**Bruno da Silva de Oliveira**

# *Hydra: Compilação Distribuída de Código Fonte*

Florianópolis – SC

Novembro / 2004

**Bruno da Silva de Oliveira**

# *Hydra: Compilação Distribuída de Código Fonte*

Orientador:
Prof. Dr. Mario Dantas

Bacharelado em Ciências da Computação
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina

Florianópolis – SC

Novembro / 2004

Trabalho de Final de Curso sob o título *"Hydra - Compilação Distribuída de Código Fonte"*, proposta por Bruno da Silva de Oliveira e aprovado em 9 de novembro de 2004, em Florianópolis, Santa Catarina, por:

---

Prof. Dr. Mario Dantas
Departamento de Informática e Estatística - UFSC
Orientador

---

Prof. Dr. João Bosco Mangueira Sobral
Departamento de Informática e Estatística - UFSC

---

Prof. Dr. José Mazzucco Jr.
Departamento de Informática e Estatística - UFSC

*Dedico este trabalho a meus pais:*
*Minha mãe, companheira de todas as horas;*
*Meu pai, exemplo de honestidade e sem o mesmo nada disto seria possível (in memorium).*

# *Agradecimentos*

Dedico meus sinceros agradecimentos para:

– o professor Dr. Mario Dantas, pela orientação, dedicação e sincera amizade;

– Clovis R. Maliska Júnior, diretor presidente da ESSS, pela amizade e dicas relacionadas ao trabalho e à metodologia adotada;

– o colega M. Sc. Eduardo Kern da ESSS, pela revisão deste trabalho e auxílio na implementação;

– a equipe de desenvolvimento da ESSS, pela colaboração nos testes da ferramenta;

# *Resumo*

As linguagens de programação são as ferramentas mais utilizadas pelo desenvolvedor de software durante seu dia de trabalho. Linguagens compiladas trazem um problema ao desenvolvimento, que é o tempo necessário para a compilação do código fonte. Ao longo do dia, o desenvolvedor gasta vários minutos, às vezes horas, compilando seu código. Este tempo é desperdiçado, quebrando a linha de raciocínio do desenvolvedor, gerando perda de produtividade e frustração.

Este trabalho apresenta um projeto que visa diminuir o tempo de compilação de projetos codificados na linguagem de programação C ou C++, através da distribuição do trabalho de compilação entre os computadores de uma rede local. Implementado na linguagem de programação de código aberto Python e seguindo a filosofia XP, tem como principais características ser multi-plataforma, independente de compilador usado e pouca intrusão no sistema de *build* utilizado pelo usuário.

Experimentações foram realizadas em projetos reais em uma empresa de desenvolvimento de software, com bibliotecas utilizadas por softwares comerciais. A ferramenta se provou eficaz para certos tipos de bibliotecas, enquanto mostrou baixo desempenho em outras. Propostas são apresentadas como trabalhos futuros que visam eliminar totalmente os problemas encontrados, sem perder as características desejadas originalmente.

# *Sumário*

**Lista de Figuras**

**1 Introdução** p. 9

**2 Embasamento Teórico** p. 11

2.1 Desenvolvimento de Software . . . . . p. 11

2.2 Linguagens de Programação . . . . . p. 11

2.3 Compilação . . . . . p. 12

2.4 C++ . . . . . p. 14

2.4.1 Template Meta-Programming . . . . . p. 14

2.4.2 Pre-Processamento . . . . . p. 15

2.5 Motivação . . . . . p. 16

2.5.1 Imago . . . . . p. 17

2.5.2 Cyclope . . . . . p. 18

2.6 Soluções Existentes . . . . . p. 19

2.6.1 distcc . . . . . p. 19

2.6.2 ccache . . . . . p. 20

2.6.3 IncrediBuild . . . . . p. 20

2.6.4 DMS . . . . . p. 21

2.6.5 TeamBuilder . . . . . p. 21

**3 Projeto** p. 22

3.1 Descrição do Sistema . . . . . p. 22

3.1.1 Hydra . . . . . p. 24
3.1.2 Hydra Server . . . . . p. 24
3.1.3 Hydra Controller . . . . . p. 25
3.1.4 Configuração . . . . . p. 26
3.1.5 Diferenciais . . . . . p. 27
3.2 Ferramentas de Desenvolvimento Adotadas . . . . . p. 28
3.2.1 Python . . . . . p. 28
3.2.2 Metodologia: XP . . . . . p. 28
3.2.3 Editor/IDE . . . . . p. 29
3.2.4 Hardware . . . . . p. 29
3.2.5 Sistemas Operacionais . . . . . p. 30
3.3 Código Fonte . . . . . p. 30

**4 Resultados** p. 31
4.1 Resultados Desejados . . . . . p. 31
4.2 Metodologia de Testes . . . . . p. 31
4.2.1 COI-Lib . . . . . p. 33
4.2.1.1 Análise de Escalabilidade . . . . . p. 33
4.3 Cyclope . . . . . p. 38
4.3.0.2 Análise de Escalabilidade . . . . . p. 41

**5 Conclusões e Trabalhos Futuros** p. 44

**Referências** p. 46

**Anexo** p. 47
Anexo A: Código Fonte . . . . . p. 47

# *Lista de Figuras*


1 Imago 2.1 . . . . . p. 17

2 Cyclope 2.0 . . . . . p. 18

3 Distcc Monitor . . . . . p. 20

4 IncrediBuild . . . . . p. 21

5 Hydra . . . . . p. 22

6 Visão Geral do Sistema . . . . . p. 23

7 Compilação COI-Lib: 2 máquinas . . . . . p. 34

8 Compilação COI-Lib: 3 máquinas . . . . . p. 34

9 Compilação COI-Lib: 4 máquinas . . . . . p. 35

10 Compilação COI-Lib: 5 máquinas . . . . . p. 35

11 COI-Lib: Escalonamento . . . . . p. 36

12 COI-Lib: Profiling . . . . . p. 37

13 COI-Lib: Profiling 2 . . . . . p. 38

14 Compilação Cyclope: 2 máquinas . . . . . p. 39

15 Compilação Cyclope: 3 máquinas . . . . . p. 40

16 Compilação Cyclope: 4 máquinas . . . . . p. 40

17 Compilação Cyclope: 5 máquinas . . . . . p. 41

18 Cyclope: Escalonamento . . . . . p. 42

19 Cyclope: Profiling . . . . . p. 42

20 Cyclope: Profiling 2 . . . . . p. 43

# 1 Introdução

A informatização é um dos fenômenos que mais causou impacto no modo de vida da população mundial nos últimos anos. Coisas até então impensáveis se tornaram prática comum: computadores pessoais acessíveis, pessoas se comunicando instaneamente de diferentes partes do globo de maneira simples e barata, ferramentas computacionais auxiliando cada vez mais o dia-a-dia das pessoas.

Um personagem importante neste cenário é o software. Ferramenta intangível, só existindo na memória de um computador ou dentro de um CD-ROM, é o que permite que nossos computadores gerenciem e-mails, calculem panilhas, analisem resultados coletados de reservatórios de petróleo, façam caracterização de imagens, entre tantos outros usos. Vários outros objetos do nosso dia-a-dia contém um software que controla todas ou partes de suas funções, desde o descongelador automático de uma geladeira até a injeção eletrônica dos automóveis.

Desenvolvedor de Software, ou Programador, é o profissional que cria estas ferramentas. Ele comunica suas intenções ao computador através de uma *linguagem de programação*, que são uma série de regras sintáticas e semânticas que lhe permitem comunicar seu intento em um dialeto que a máquina posteriormente é capaz de compreender; este código é chamado de *código fonte*. Este código fonte deve ser traduzido em uma linguagem que possa ser compreendida pela máquina, processo chamado de *compilação*, que precisa ser realizado diversas vezes por dia à medida que o software vai sendo desenvolvido.

Com o aumento dos sistemas de software, tanto em matéria de escopo de funcionalidades quanto custo e tempo de desenvolvimento, vêm-se investindo cada vez mais em metodologias e práticas que melhorem o desempenho do programador, buscando um processo de desenvolvimento mais eficiente e que gere resultados mais confiáveis.

Uma das maneiras de melhorar a eficiência do programador é evitar interrupções e distrações desnecessárias, pois estas tendem a quebrar a linha de raciocínio do mesmo, gerando perda de tempo, frustração e baixa produtividade. A compilação do código

fonte tende a ser uma interrupção constante ao longo do processo de desenvolvimento, e dependendo do tempo gasto pode ter um impacto significativo na produtividade.

Este trabalho visa desenvolver um sistema que diminua o tempo gasto com o processo de compilação, enviando os diversos arquivos de código fonte que compõem um projeto de software para as máquinas de uma rede local, dividindo assim o tempo necessário. Ele surgiu a partir da necessidade de uma empresa de software, onde foi verificado que muito do tempo dos desenvolvedores era gasto neste processo, causando uma perda significativa de produtividade. Espera-se que a utilização do sistema reduza esta perda, diminuindo custos e melhorando o produto final.

# 2 Embasamento Teórico

## 2.1 Desenvolvimento de Software

O desenvolvimento de software tem amadurecido bastante nos últimos anos. Nos primórdios do desenvolvimento haviam poucas metodologias que visavam melhorar a produtividade, estabilidade e confiabilidade dos sistemas que eram desenvolvidos. Cada desenvolvedor simplesmente trabalhava da maneira que achava mais conveniente e simples, normalmente sem nenhum rigor técnico ou análise, levando a sistemas inseguros, pouco flexíveis e de cara manutenção.

As causas descritas acima levaram ao início da *Engenharia de Software*, que busca identificar muitos dos problemas que os desenvolvedores enfrentam e propor soluções, visando a produtividade e a qualidade do software. A produtividade de um ambiente de desenvolvimento é resultado de diversos fatores, como metodologias, linguagens de programação, ferramentas utilizadas, etc. Todos estes fatores possuem uma influência na produtividade, qualidade e flexibilidade de um sistema, e por consequência, no sucesso e no fracasso de um projeto de desenvolvimento de software.

## 2.2 Linguagens de Programação

As linguagens de programação são uma das principais ferramentas utilizadas pelo desenvolvedor de software. O mesmo lida com a linguagem e suas ferramentas diariamente, constituindo um fator de grande influência no sistema a ser desenvolvido, tanto em termos de produtividade quanto de futura manutenção. A linguagem também é responsável por diversos outros aspectos, como flexibilidade, performance e custo.

Uma linguagem de programação pode ser assim definida (WIKIPEDIA, 2004):

> "'Uma **linguagem de programação** ou **linguagem de computador** é técnica de comunicação para expressar instruções a um computador. É

> um conjunto de regras semânticas e sintáticas usadas para criar programas de computador. Uma linguagem permite ao programador especificar precisamente que dados um computador vai agir sobre, como esses dados vão ser guardados/transmitidos, e precisamente que ações tomar sobre várias circunstâncias. "'

Elas são normalmente classificadas em *compiladas* ou *interpretadas*. Como exemplo de linguagens compiladas, temos C, C++, Pascal e Fortran. Como linguagens interpretadas, temos Java, Python, Perl, PHP, Bash. Apesar de uma linguagem por si só não exigir que ela seja compilada ou interpretada, as linguagens são assim normalmente classificadas devido à sua implementação mais comum ou tradicional. Por exemplo, apesar de C ser considerada uma linguagem compilada, existe um interpretador para a mesma chamado *Ch* (SOFTINTEGRATION, 2001), que permite que o código seja executado diretamente sem necessitar de um passo intermediário de compilação.

A diferença básica entre elas é que as linguagens compiladas traduzem seus códigos fontes normalmente para código nativo da máquina, com o objetivo de obter maior performance computacional (tanto em termos de utilização de CPU quanto memória); esta técnica possui a desvantagem de restringir o código à maquina em que foi gerado, sendo que caso seja necessário utilizar o software em outra máquina com diferente arquitetura, uma nova compilação do código fonte será necessária.

Em uma linguagem interpretada, o código é interpretado por uma *máquina virtual*. Uma máquina virtual é um programa de computador que interpreta um código fonte escrito em determinada linguagem e traduz as diretivas encontradas em tempo real para instruções entendidas pela máquina verdadeira. O código fonte pode ser interpretado diretamente pela máquina virtual, ou pode passar por um passo de compilação, em que o código é transformado em um código objeto que a máquina virtual entende, normalmente denominado *bytecode*. Esse código objeto é portável entre diferentes sistemas operacionais e até mesmo diferentes arquiteturas, bastando para isto que exista um interpretador nativo. Esta característica facilita bastante a distribuição do programa entre diferentes plataformas.

## 2.3 Compilação

Como descrito em (WIKIPEDIA, 2004), um compilador é um programa de computador que traduz o código fonte escrito em uma linguagem de alto-nível para uma linguagem que poderá ser executada diretamente por um computador ou uma máquina virtual.

Muitos compiladores modernos compartilham um design de compilação em *dois estágios*. O primeiro estágio é chamado de *front-end*, e traduz o código fonte em uma representação intermediária. O segundo estágio é chamado de *back-end*, que trabalha com esta representação intermediária para produzir a linguagem de saída. Esta abordagem permite diminuir a complexidade separando as preocupações do *front-end*, que normalmente envolvem semânticas da linguagem, checagem de erros, etc, das preocupações do *back-end*, que se concentra em produzir saída que seja eficiente e correta. Também possui a vantagem de permitir o uso de somente um *back-end* para múltiplas linguagens de programação, e similarmente o uso de diferentes *back-ends* para diferentes máquinas-alvo.

