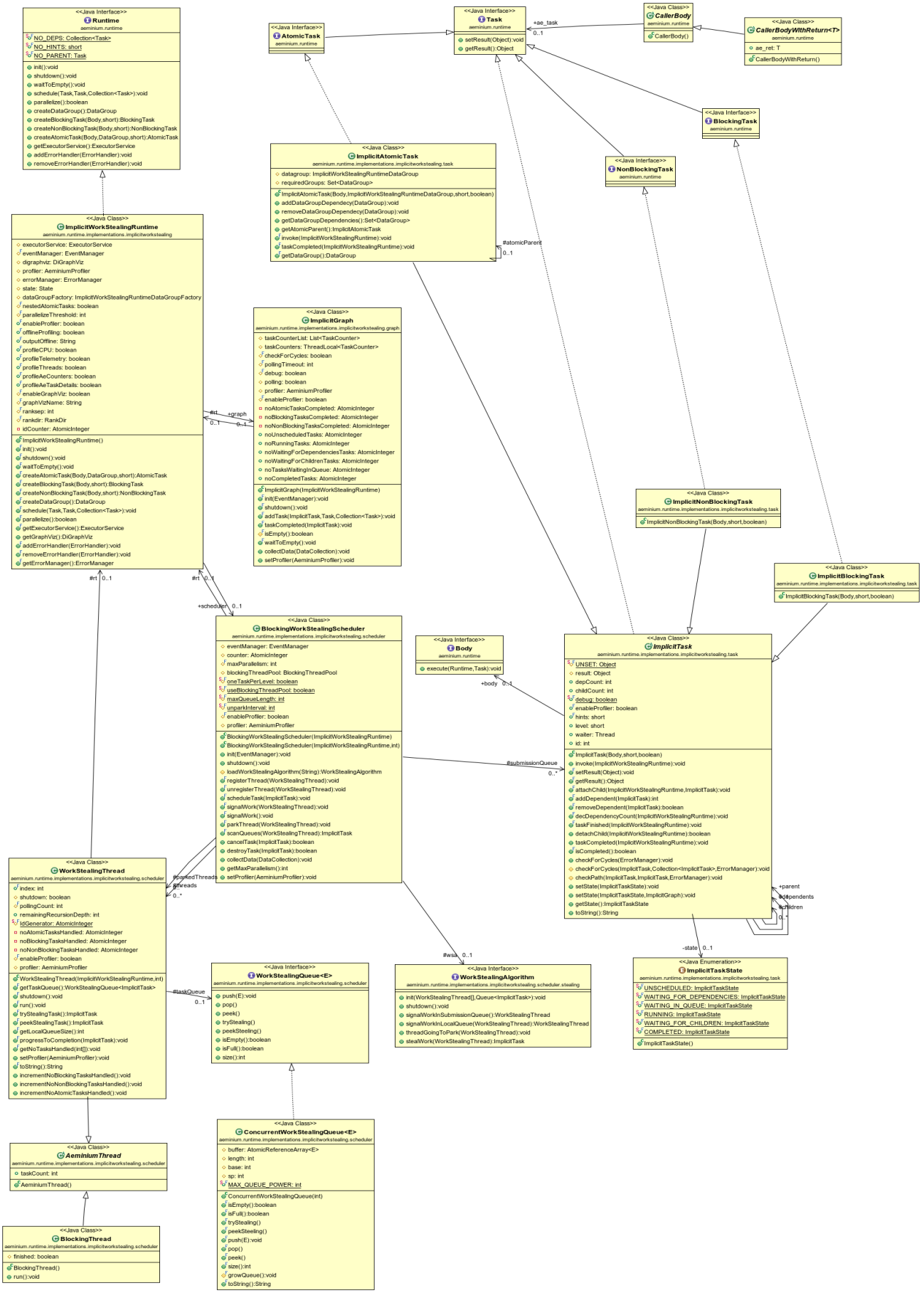

Fig. 26 - Diagrama de classes do Aeminum Runtime

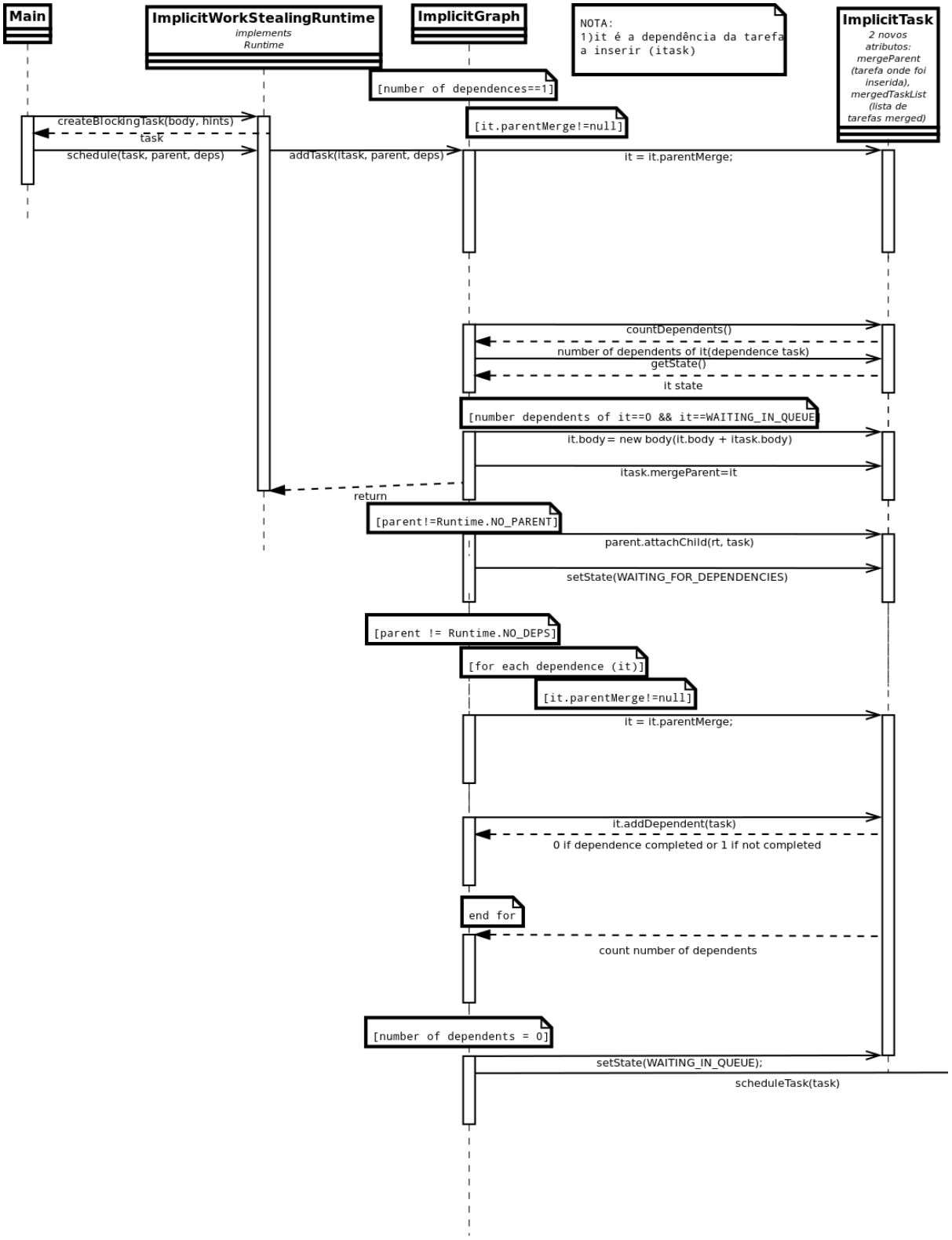

Fig. 27 - Diagrama de sequência antes do escalonamento da tarefa (otimização com falhas)

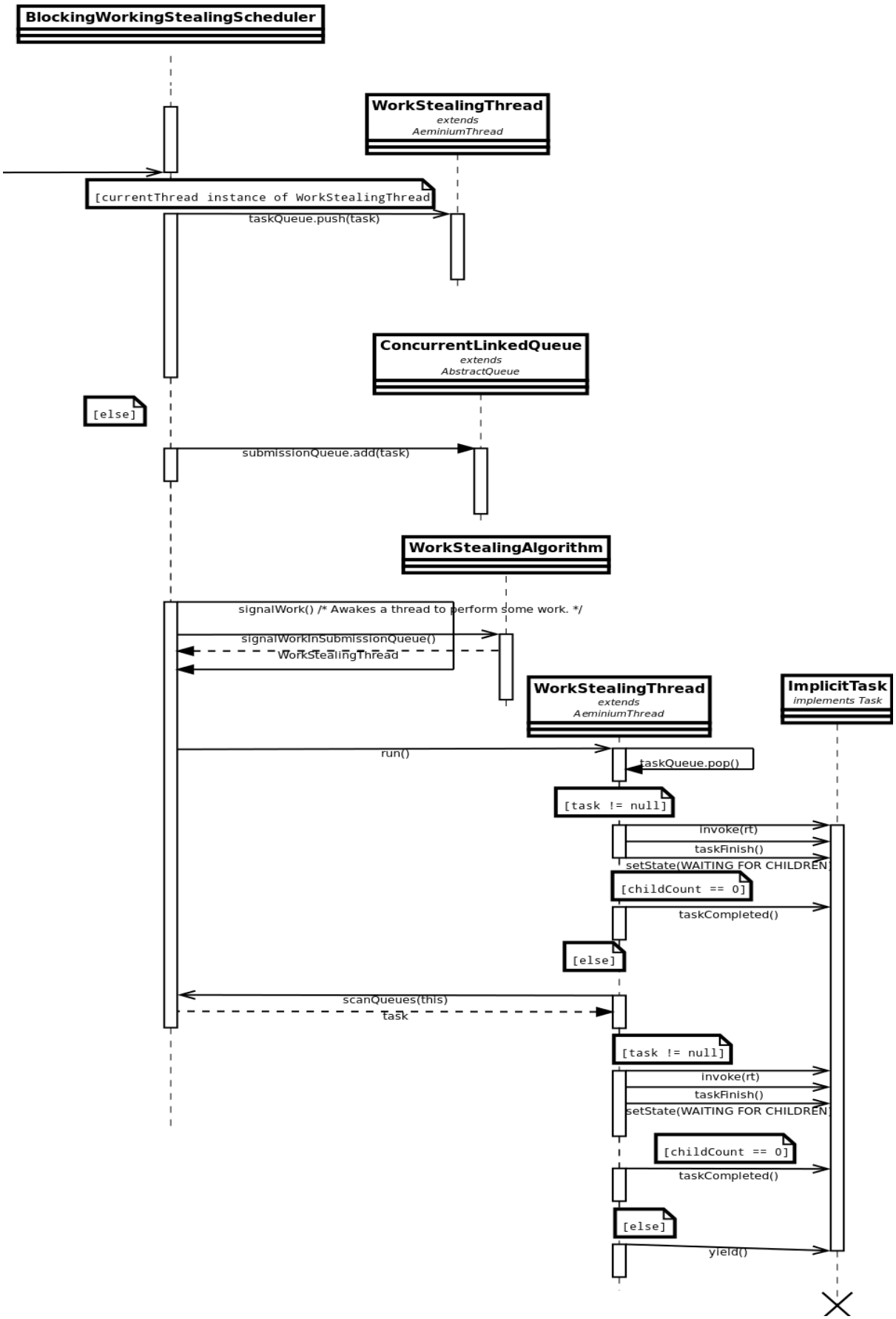

Fig. 28 - Diagrama de sequência depois do escalonamento da tarefa (otimização com falhas)

```
**************************************STATISTICS BY WORKER*****************************
TOTAL TASKS PERFORMED: 28400.0
workerId0: 596(2.1%) | workerId1: 1016(3.58%) | workerId2: 553(1.95%) | workerId3: 607(2.14%) | workerId4: 340(1.2%) | workerId5:
275(0.97%) | workerId6: 290(1.02%) | workerId7: 758(2.67%) | workerId8: 1012(3.56%) | workerId9: 1717(6.05%) | workerId10:
1007(3.55%) | workerId11: 1113(3.92%) | workerId12: 705(2.48%) | workerId13: 822(2.89%) | workerId14: 481(1.69%) | workerId15:
298(1.05%) | workerId16: 74(0.26%) | workerId17: 8305(29.24%) | workerId18: 3100(10.92%) | workerId19: 1680(5.92%) | workerId20:
871(3.07%) | workerId21: 522(1.84%) | workerId22: 1450(5.11%) | workerId23: 808(2.85%) |