O processo do *front-end* consiste normalmente de múltiplos passos, cada um descrito pela teoria das Linguagens Formais:

1. **Análise Léxica** - Consiste em quebrar o código fonte em pedaços menores (chamados de *tokens*), cada um representando unidade atômica da linguagem, como por exemplo, uma palavra-chave, identificador ou números.

2. **Análise Sintática** - Identifica estruturas sintáticas no código, focando somente nas estruturas hierárquicas e ordem dos *tokens*.

3. **Análise Semântica** - reconhece o *significado* do código fonte e prepara para gerar a saída. Nesta fase, checagem de tipos é feita e aparecem a maioria dos erros de compilação.

4. **Geração da Linguagem Intermediária** - um equivalente do programa de entrada é gerado em uma linguagem intermediária.

Enquanto existem aplicações que somente o *front-end* é necessário, como ferramentas de análise estática de código, um compilador real passa a representação intermediária gerada pelo *front-end* para o *back-end*, que gera um programa equivalente na linguagem-alvo. Isto é feito em múltiplos passos:

1. **Análise de Compilação** - É o processo de recolher informação da representação intermediária de entrada. Este processo é vital para geração de otimizações do código.

2. **Otimização** - A representação intermediária é transformada em funcionalidade equivalente mas mais rápida (ou menor).

3. **Geração de Código** - A representação intermediária transformada é traduzida para a linguagem de destino, normalmente a linguagem de máquina nativa do sistema. Isto envolve decisões de alocação de recursos e memória, além da seleção e agendamento das instruções de máquinas apropriadas.

## 2.4 C++

A linguagem de programação C++ é assim descrita pela *Wikipedia* (WIKIPEDIA, 2004):

> C++ é uma linguagem de programação de propósito geral. É *estaticamente tipada*, de *forma-livre*, *multi-paradigma*, suportando programação procedural, abstração de dados, programação orientada a objetos e programação genérica. Durante os anos 90, C++ se tornou uma das linguagens de programação mais populares. Ninguém detém os direitos da linguagem, e ela é livre para uso.

C++ é bastante popular em várias áreas do desenvolvimento de software. Suas características a tornam especialmente atrativa para o a área de softwares voltados à engenharia, como simuladores, pós-processadores, solvers, etc. Este ramo é dominado por softwares que lidam com grandes quantidades de dados e/ou algoritmos intensivos, sendo portanto necessário a geração de um código eficiente em termos tanto de CPU quanto memória. Associado à técnicas de *Generic Programming*, uma eficiência ainda maior pode ser obtida.

### 2.4.1 Template Meta-Programming

De acordo com a *Wikipedia* (WIKIPEDIA, 2004), *Template Meta-Programming* é uma técnica de programação utilizada para obter o resultado de algumas expressões em tempo de compilação usando as facilidades de templates e programação genérica do C++.

Como exemplo das capacidades desta técnica, vamos analisar um exemplo simples, o cálculo de uma fatorial.

Utilizando recursividade (a implementação tradicional), o código resultante seria este:

```
int factorial(int n) {
  if (n == 1)
    return 1;
```

```
  else
    return n * factorial(n - 1);
}

//factorial(4) == (4*3*2*1) == 24
```

A solução utilizando template metaprogramming poderia ser esta:

```
template <int N>
struct Factorial {
  enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<1> {
  enum { value = 1 };
};

//Factorial<4>::value == 24
```

Esta solução utiliza *template specialization* para prover a condição de finalização da recursividade.

A diferença entre as duas técnicas é que na primeira, implementada tradicionalmente, o cálculo da fatorial é feito em tempo de execução, enquanto na segunda, todo o cálculo é feito em tempo de compilação.

Template metaprogramming, quando bem aplicada, resulta em código extremamente eficiente, a um custo maior de tempo de compilação.

### 2.4.2 Pre-Processamento

Em se tratando da linguagem de programação C++, antes mesmo de o código ser enviado para o *front-end* do compilador, ele passa por um processo chamado de *Pré-Processamento*. Este processo normalmente é executado automaticamente por uma ferramenta chamada de pré-processador.

A fase de pré-processamento basicamente substitui texto no código fonte antes de ser enviado ao compilador, sendo basicamente as seguintes ações:

- Linhas que terminam com uma barra invertida são unidas com a linha seguinte.
- Comentários são removidos e substituídos por um espaço simples.
- Diretivas de pré-processamento são executadas.

Diretivas de processamento são aquelas que iniciam com o símbolo #, e são usadas para incluir outros arquivos, criar macros, substituição de constantes, e compilação condicional.

## 2.5 Motivação

Durante o processo de desenvolvimento na linguagem de programação C++, o desenvolvedor passa grande parte do tempo na fase de compilação-linkagem do código.

Em sistemas com grande quantidade de linhas de código, ou que fazem o uso de *Template Metaprogramming* com o objetivo de obter o melhor desempenho possível, o tempo necessário para fazer a compilação completa do sistema pode ultrapassar horas. Apesar de nem todo o código precisar ser compilado a cada modificação do programador, muitas vezes uma pequena modificação em um arquivo importante acaba gerando a necessidade de uma compilação completa do sistema. O resultado é frustração do desenvolvedor, perda de produtividade e tempo.

A empresa *ESSS* (Engineering Simulation and Scientific Software - (ESSS, 2004b)) presta serviços de consultoria e desenvolvimento de software à empresas de diversos setores voltados à engenharia. No setor de desenvolvimento de software, a equipe busca produzir tecnologias voltadas a atender as necessidades de visualização, estruturas de dados e processos científicos das empresas com quem trabalha.

A equipe produziu diversas destas bibliotecas utilizando-se de técnicas de *Template Meta-Programming*, visando a melhor performance e flexibilidade possível. No entanto, com o tempo verificou-se que o tempo necessário para se compilar estas bibliotecas estavam causando um impacto perceptível na produtividade dos desenvolvedores, sendo apresentados alguns exemplos nas próximas seções.

### 2.5.1 Imago

O software Imago System (IMAGO, 2004), desenvolvido pela ESSS, diversas técnicas e algoritmos para caracterização e quantificação das propriedades microestruturais de materiais analisados em laboratório. As propriedades físicas macroscópicas dos materiais são decorrentes da sua microestrutura, e o software permite uma análise precisa e confiável, sem a necessidade de ensaios físicos muitas vezes caros e/ou demorados. A figura 1 mostra uma imagem da interface do Imago versão 2.1.

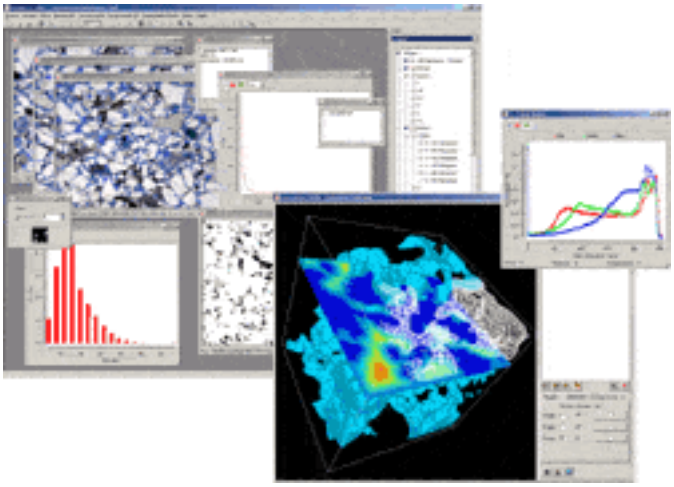

Figura 1: Interface do Imago 2.1

Contando com cerca de 800 classes espalhadas em 1710 arquivos, o tempo necessário para compilar o sistema por completo é de cerca de 40 minutos em um computador Pentium 4 de 2.8 GHz, considerado *"'top de linha"'* no momento da escrita deste documento.

Um estudo realizado junto ao principal desenvolvedor do projeto concluiu que cerca de 36% do tempo de desenvolvimento era gasto com o processo de compilação, sendo a máquina utilizada um Athlon 1.4 GHz. Depois, este desenvolvedor passou a utilizar uma máquina equipada com um Pentium 4 de 2.8GHz, e o tempo gasto com compilação caiu para 11%. Considerando-se uma semana de trabalho de 40 horas semanais, cerca de **quatro horas e meia** são desperdiçadas atualmente somente com a compilação do sistema, tempo que o desenvolvedor fica ocioso e sem produzir. A situação era ainda pior na antiga máquina, onde eram gastas **quatorze horas e meia** por semana, um considerável tempo disperdiçado.

### 2.5.2 Cyclope

O software Cyclope (ESSS, 2004a), desenvolvido também pela ESSS, é um pós-processador de reservatórios de petróleo, onde os dados gerados por simuladores podem ser analisados e visualizados com precisão. Conta ainda com a capacidade de conversão entre diversos formatos de arquivos usados por diferentes simuladores, permitindo assim uma fácil cooperação entre diferentes ferramentas utilizadas pelos engenheiros de reservatório.

Entre os requisitos do software, está a necessidade de um *kernel* numérico de processamento de dados robusto e eficiente, tanto em termos de memória quanto de CPU. Pensando nisto, a equipe desenvolveu uma biblioteca de mesmo nome, que provém as facilidades necessárias para lidar com as peculiaridades das malhas de petróleo, e que faz um pesado uso de *Template Metaprogramming*.

Na figura 2, é mostrada uma imagem da interface gráfica do Cyclope 2.0.

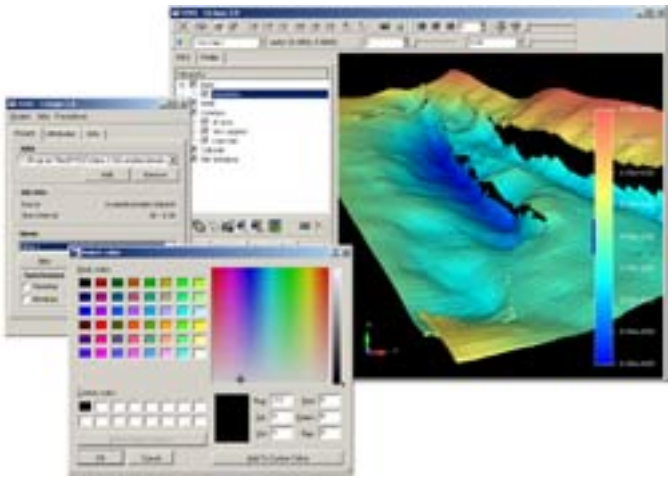

Figura 2: Interface do Cyclope 2.0

Apesar de extremamente eficiente, o tempo de compilação da biblioteca é muito alto: sendo formada por apenas 24 arquivos fonte, consome cerca de 20 minutos e 20 segundos, isto em um AMD Athlon 1400+ com 1.5GB de memória RAM.

Os dados apresentados indicam que o tempo gasto com a compilação pode ser um problema sério no dia a dia do desenvolvedor, causando frustração, perda de produtividade e tempo. Durante o desenvolvimento, o programador mantém uma linha de raciocínio bem específica, voltada para o problema à sua frente. Se esta linha for quebrada, é necessário depois mais um tempo para que o desenvolvedor se adeque novamente àquela linha, diminuindo sua eficiência. Uma compilação que gasta cerca de uma hora para ficar pronta é uma maneira certa de se quebrar este raciocínio.

No entanto, estudos mostram que em um ambiente normal de desenvolvimento de software menos de 5% do potencial computacional é aproveitado (INCREDIBUILD, 2001): o desenvolvedor trabalha a maior parte do seu tempo com o código fonte através de um editor de texto, que exige no geral poucos recursos computacionais.

Este trabalho busca então criar uma ferramenta que distribua o trabalho de compilação entre as máquinas do ambiente de trabalho, de maneira a aproveitar esse recurso computacional desperdiçado. Desta maneira, deseja-se que o tempo de compilação seja reduzido sensivelmente, melhorando assim a produtividade dos desenvolvedores. Mesmo que o sistema obtenha uma modesta melhora, como reduzir o tempo de compilação de uma biblioteca de 10 para 7 minutos, pode-se ter ganhos excepcionais a longo prazo; deve-se considerar que apesar de ser uma pouca diferença, uma biblioteca é compilada normalmente dezenas de vezes por dia, resultando em um considerável tempo poupado ao final de um dia de trabalho.

## 2.6 Soluções Existentes

Existem diversas soluções disponíveis na internet que, assim como este trabalho, buscam diminuir o tempo de compilação de projetos escritos em C e C++. Abaixo são citados alguns que são de conhecimento do autor, através de pesquisa:

### 2.6.1 distcc

O *distcc* (POOL, 2002), é um programa que distribui a compilação de código C, C++, Objective C ou Objective C++ através de várias máquinas em uma rede local. É de simples instalação e uso, e é frequentemente duas vezes ou mais rápido que uma compilação local.

*distcc* não é em si um compilador, mas é um *front-end* ao compilador *GNU C/C++* (GNU, ). Quase todas opções e características do compilador funcionam normalmente. O sistema é designado para trabalhar com a opção de build paralelo presente em sistemas de compilação como por exemplo*make* (GNU, 2002) e *SCons* (SCONS, 2004).

Na figura 3 é apresentado o *distcc-monitor*, uma ferramenta para acompanhamento do progresso de compilação quando se utiliza o *distcc*.