TOTAL STEALS PERFORMED BY WORKER: 27908.0
workerId0: 567(2.03%) | workerId1: 981(3.52%) | workerId2: 540(1.93%) | workerId3: 586(2.1%) | workerId4: 340(1.22%) | workerId5:
266(0.95%) | workerId6: 290(1.04%) | workerId7: 738(2.64%) | workerId8: 995(3.57%) | workerId9: 1689(6.05%) | workerId10:
987(3.54%) | workerId11: 1084(3.88%) | workerId12: 698(2.5%) | workerId13: 812(2.91%) | workerId14: 465(1.67%) | workerId15:
293(1.05%) | workerId16: 72(0.26%) | workerId17: 8228(29.48%) | workerId18: 3037(10.88%) | workerId19: 1640(5.88%) | workerId20:
857(3.07%) | workerId21: 518(1.86%) | workerId22: 1432(5.13%) | workerId23: 793(2.84%) |
TOTAL OF NO STEALS PERFORMED BY WORKER: 492.0
workerId0: 29(5.89%) | workerId1: 35(7.11%) | workerId2: 13(2.64%) | workerId3: 21(4.27%) | workerId4: 0(0%) | workerId5: 9(1.83%)
| workerId6: 0(0%) | workerId7: 20(4.07%) | workerId8: 17(3.46%) | workerId9: 28(5.69%) | workerId10: 20(4.07%) | workerId11:
29(5.89%) | workerId12: 7(1.42%) | workerId13: 10(2.03%) | workerId14: 16(3.25%) | workerId15: 5(1.02%) | workerId16: 2(0.41%) |