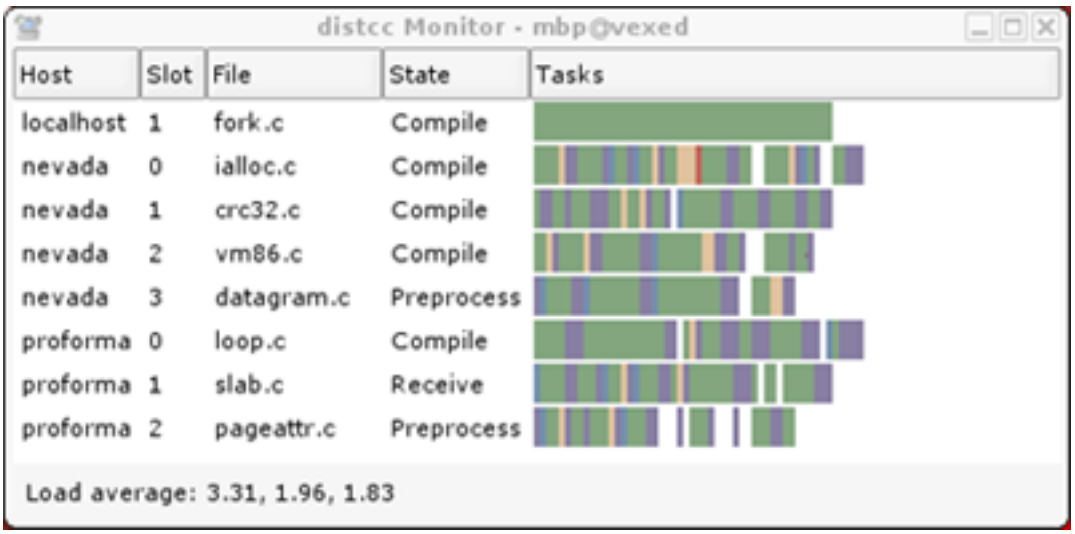


Figura 3: O distcc monitor acompanhando um processo de compilação

### 2.6.2 ccache

O projeto *ccache* (TRIDGELL, 2002) visa melhorar a velocidade de compilação usando uma *cache* para compiladores C/C++, evitando que um arquivo seja re-compilado desnecessariamente. De acordo com o site oficial, isto resulta em uma melhora de 5 a 10 vezes em compilações comuns.

Foi concebido inicialmente para auxiliar na compilação do projeto Samba, e vêm sido utilizado com sucesso em outros projetos. Pode ser combinado com o *distcc* para melhorar ainda mais a eficiência do mesmo.

### 2.6.3 IncrediBuild

A ferramenta *IncrediBuild* (INCREDIBUILD, 2001), desenvolvida pela *Xoreax Software*, tem propósito similar ao *distcc*, mas focando no uso do *Microsoft Visual C++*, o compilador C++ da Microsoft; ele provém integração direta com o ambiente de desenvolvimento do Visual Studio, sendo usado de maneira muito natural pelo desenvolvedor. Uma desvantagem do sistema é que ele não é gratuito, sendo necessário obter uma licença para cada máquina em que se deseja utilizá-lo. A figura 4 mostra a interface de acompanhamento de builds, integrada ao ambiente de desenvolvimento.

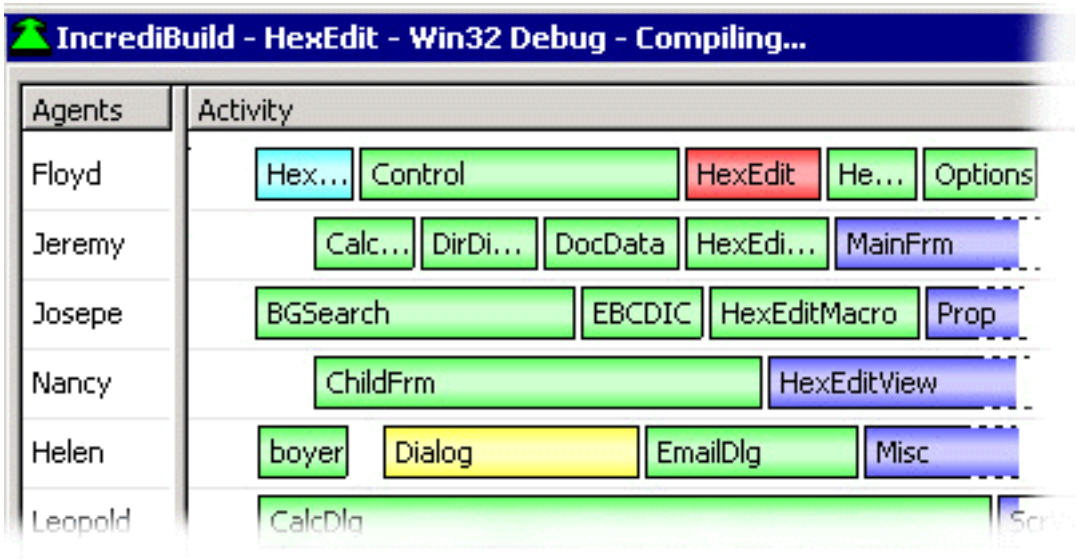


Figura 4: Interface do software IncrediBuild

### 2.6.4 DMS

O projeto *DMS - Distributed Make System* (DMS..., 2004) é também designado para distribuir trabalhos de compilação através de uma rede local. Desenvolvido por estudantes como um projeto escolar, deve rodar nos sistemas operacionais FreeBSD, OpenBSD, NetBSD, Solaris, MacOS X e Digital Unix, e é programado na linguagem de programação *Python* (ROSSUM, 2004). Assim como *distcc*, ele tem como alvo o compilador *GNU C/C++*, sendo portanto similar ao projeto *distcc*.

Em fase inicial de desenvolvimento no momento de escrita deste trabalho, ainda não liberou nenhum arquivo, e não há uma previsão para um lançamento no futuro próximo.

### 2.6.5 TeamBuilder

Desenvolvido pela empresa *Trolltech*, criadores da biblioteca de interface gráfica *Qt*, o *Teambuilder* (TROLLTECH, ) segue filosofia similar ao *distcc* e *Incredibuild*, distribuindo os trabalhos de compilação pela rede local para outras máquinas. O sistema operacional alvo é Linux, e ao contrário do *distcc*, não é limitado somente ao compilador *GNU C/C++*. Assim como o *Incredibuild*, uma licença é necessária para usar o produto.

# 3 Projeto

Como foi descrito no capítulo 2, o tempo de compilação causa um grande impacto na produtividade dos desenvolvedores em um projeto de grande porte.

Este trabalho visa desenvolver uma ferramenta para auxiliar equipes que fazem uso da linguagem C e C++ para o desenvolvimento, visando diminuir consideravelmente o tempo de compilação necessário, reduzindo assim custos e aumentando a produtividade.

## 3.1 Descrição do Sistema

Figura 5: Uma hydra luta ferozmente contra um guerreiro

O projeto busca a melhora no tempo de compilação pela divisão da compilação dos vários arquivos que compõem um projeto C/C++ entre as máquinas que compõem um ambiente de trabalho, através da rede local. Ele foi batizado de **Hydra**, em alusão a

legendária besta de múltiplas cabeças (ver figura 5[1]).

A idéia geral é que os diversos arquivos que compõem um projeto sejam compilados não somente na máquina do usuário, mas em todas as máquinas do ambiente de trabalho. O usuário dispara o processo *Hydra*, que se comunica com um *Hydra Controller*, rodando em outra máquina na rede; este então escolhe uma das máquinas que estão executando um *Hydra Server* e estão prontas para compilar, e devolve ao cliente o identificador de rede desta máquina. Então, o cliente envia o arquivo ao servidor, que o compila e retorna o código objeto, que é finalmente escrito no local apropriado no disco do usuário.

Durante a compilação remota de um arquivo por um servidor, o cliente fica ocioso, sem consumir recursos da CPU. Como são disparados vários processos *Hydra* simultaneamente, consegue-se paralelizar a compilação entre as máquinas da rede. A figura 6 demonstra uma visão geral do sistema.

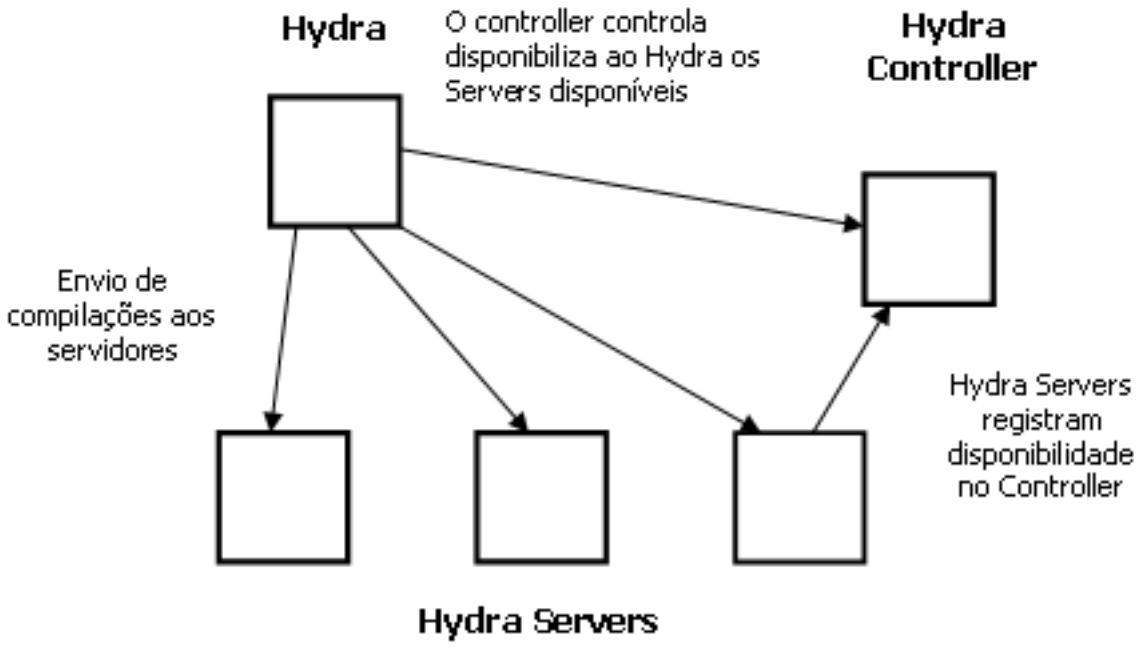


Figura 6: Visão Geral do Sistema

Nas próximas seções, os elementos do sistema e a interação entre eles é explicada em maiores detalhes.

[1] © 1995-2003 Wizards of the Coast, Inc., a subsidiary of Hasbro, Inc. All Rights Reserved

### 3.1.1 Hydra

A máquina do desenvolvedor de software é normalmente um desktop ou workstation conectado às outras máquinas do ambiente de trabalho através de uma rede local. Ele compila seus projetos utilizando um sistema de compilação (tais como *GNU Make* (GNU, 2002) ou *SCons* (SCONS, 2004)), que são encarregados de gerenciar as dependências, parâmetros de compilação, arquivos a serem compilados, etc. Este sistema então executa o compilador com os parâmetros apropriados uma vez para cada arquivo do projeto, gerando o *código objeto* em cada uma delas. O processo Hydra, encarregado de simular um compilador mas distribuindo o trabalho real de compilação entre outras máquinas da rede, é disparado a partir da máquina do usuário.

As seguintes mudanças devem ser feitas na máquina do usuário para o funcionamento do sistema:

1. Executar o processo *Hydra* ao invés do compilador nativo, com os mesmos parâmetros que passaria ao compilador normal. Como isto é feito depende de sistema operacional e de sistema de *builds*, como por exemplo um *symbolic link* no Linux, ou passando na linha de comando a opção `"-CC=hydra"` se o usuário utiliza o sistema de compilação *GNU Make*.

2. Configurar o sistema de compilação para executar mais de um trabalho ao mesmo tempo. A maioria dos ambientes de compilação possui uma opção de executar mais de um trabalho de compilação simultaneamente; isso permite que máquinas com mais de um processador façam uso total dos mesmos, diminuindo o tempo de compilação consideravelmente. Esta opção permite que o Hydra distribua o trabalho de compilação entre os outros computadores da rede local, se comportando como se fosse um compilador real.

### 3.1.2 Hydra Server

O Hydra Server é o componente do sistema que fica instalado nas outras máquinas da área de trabalho. Ele é capaz de compilar um arquivo fonte na máquina em que está instalado, usando para isto um compilador igual àquele que se encontra na máquina do usuário.

Quando o Hydra Server está em estado ocioso, ele se registra periodicamente no *Hydra Controller*, indicando assim que está pronto para compilar um arquivo remoto, além de

informar informações sobre sua configuração e estado atual, como quantidade de memória disponível, CPU, uso atual da CPU, etc; estes dados vão ser usados pelo *Controller* para decidir qual a máquina mais apta para determinado trabalho de compilação.

Quando um usuário do ambiente executa o Hydra, este consulta o *Controller* requisitando um *Server* disponível. O *Hydra* então pre-processa o arquivo na máquina do usuário, e o envia através da rede local para o *Server*. Este, por sua vez, compila o arquivo localmente e retorna o código objeto resultante. O trabalho de pré-processamento feito pelo cliente garente que o arquivo fonte esteja pronto para ser compilado, sem depender de quaisquer outros arquivos.

Um fator importante que deve ser lembrado é que as máquinas em que o *Server* roda estão sendo utilizadas por outros desenvolvedores. Não é interessante que o trabalho de um desenvolvedor seja pertubado devido à uma compilação de outro servidor; isto causaria uma perda de produtividade que vai contra o objetivo do trabalho. Como solução, o processo de compilação do servidor é disparado com a mais baixa prioridade do sistema operacional, garantindo assim que o trabalho na máquina não seja perturbado: a compilação remota só vai ser executada quando o sistema estiver ocioso. Por questão de restrição de tempo, essa solução só foi implementada para o sistema operacional Windows; suporte para esta característica a outros sistemas operacionais será adicionada no futuro.

### 3.1.3 Hydra Controller

Dentro da rede local onde o sistema vai ser utilizado deve haver pelo menos um *Hydra Controller* instalado em um dos computadores. Ele é responsável por conhecer todas as máquinas que possuem um *Hydra Server* instalado, e ainda fornecer essa informação aos processos *Hydra* que requisitarem uma compilação. Tanto os servidores quanto os clientes automaticamente localizam o *Controller*, através de um mecanismo de *broadcast*.

O *Controller* possui um suporte para a fácil implementação de algoritmos que permitem selecionar a melhor máquina para o trabalho a ser realizado baseado nas características da mesma, tais como velocidade e utilização da CPU, memória RAM, etc. Por questão de restrição de tempo, nenhum algoritmo deste tipo foi implementado; somente uma implementação simples que escolhe uma máquina aleatoriamente dentre as disponíveis. Algoritmos mais elaborados vão ser elaborados no futuro ((DANTAS; ZALUSKA, 1998)).

### 3.1.4 Configuração

O sistema é configurado através de um arquivo que reside na máquina cliente, chamado de `.hydrarc`, onde são definidas as características dos compiladores suportados pelo sistema. Cada compilador deve ter algumas de suas opções descritas para o *Hydra*, que faz uso das mesmas nas diferentes etapas de compilação. As opções a serem configuradas são as seguintes:

- Executável: o nome do arquivo executável do compilador. Como por exemplo, `"cl"` no Microsoft Visual C++ ou `"g++"` para o compilador C++ da GNU.
- Marcas de Opção: uma lista de caracteres que o compilador utiliza para marcar suas opções na linha de comando. Por exemplo, o Microsoft Visual C++ utiliza os caracteres `"/"` ou `"-"`, enquanto o GNU GCC permite somente o caracter `"-"`.
- Opção de Saída: indica ao compilador o nome do arquivo de saída. Por exemplo, no compilador Microsoft Visual C++ a opção é `"/Fo"` seguida do nome do arquivo, enquanto no GNU C++ é `"-o"` ou `"--output="`.
- Opção de Entrada: indica ao compilador o nome do arquivo fonte. Usualmente nenhuma opção é necessária, sendo o arquivo fonte o último parâmetro da linha de comando; no entanto certos compiladores possuem uma opção que indica explicitamente o arquivo, como pro exemplo `"/Tp"` no Microsoft Visual C++.
- Opção de Pré-Processamento: indica que o compilador deve somente pré-processar o arquivo, sem gerar nenhum código. O arquivo pré-processado é então jogado na saída (`stdout`, ou *standard output*). Por exemplo, a opção utilizada tanto pelo Microsoft Visual C++ e GNU C++ é `"-E"`.

Somente com estas opções, o *Hydra* é capaz de lidar com qualquer compilador, tornando o sistema facilmente extensível. A título de ilustração, a configuração do Microsoft Visual C++ ficaria assim no arquivo `.hydrarc`:

```
[Microsoft Visual C++ 2003.NET]
executable    = cl.exe
option-marks = -,/
output-flags = /Fo
preproc-flag = /E
source-flags = /Tp
```

Por restrição de tempo, a leitura deste arquivo não foi implementada; por enquanto o compilador utilizado é fixo dependendo do sistema operacional, sendo utilizado no Windows o Microsoft Visual C++, e o no Linux o GNU GCC. Implementação da leitura do arquivo de configuração é trivial e será implementada no futuro.

### 3.1.5 Diferenciais

O sistema conta com algumas características que o diferencia de outras soluções existentes:

- Multiplataforma: o sistema deve rodar em diversos sistemas operacionais, incluindo Windows e Linux. O requerimento necessário é basicamente que o sistema operacional seja suportado pelo interpretador Python, já que nenhuma outra ferramenta externa é requerida.
- Independência de Compilador: deverá funcionar com qualquer compilador, desde que suporte a execução do passo de pré-processamento independente da compilação completa. Novos compiladores podem ser facilmente adicionados à lista dos já suportados através do arquivo de configuração.
- Independência de Sistema de Build: funciona com qualquer sistema de build que suporte execução simultânea de múltiplos trabalhos. Tanto o *GNU Make* quanto o *SCons* suportam esta opção.
- Código Aberto: o sistema estará disponível na internet como um projeto de código aberto, podendo ser usado gratuitamente e sem restrições. A página se encontra no endereço `http://sourceforge.net/projects/hydrasystem` no momento da escrita deste documento.

Um outro detalhe interessante é que o Hydra suporta o processo de *cross-compiling*, que é utilizar um compilador que gera código para outra máquina ou sistema operacional. Apesar de o sistema não ter sido designado para isto, basta que um *cross-compiler* com o mesmo nome do compilador que foi configurado esteja instalado no ambiente do usuário.

## 3.2 Ferramentas de Desenvolvimento Adotadas

### 3.2.1 Python

O sistema foi implementado na linguagem de programação *Python* (ROSSUM, 2004); ela é uma linguagem de alto-nível, interpretada, interativa e orientada a objetos. Ela é comumente comparada a *Tcl*, *Perl*, *Scheme* e *Java*.

Esta linguagem foi escolhida pelos seguintes motivos:

- A implementação do interpretador é extremamente portável, rodando em diversos tipos de UNIX, Windows, OS/2, Amiga, entre outras.
- Conta com diversos pacotes de comunicação de rede, simplificando a implementação do sistema.
- Permite um desenvolvimento ágil e produtivo.
- O autor está bastante familizarizado com a mesma.

Foi utilizado também a biblioteca *Pyro* (JONG, 2003): ela é um pacote de Objetos Distribuídos escrito totalmente em Python, sendo concebido para fácil utilização; ele gerencia toda a comunicação da rede entre os objetos do usuário, que não precisa se preocupar com o fato de seus objetos estarem distribuídos, tratando-os como se fossem objetos locais normais.

Python, associado com o pacote Pyro, permitiram que o sistema fosse desenvolvido em cerca de menos de 30 horas, ao longo de cerca de um mês e meio.

### 3.2.2 Metodologia: XP

*Extreme Programming*, ou *XP*, é uma metodologia de desenvolvimento de software baseada em comunicação, simplicidade e retorno de informação. Busca fazer um time de desenvolvimento trabalhar junto baseado em práticas simples, de maneira a obter informações para perceber onde o projeto se encontra exatamente e permitir melhorar suas metodologias de maneira a melhor se adequar à situação atual (JEFFRIES, 2004).

XP não é um conjunto de regras de como os projetos devem ser executados, sendo mais um apanhado de práticas que buscam prover qualidade e flexibilidade ao desenvolvimento de um sistema. Dentre estas práticas, uma das principais é a prática do *Unit*

*Testing*, que permite que o software evolua de maneira sustentável; é comum sistemas de grande porte não suportarem adição de novas funcionalidades de maneira simples, pois esta nova funcionalidade constuma quebrar outras. Com a prática do *Unit Testing*, testes automatizados vão sendo escritos à medida que o software vai sendo desenvolvido, garantindo assim que as novas funcionalidades realmente funcionem da maneira esperada. Como estes testes são automatizados, eles vão ser sempre executados, garantindo que o sistema como um todo continue funcionando e que qualquer quebra de funcionalidade seja detectada o mais rápido possível.

A qualidade de um sistema de software pode ser medida de várias formas, entre elas sua estabilidade, manutenção, flexibilidade à mudanças e performance.

No desenvolvimento do *Hydra*, foi adotada a prática XP de Unit-Testing, procurando obter certificação da estabilidade e qualidade do sistema e garantindo que futuras modificações não gerem problemas em outras partes; ele conta com cerca de 10 arquivos de testes automatizados, cobrindo todas as funcionalidades do sistema.

Diversos testes foram executados em bibliotecas reais escritas em C++, com o objetivo de se medir a performance do *Hydra* e detectar possíveis problemas. Esses testes e resultados são mostrados em detalhes no Capítulo 4, na página 31.

### 3.2.3 Editor/IDE

Como IDE para o desenvolvimento do *Hydra* foi utilizado o *Eclipse* (FOUNDATION, 2004), que é uma plataforma universal e extensiva para ferramentas de desenvolvimento, contando com editores, gerenciadores de projetos, integração com sistemas de controle de versão, entre muitos outros.

Como a linguagem utilizada para o desenvolvimento foi Python, foi utilizado a extensão *PyDev* (ZADROZNY, 2004), que adiciona diversas características que tornam o Eclipse um excelente ambiente de trabalho para esta linguagem, como completação automática de código, verificação de cobertura do código por parte de testes, ferramentas de *refactoring*, depurador, entre outras.

### 3.2.4 Hardware

Como laboratório de desenvolvimento e testes, foi utilizada a sede da ESSS, na seção de desenvolvimento de software. O ambiente conta com 8 computadores conectados em

uma rede local Ethernet de 100 MBps, sendo eles equipados com processadores AMD Athlon de 1400 MHz a 2000 MHz e entre 512MB e 2GB de memória RAM.

### 3.2.5 Sistemas Operacionais

O sistema foi totalmente desenvolvido e testado no sistema operacional Windows XP, utilizando o compilador C++ do *Microsoft Visual C++ .NET 2003*.

## 3.3 Código Fonte

A arquitetura do sistema, suas classes e suas relações, não são apresentadas neste trabalho de maneira descritiva. Caso seja de interesse do leitor conhecer melhor o funcionamento interno do sistema, o próprio *Python* conta com uma ferramenta que permite a fácil geração de um manual para estudo, o *pydoc*, que acompanha a distribuição oficial. Outra ferramenta que também pode ser usada é o *epydoc* (LOPER, 2004), que pode ser obtida na internet gratuitamente. O código fonte completo se encontra disponível no Anexo A, na página 47.

# 4 Resultados

## 4.1 Resultados Desejados

Com a utilização do sistema, deseja-se uma diminuição significativa no tempo de compilação, sendo proporcional ao número de máquinas que estejam executando o *Hydra Server*. Existem pontos no sistema que geram um custo extra quando comparado à compilação tradicional na máquina local, como por exemplo o processamento requerido pelo sistema, o envio de arquivos através da rede, a inicialização do interpretador Python, etc. Considerando isto, a eficiência do sistema não deve ser 100%, devendo ficar mas próxima de 80-90%, significando que com o uso de 10 máquinas no sistema, a velocidade de compilação aumente em cerca de 8 a 9 vezes.

## 4.2 Metodologia de Testes

Quando se busca testar um sistema em matéria de performance, várias variáveis que afetam o desempenho e o comportamento do mesmo devem ser consideradas e analisadas. Abaixo são descritas as que mais tem importância no sistema:

**Tempo de Comunicação** é o tempo gasto com o sistema com toda a comunicação de rede, desde a conexão entre os clientes e servidores, quanto o tempo gasto com a transferência dos arquivos entre as máquinas. Esta variável é afetada tanto pela velocidade da rede local quanto a performance da biblioteca utilizada para a comunicação de dados. Esta variável não foi considerada nos testes, devido à falta de acesso à uma rede de velocidade diferente à utilizada, que foi de 100 MBps.

**Processamento Extra** ao processo de compilação que antes não existia. Entre estes podem ser citados a leitura e interpretação da linha de comando usada até a escrita do arquivo pelos *Hydra Servers* em um diretório temporário para a compilação remota.

**Custo de Compilação dos Arquivos Individuais** da biblioteca compilada pode ser um fator chave no comportamento do sistema. Quanto maior o tempo gasto com o processo de compilação em si, menor a influência das outras variáveis.

**Número de Processos Simultâneos** disparados na máquina cliente pelo sistema de build. A máquina que executa o *Hydra* tem a responsabilidade de pré-processar os arquivos a serem compilados remotamente; isso pode gerar tempo perdido caso um *Hydra Server* seja obrigado a ficar ocioso aguardando que um novo pré-processamento seja realizado para que possa iniciar a compilação. É necessário achar um equilíbrio neste número, de maneira que o número de processos simultâneos seja suficiente para que uma máquina rodando o servidor possa começar a compilar um novo arquivo imediatamente, mas sem causar sobrecarga na máquina do cliente. Para os testes, o número mínimo de processos é igual ao número de máquinas contendo um *Hydra Server*, pois de outra forma o desempenho ótimo não seria possível de ser alcançado, e depois foi-se aumentando este número, até 2 vezes o número base.

**Heterogeniedade das Máquinas** na rede local. A performance deve ser afetada caso a o ambiente de trabalho seja muito heterogênio, com máquinas muito mais poderosas em termos de processamento que outras. Esta variável não foi estudada, devido ao ambiente relativamente homogênio do setor desenvolvimento da ESSS, onde foram realizados os testes.

Os testes foram realizados em duas bibliotecas desenvolvidas e utilizadas pela ESSS, *COI-Lib* e *Cyclope*. Foram feitas compilações desta bibliotecas em um número progressivo de máquinas, e comparado com o tempo gasto com uma compilação local comum. Foi também variado o número de processos simultâneos disparados para cada configuração do sistema, de maneira a se conseguir achar um número ótimo dado baseado no número de servidores na rede.