workerId17: 77(15.65%) | workerId18: 63(12.8%) | workerId19: 40(8.13%) | workerId20: 14(2.85%) | workerId21: 4(0.81%) |
workerId22: 18(3.66%) | workerId23: 15(3.05%) |
TASK ID SEQUENCE BY WORKER:
workerId0: 0,4,295,296,297,300,310,309,...
TASK TYPE ID SEQUENCE BY WORKER:
workerId0: 1,1,2,2,2,3,4,4,4,4,2,2,4,5,2,4
TOTAL OF TASKS INSERTED BY WORKER QUEUE: 28299.0
workerId0: 890(3.14%)
workerId1: 611(2.16%)
workerId2: 847(2.99%)
workerId3: 49(0.17%)
workerId4: 279(0.99%)
workerId5: 284(1%)
workerId6: 267(0.94%)
workerId7: 1117(3.95%)
workerId8: 2430(8.59%)
workerId9: 1454(5.14%)
workerId10: 1099(3.88%)
workerId11: 868(3.07%)
workerId12: 551(1.95%)
workerId13: 559(1.98%)
workerId14: 285(1.01%)
workerId15: 17(0.06%)
workerId16: 7000(24.74%)
workerId17: 3617(12.78%)
workerId18: 2074(7.33%)
workerId19: 880(3.11%)
workerId20: 44(0.16%)
workerId21: 1399(4.94%)
workerId22: 858(3.03%)
workerId23: 820(2.9%)
**************************************STATISTICS BY TASK TYPE**************************
TOTAL OF TASKS BY TYPE: 28400.0