As características das máquinas utilizadas e seus respectivos nomes são as seguintes:

- *frodo*: processador AMD XP 1400+, com 1.5GB de memória RAM.
- *sentinel*: processador AMD XP 1400+, com 1 GB de memória RAM.
- *flash*: processador Pentium II 800 MHz, com 768 MB de memória RAM.
- *wolverine*: processador Pentium IV 2.8 GHz, com 1 GB de memória RAM.
- *superman*: processador AMD XP 1400+, com 2 GB de memória RAM.

Em todos os testes, a máquina cliente foi o *frodo*, e as outras máquinas se encontravam ociosas.

### 4.2.1 COI-Lib

A *COI-Lib*, biblioteca voltada para o processamento científico, foi desenvolvida internamente pela ESSS e provém algoritmos de visualização e processamento de dados, sendo implementada nas linguagens de programação C++ e Python. As partes da biblioteca que são críticas em matéria de performance foram implementadas em C++ e possuem uma interface para *Python*, permitindo que se utilize suas classes e funcões a partir desta linguagem. Esta interface entre as duas linguagens foi feita utilizando a biblioteca *Boost.Python* (DAWES, 1998). Para os testes, somente a parte codificada em C++ foi utilizada, já que o código Python é interpretado. Conta com cerca de 165 arquivos de código fonte (sem contar o número simular arquivos *headers*) e sua compilação, sem usar o sistema do Hydra, leva um total de **4 minutos e 20 segundos** no computador frodo.

Inicialmente compilou-se a COI-Lib utilizando-se o Hydra apenas uma máquina, com um *Server* e *Controller* rodando no frodo, e a compilação sendo iniciada a partir do mesmo. O objetivo desse teste é medir o *overhead* gasto com o sistema, isto é, o tempo extra que o sistema exige do processo de compilação.

O tempo gasto foi de 8 minutos e 13 segundos, um aumento de 45% em relação ao tempo original. Este overhead foi surpreendente a princípio e será analisado mais adiante. Continuando os testes, foi-se adicionando as máquinas restantes e variando o número de processos simultâneos em cada configuração. Os resultados podem ser vistos nas figuras 7, 8, 9 e 10.

Percebe-se que o sistema não é muito mais eficiente que uma compilação local, sendo em alguns casos até mais lento. Foi necessário que três máquinas fossem utilizadas pelo sistema até que ele conseguisse ser tão rápido quanto uma compilação comum, e mesmo assim depois sua eficiencia não melhorou muito. Na seção 4.2.1.1, a escalabilidade do sistema é analisada com maiores detalhes.

#### 4.2.1.1 Análise de Escalabilidade

Utilizando-se os dados obtidos na seção 4.2.1, podemos então analisar a escalabilidade do sistema de acordo com o número de máquinas utilizadas. Na figura 11, é apresentado um gráfico mostrando o tempo necessário para compilar a COI-Lib baseado no número

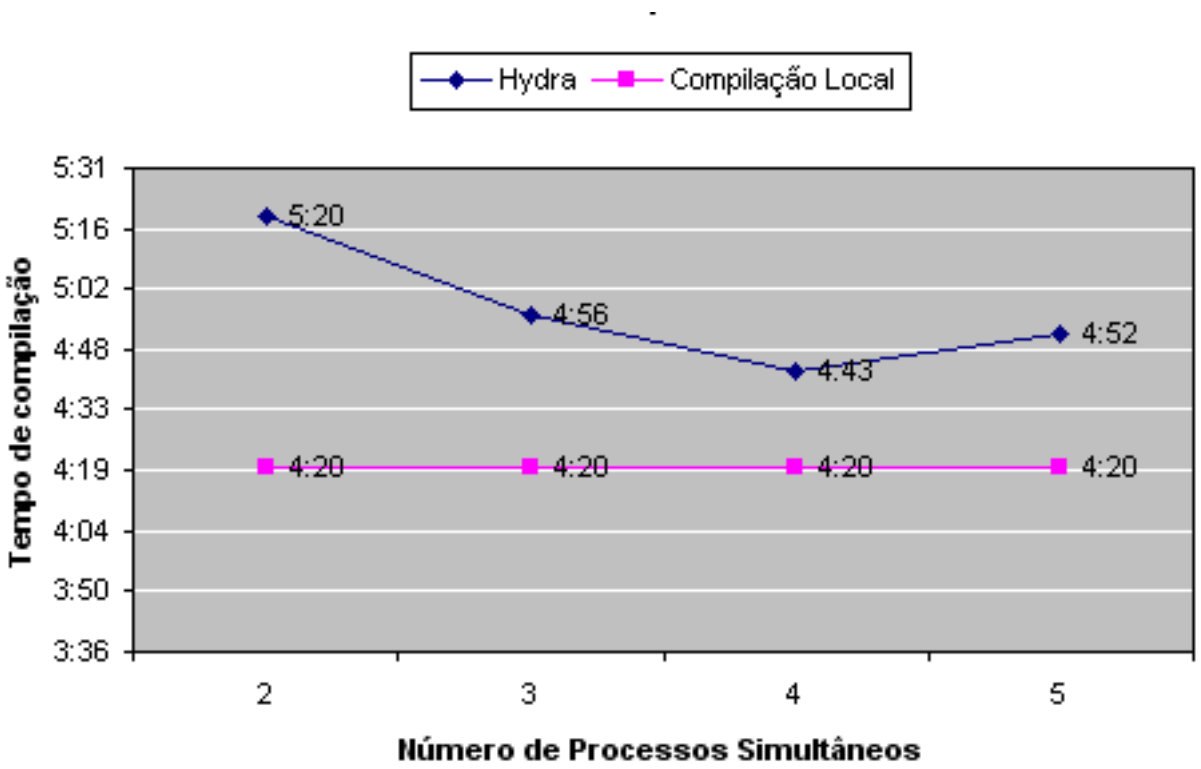


Figura 7: Compilação da COI-Lib utilizando-se duas máquinas

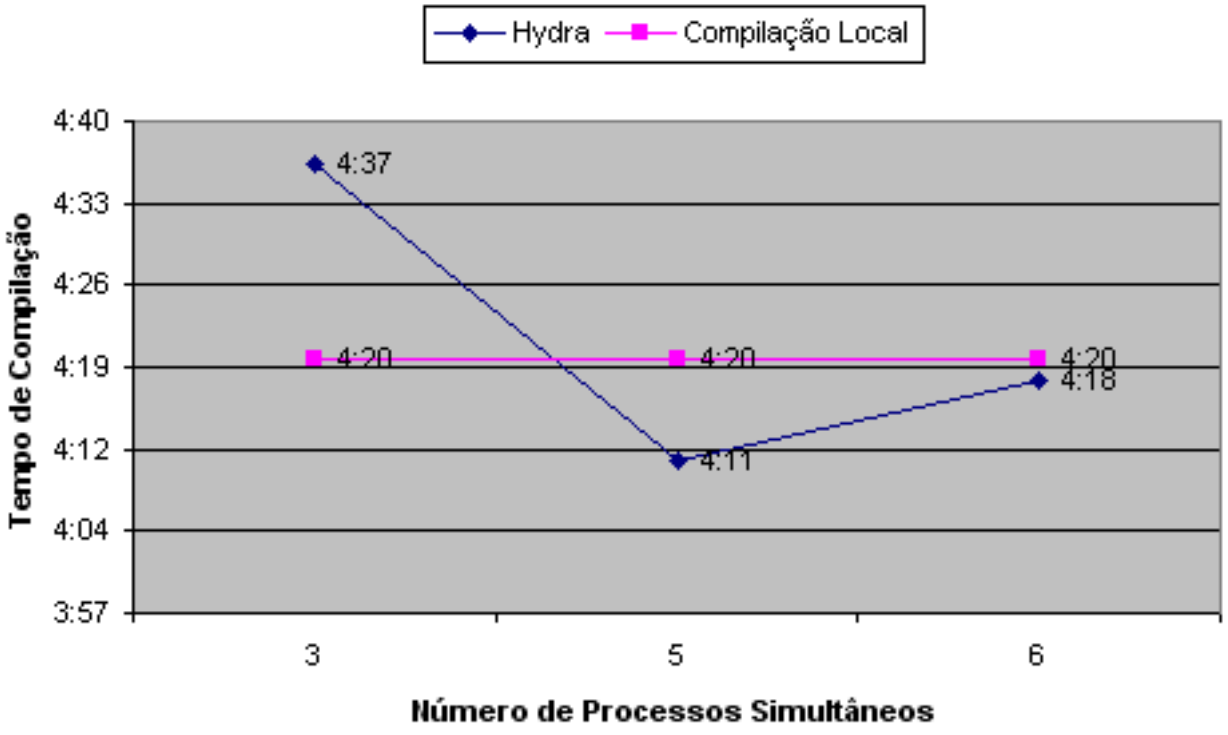


Figura 8: Compilação da COI-Lib utilizando-se três máquinas

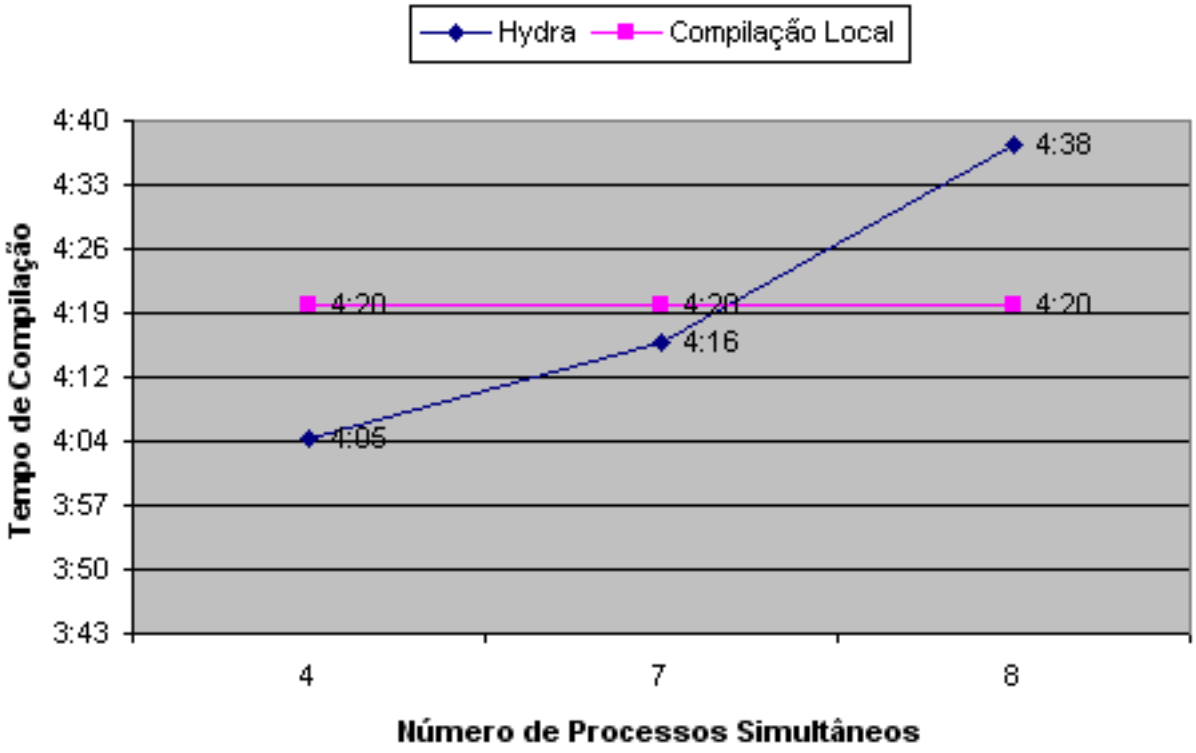


Figura 9: Compilação da COI-Lib utilizando-se quatro máquinas

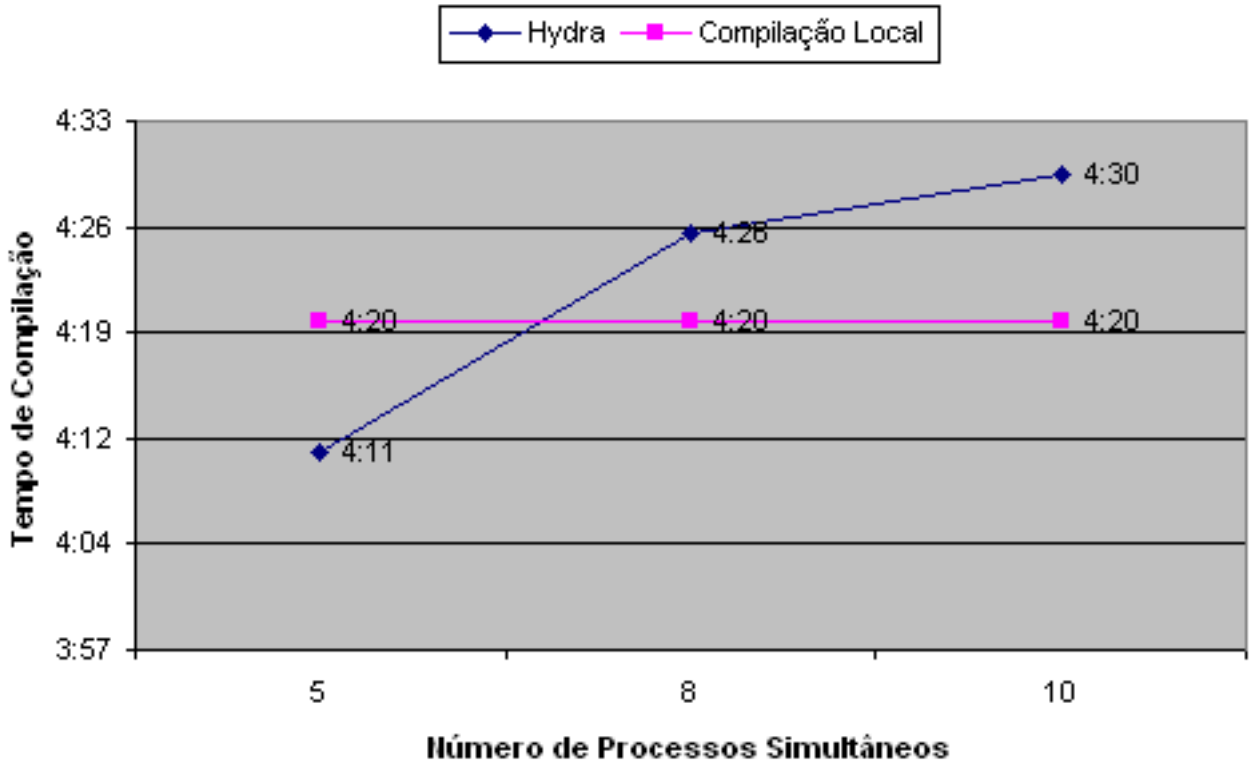


Figura 10: Compilação da COI-Lib utilizando-se cinco máquinas

de máquinas utilizados, sendo que o tempo é dado em número de vezes que o sistema é mais rápido que a compilação local. Para cada configuração, foi utilizado o menor tempo obtido na etapa anterior.