typeId1: 100(0.35%) | typeId2: 10000(35.21%) | typeId3: 100(0.35%) | typeId4: 9000(31.69%) | typeId5: 9000(31.69%) | typeId6:
100(0.35%) | typeId7: 100(0.35%) ...
TOTAL OF DEPENDENTS (id) BY TASKS TYPE: 1253.0
typeId2: 3 dependents -> id [17220][20604][28218](0.24%) …
typeId3: 1150 dependents -> id [301][302][303][304][305][306][307][308][309][310][583][865][1147][1429][1711][1993]...
typeId5: 1 dependents -> id [25015](0.08%) |
typeId7: 99 dependents -> id [101][102][103][105][107][109][111][113][114][116][118][120][122]…
typeId2: 3 dependents -> typeId [3][3][3](0.24%) |
typeId3: 1150 dependents -> typeId [4][4][4][4][4][4][4][4][4][4][4][4][4][4][4]…
typeId5: 1 dependents -> typeId [6](0.08%) |
typeId7: 99 dependents -> typeId [7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7][7]…
**************************************STATISTICS BY QUEUE TYPE**************************
TOTAL OF TASKS BY QUEUE: 28400.0
taskQueue: 492 | stealQueue: 27908 | executed: 0
```

Fig. 29 - Output do profiler implementado no Aeminium Runtime