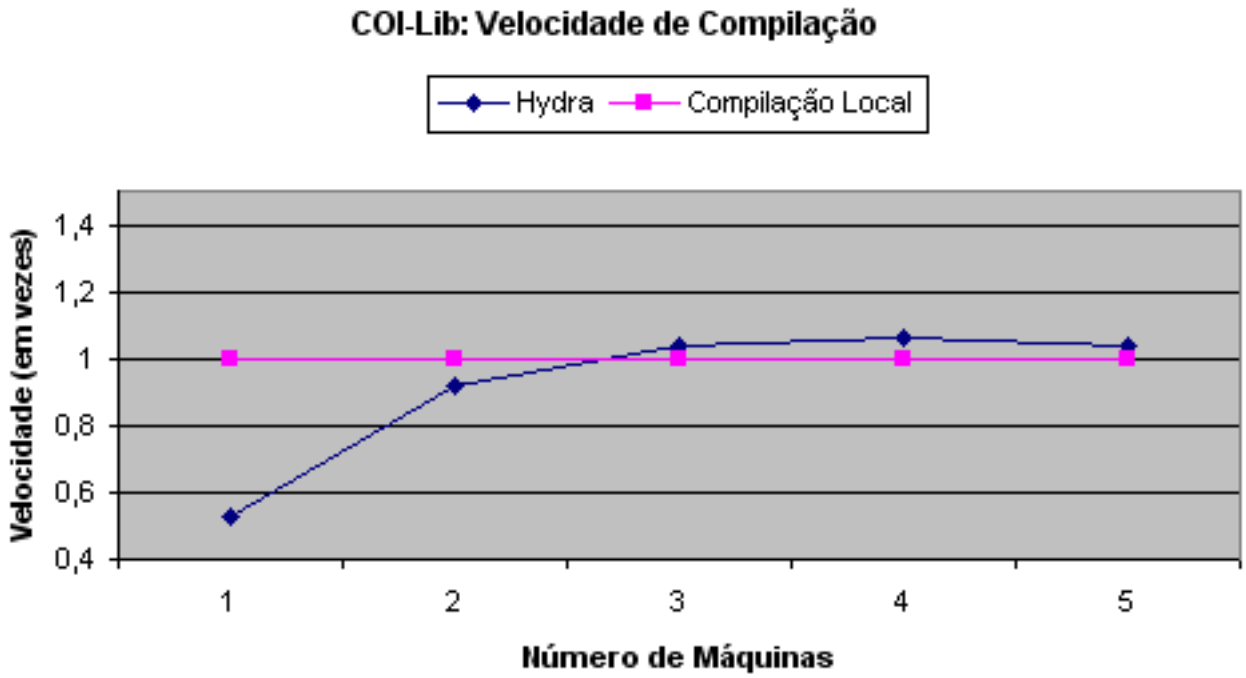


Figura 11: Escalonamento do Hydra de acordo com o número de máquinas na COI-Lib

O gráfico demonstra que o sistema escalona de forma que deixa muito a desejar. Utilizando-se apenas uma máquina, o sistema leva quase o dobro do tempo necessário (0.52 vezes mais rápido) para compilar a COI-Lib. Adicionando-se mais máquinas o sistema apresenta uma melhora, chegando a ser 1.06 vezes mais rápido que a compilação local com 4 máquinas, mas o ganho não justifica a utilização do sistema para a compilação desta biblioteca.

Para a compilação de cada arquivo, é necessário a execução de uma série de passos para que o *Hydra* compra seu papel. Podemos analisar e medir cada um deles individualmente para tentarmos explicar a ineficiência do sistema em compilar a COI-Lib.

**Inicialização** é o tempo gasto para inicializar o sistema, especificamente iniciar o interpretador Python e carregar as bibliotecas utilizadas.

**Pré-Processamento** é o tempo gasto pelo compilador para pré-processar o código fonte. Este arquivo pré-processado vai ser então enviado para outros *Servers* na rede para ser compilado.

**Compilação** foi medido como o tempo gasto pelo servidor remoto para compilar o arquivo, considerando somente o tempo gasto pelo processo de compilação.

**Comunicação de Rede** é o tempo gasto com toda a interação pela rede local, desde a comunicação com os *Controllers* e *Servers* até a transferência de arquivos pela rede.

**Finalização** é o tempo necessário para a finalização do interpretador Python, onde ele finaliza seus módulos, libera a memória utilizada, etc.

Analisando a compilação da COI-Lib para as variáveis descritas acima, obtemos o gráfico mostrado na figura 12, que indica a porcentagem do tempo total gasto em cada categoria.

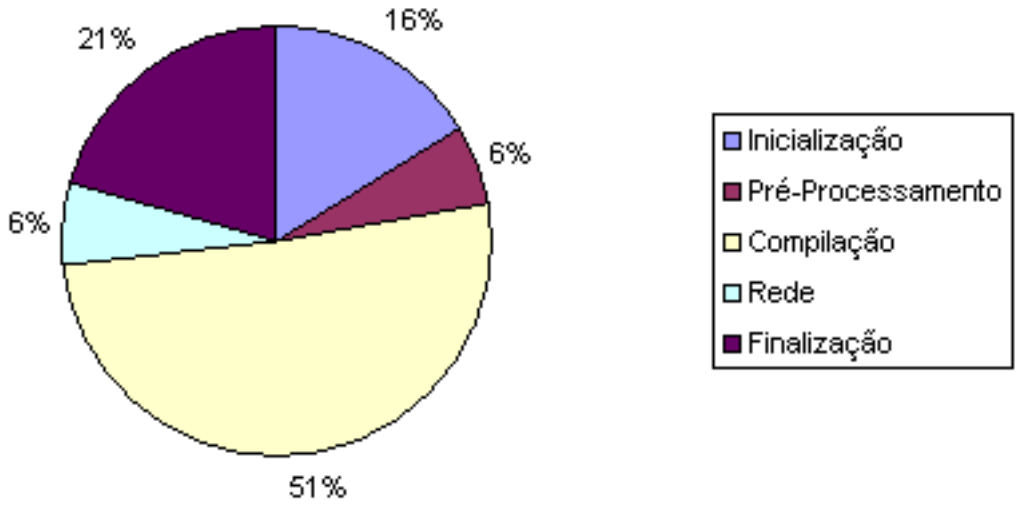


Figura 12: Porcentagem do tempo em cada etapa para COI-Lib

É interessante notar o grande *overhead* do sistema com relação à inicialização e finalização: mais de 35% do tempo total. Como pré-processar o arquivo e em seguida compilá-lo causa um overhead de somente 3% sobre o processo de compilação normal, podemos juntar as duas categorias em uma só para melhor visualizarmos os pontos fracos do sistema. O resultado é mostrado na figura 13.

Analisando a figura, nota-se que o grande problema do sistema neste caso é a inicialização e finalização do interpretador Python, resultando em um *overhead* de mais de 35% sobre o tempo total do Hydra.

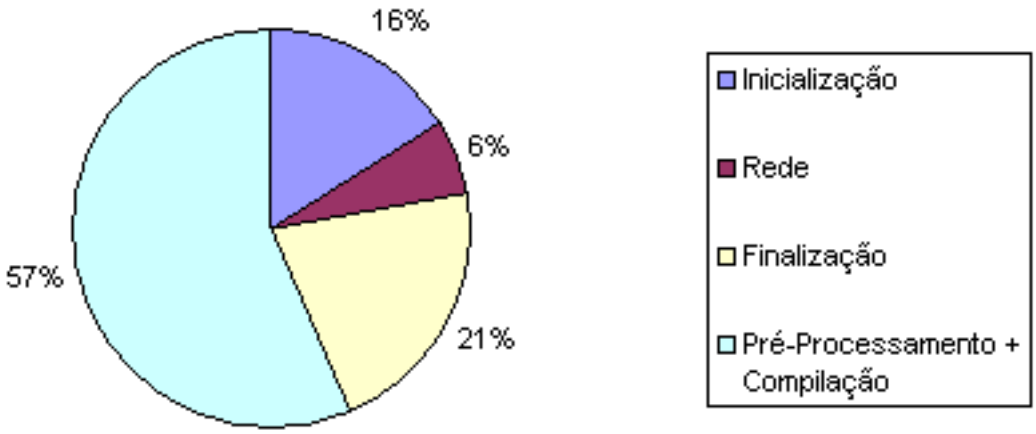


Figura 13: Porcentagem do tempo em cada etapa para COI-Lib

## 4.3 Cyclope

A biblioteca *Cyclope*, desenvolvida para a aplicação de mesmo nome, provém algoritmos e estruturas de dados voltadas para a análise e simulação de reservatórios de petróleo. Sua implementação faz forte uso de técnicas de *Template Metaprogramming* com o propósito de obter a melhor performance possível. Com o uso destas técnicas, é possível que chamadas de métodos sejam todas resolvidas em tempo de compilação, resultando em um ganho de performance bastante significativo: testes que comparam ela com soluções implementadas usando técnicas mais tradicionais demonstram que o Cyclope é de 2 a 10 vezes mais rápido em operações comuns de manipulação de dados e visualização. Este poder tem um preço, que é o aumento significativo do tempo e memória necessário para compilá-la.

O *Cyclope* é composto de cerca de 120 arquivos *headers*, onde a maioria da implementação está concentrada (uma exigência da técnica de *Template Metaprogramming*), e conta com apenas 24 arquivos fontes que são efetivamente compilados. Apesar do número muito menor de arquivos comparando-se com a *COI-Lib*, o tempo de compilação do sistema é muito maior: **20 minutos e 16 segundos** no *frodo*, com cada arquivo necessitando em média 50 segundos, em contraste com os cerca de 2 segundos por arquivo em média da COI-Lib.

Assim como os testes da *COI-Lib*, compilou-se o Cyclope utilizando-se o Hydra apenas em uma máquina, com um *Server* e *Controller* rodando no frodo, e a compilação sendo iniciada a partir do mesmo. O tempo gasto desta vez foi de 23 minutos e 12 segundos, um *overhead* de 12% em relação ao tempo original gasto pela compilação local. Seguindo a mesma filosofia de testes da COI-Lib, os tempos obtidos são apresentados nas figuras 14, 15, 16 e 17.

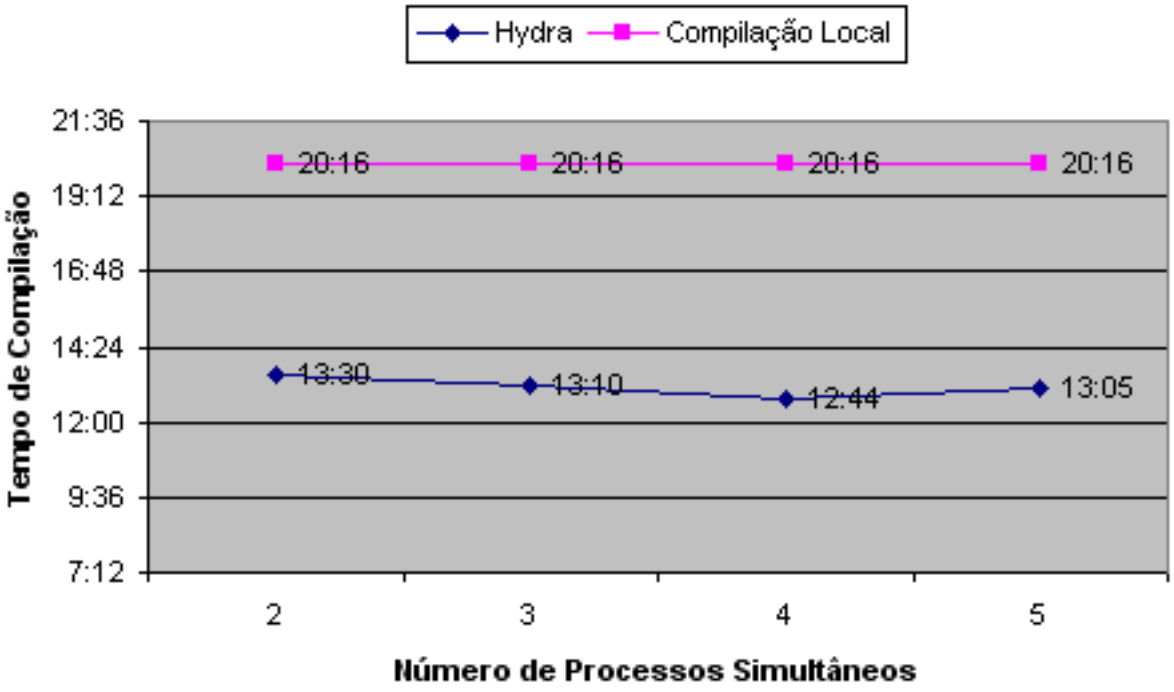


Figura 14: Compilação do Cyclope utilizando-se duas máquinas

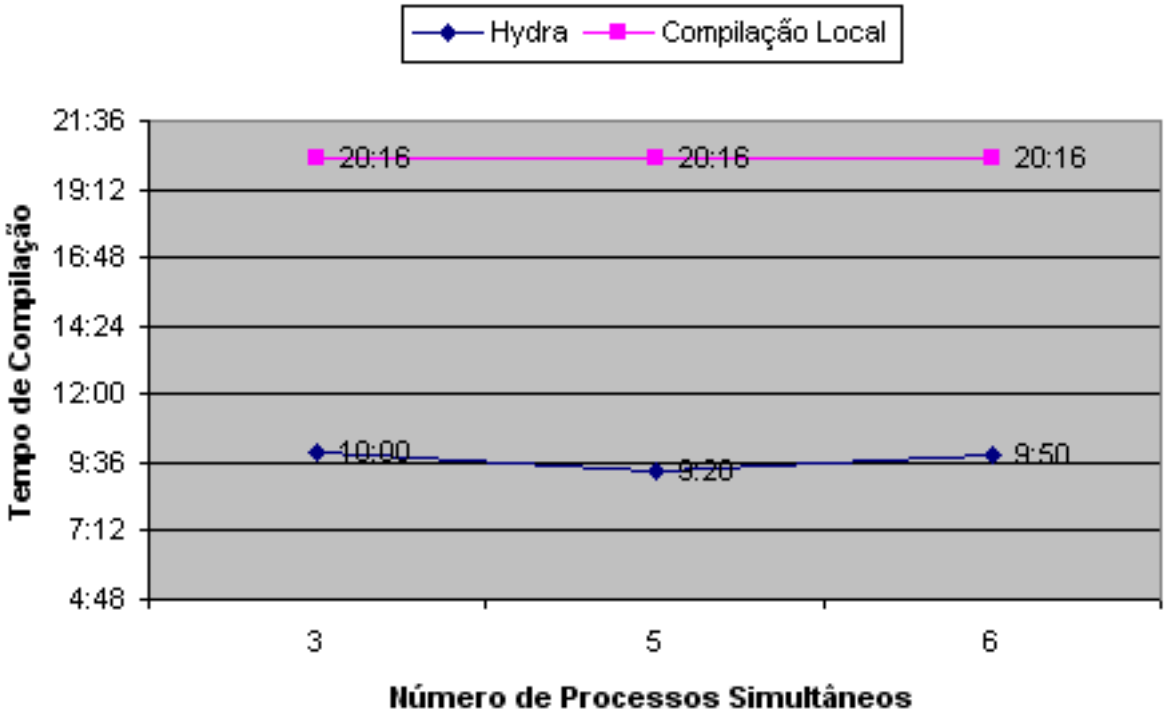


Figura 15: Compilação do Cyclope utilizando-se três máquinas

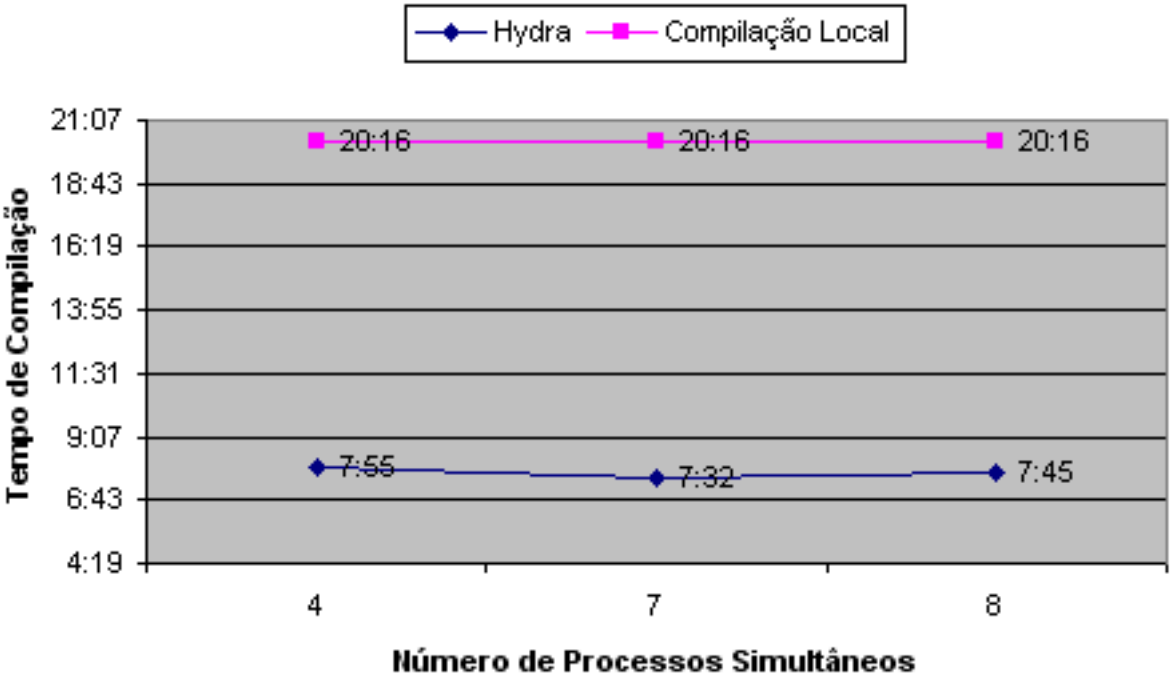


Figura 16: Compilação do Cyclope utilizando-se quatro máquinas

Com os dados dos testes, percebe-se que o sistema se comporta muito melhor com a biblioteca Cyclope do que a biblioteca COI-Lib, conseguindo uma redução de até 70% no

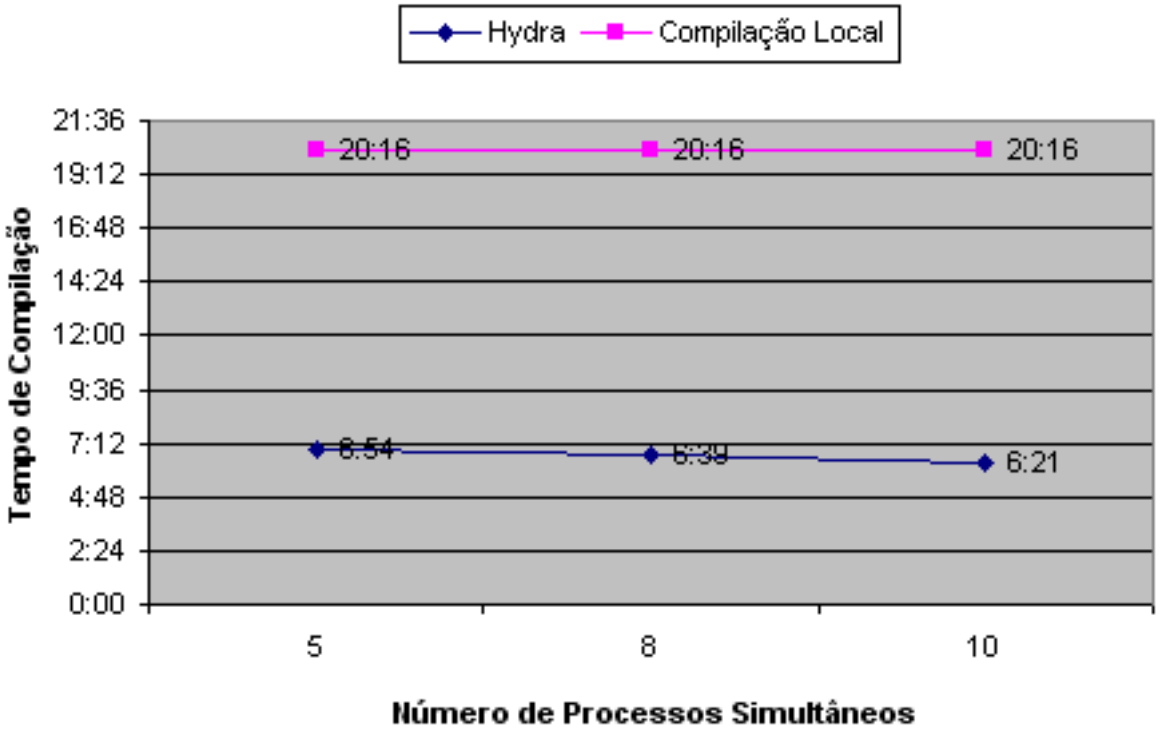


Figura 17: Compilação do Cyclope utilizando-se cinco máquinas

tempo de compilação do sistema.

#### 4.3.0.2 Análise de Escalabilidade

Com os testes da biblioteca do *Cyclope*, percebeu-se que o sistema escalona muito melhor quando utilizado em uma biblioteca em que o custo de compilação dos arquivos individuais é alto comparado ao *overhead* causado pelo Hydra. O gráfico de escalonamento baseado no número de máquinas é apresentado na figura 18.

Analisando a compilação do Cyclope para as variáveis descritas na seção 4.2.1.1, obtemos o gráfico apresentado na figura 19. Percebe-se que o *overhead* do sistema é muito mais baixo que no caso da COI-Lib, explicando assim a sua performance muito superior para este caso.

Juntando-se a categoria de Pré-Processamento com a de Compilação, verifica-se que o principal *overhead* do sistema está no tráfego de arquivos pela rede, como pode ser visualizado na figura 20.

Foi feito também um *profiling* do sistema, onde foi verificado o tempo gasto por cada método, com o objetivo de detectar pontos que podem ser melhorados em matéria de

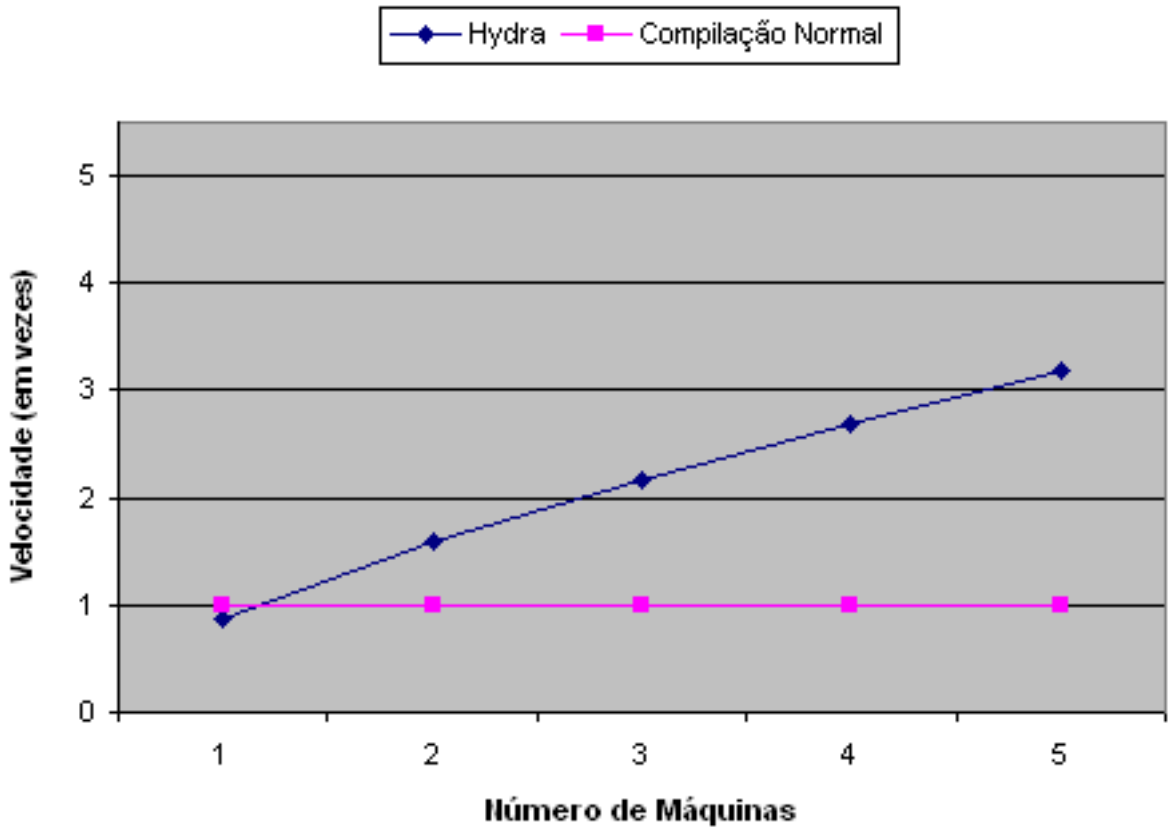


Figura 18: Escalonamento do Hydra de acordo com o número de máquinas no Cyclope

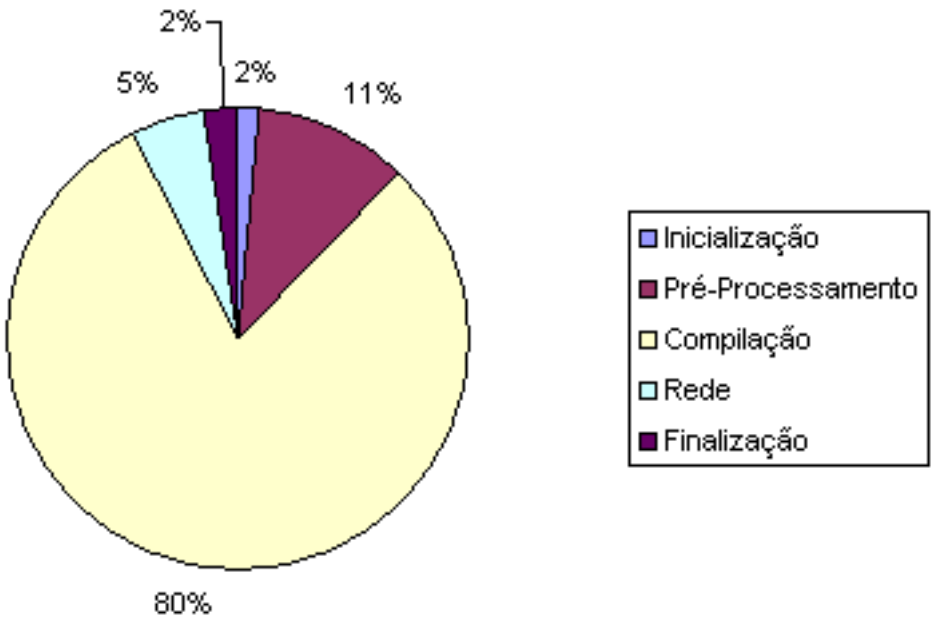


Figura 19: Porcentagem do tempo em cada etapa para o Cyclope

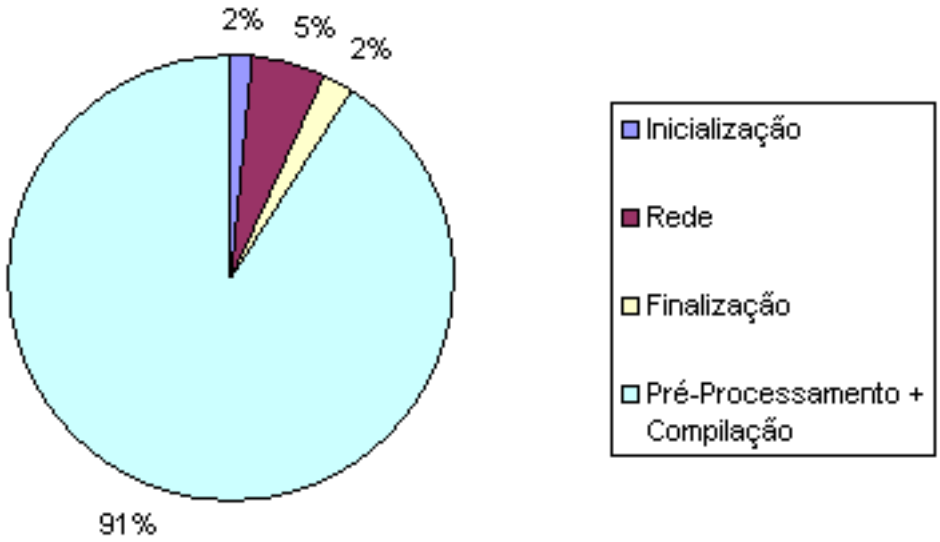


Figura 20: Porcentagem do tempo em cada etapa para o Cyclope

velocidade. Contudo, não se detectou nenhum método ou função que tivesse um tempo de execução significativo.

Considerando os casos da COI-Lib e Cyclope, percebe-se que atualmente o Hydra possui um *overhead* muito grande, principalmente no que se diz à inicialização e finalização do sistema; o interpretador Python leva cerca de meio segundo para completar sua inicialização e um pouco mais de meio segundo para finalizar, tempo que normalmente é insignificante para a maioria das aplicações, mas se torna um problema quando o interpretador tem que ser iniciado dezenas de vezes por minuto para realizar um trabalho que consome apenas dois segundos para ser completado. Este tempo vêm da natureza extremamente dinâmica da linguagem, onde tudo é realizado em tempo de execução, até mesmo as declarações de classe, além do grande número de módulos e funcionalidades que acompanham o intepretador. Deve ficar claro, no entanto, que o uso desta linguagem permitiu um tempo de desenvolvimento muito mais rápido do que aquele que seria obtido se fosse utilizada uma linguagem que gerasse uma aplicação mais leve em termpos de inicialiazação, como por exemplo o próprio C++.

# 5 Conclusões e Trabalhos Futuros

Com os resultados obtidos, conclui-se que o sistema do *Hydra* funciona bem para bibliotecas formadas por arquivos que necessitam de um tempo relativamente alto para serem compilados individualmente; em bibliotecas em que o tempo requerido por cada arquivo é baixo, o sistema acaba não conseguindo melhorar o tempo de compilação de maneira significativa, sendo de pouca utilidade.

Como foi verificado que o principal problema é o tempo de inicialização e finalização, certas mudanças podem ser feitas com o objetivo de melhorar a velocidade do sistema. Um programa escrito em C++, visando ser o mais leve possível, substituiria o atual executável do *Hydra*, que serve de gatilho para o sistema. Este novo programa se comunicaria com o *Hydra Server* rodando na máquina local e enviaria a linha de comando do trabalho a ser realizado. O servidor então faria o trabalho que agora é realizado pelo executável *Hydra*, se comunicando com o controller, pré-processando os arquivos fontes, enviando o mesmo para um outro servidor e por fim escrevendo os resultados na máquina local. Como desta maneira o interpretador Python não vai ser iniciado uma vez para cada arquivo do sistema, o problema do tempo de inicialização deixa de existir. Com isto, espera-se que a performance do sistema aumente consideravelmente.

Outras mudanças que são planejadas para uma futura implementação são as seguintes:

- Checagem de Compilador: atualmente o sistema não verifica se o compilador que está nas máquinas remotas é exatamente o mesmo que está instalado na máquina cliente. Diferenças na versão do compilador podem gerar erros no código gerado difíceis de detectar, portanto uma solução deve ser procurada nesse sentido. Como a maioria dos compiladores possuem uma opção de linha de comando que faz com que ele escreva na saída sua versão, essa informação pode ser comparada com a saída dos outros compiladores instalados, permitindo que diferenças de versão sejam facilmente detectadas.

- Transferência Automática de Compilador: atualmente, quando um *Server* não pos-

sui um compilador do tipo requerido pelo cliente, ele reporta um erro, e o cliente então ignora aquele servidor e tenta a comunicação com outro. Uma capacidade interessante de ser adicionada ao sistema seria a de enviar automaticamente o compilador para o *Server*, que o usaria então para compilar os arquivos do cliente. Isso permitiria que o Hydra fosse instalado em máquinas sem nenhum compilador instalado, se encarregando automaticamente de transmitir o compilador necessário.

- Ferramenta de Monitoração: uma ferramenta de interface gráfica que se comunicaria com o *Controller* com o objetivo de fornecer ao usuário o estado atual do sistema, permitindo modificar prioridades, acompanhar o andamento das compilações, etc.

Uma idéia interessante é a de se expandir o escopo do Hydra de *Compilador Distribuído* para *Executor de Trabalhos Distribuídos*. Na arquitetura do sistema, nada impede de que o trabalho executado remotamente seja outra coisa que não uma compilação, mas por exemplo uma multiplicação de matrizes em um sistema de solução de equações distribuídas, processamento de dados de um grid de reservatório, enfim, qualquer trabalho custoso em termos computacionais e que possa ser paralelizado. Provavelmente seria necessário que a adição de novos tipos de trabalho tivesse que ser feita através de codificação (criar uma subclasse de uma das classes do Hydra, por exemplo), ao contrário de um simples arquivo de configuração, mas a arquitetura é genérica o suficiente para permitir isto.

Apesar das limitações acima apontadas, o sistema se mostrou estável e conseguiu compilar todos os testes de maneira correta. Aplicando-se as soluções aos seus problemas como foram descritas neste capítulo, o sistema deverá ser utilizável por qualquer ambiente de desenvolvimento para qualquer tipo de biblioteca, diminuindo consideravelmente o tempo de compilação e melhorando a produtividade e o ânimo dos desenvolvedores de software.

# *Referências*

DANTAS, M. A. R.; ZALUSKA, E. J. Efficient scheduling of mpi applications on networks of workstations. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., v. 13, n. 6, p. 489–499, 1998. ISSN 0167-739X.

DAWES, D. A. B. *Boost*. http://www.boost.org, 1998.

DMS - Distributed Make System. http://www.nongnu.org/dms/faq.html, 2004.

ESSS. *Cyclope*. http://www.cyclope.com.br, 2004.

ESSS. http://www.esss.com.br, 2004.

FOUNDATION, E. *Eclipse*. http://www.eclipse.org/org/index.html, 2004.

GNU. *GCC*. http://gcc.gnu.org.

GNU. *Make*. http://www.gnu.org/software/make, 2002.

IMAGO. http://www.imagosystem.com.br, 2004.

INCREDIBUILD. http://www.xoreax.com/main.htm, 2001.

JEFFRIES, R. E. *XP Programming*. http://www.xprogramming.com, 2004.

JONG, I. de. *Pyro*. http://pyro.sourceforge.net/, 2003.

LOPER, E. *epydoc*. http://epydoc.sourceforge.net, 2004.

POOL, M. *distcc*. http://distcc.samba.org/, 2002.

ROSSUM, G. van. *Python*. http://www.python.org/doc/essays/blurb.html, 2004.

SCONS. http://www.scons.org, 2004.

SOFTINTEGRATION. *The Ch Language Environment*. http://www.softintegration.com/products, 2001.

TRIDGELL, A. *ccache*. http://ccache.samba.org, 2002.

TROLLTECH. *Teambuilder*. http://www.trolltech.com/products/teambuilder/index.html.

WIKIPEDIA. http://www.wikipedia.org, 2004.

ZADROZNY, A. T. F. *PyDev*. http://pydev.sourceforge.net, 2004.

# *Anexo*

## Anexo A: Código Fonte

Aqui, é apresentado o código fonte das principais classes do sistema:

Arquivo: source/hydra/controller.py

```
from hydra import net
from hydra import utils
from schedulers import Dummy

#==============================================================================
# Controller
#==============================================================================
class Controller(object):

    def __init__(self):
        self.__lock__ = utils.Lock()
        self.scheduler = Dummy()
        self.servers = {}



    def GetAvaiableServer(self):
        '''Returns an avaiable server for compiling.
        @raise net.NoServerAvaiableError: if no server in the network is
        avaiable right now.
        '''
        if self.servers:
            server = self.scheduler.Choose(self.servers.values())
            del self.servers[server]
            utils.log.debug('server "%s" removed. %d left.' % \
                            (server.name, len(self.servers)))
            return server
        else:
            raise net.NoServerAvaiableError()

    GetAvaiableServer = utils.synchronized(GetAvaiableServer)
```

```
    def RegisterReadyServer(self, server):
        '''Registers a server on the controller as ready for compiling.
        @param server: a ServerInfo instance.
        '''
        if server not in self.servers:
            utils.log.info('server "%s" registered.', server.name)
        self.servers[server] = server

    RegisterReadyServer = utils.synchronized(RegisterReadyServer)


    def ServersCount(self):
        return len(self.servers)


    def Ping(self):
        return True
```

Arquivo: source/hydra/client.py

```
import job
from hydra import net
from hydra import exceptions
from hydra import utils

#===============================================================================
# Client
#===============================================================================
class Client(object):
    '''Represents the local compiler in the user's machine.
    Implements the task of pre-processing the source file in the user
    machine, compile it in some other remote computer, and write the resuling
    object file in the correct location.
    '''

    def __init__(self, compiler, params):
        self.compiler = compiler
        self.params = params
        self.job = job.Job(self.compiler, self.params)
        self.job.PreProcess()
        self.job.owner = utils.GetHost()


    def Go(self):
```

```
        '''Compiles the source file and write the result to the output file.
        @return: a triple (status, stdout, stderr).
        @raise hydra.Error: if any error inside the hydra system occurs.
        '''
        try:
            return self.CompileRemote()
        except (exceptions.Error, net.Error), e:
            #if not isinstance(e, net.NoServerAvaiableError):
            utils.log.debug('error in Go(): %s' % e)
            return self.CompileLocally()


    def Go(self):
        while True:
            try:
                return self.CompileRemote()
            except net.NoServerAvaiableError:
                import time
                time.sleep(0.1)
            except (exceptions.Error, net.Error), e:
                utils.log.debug('error: %s' % e)
                return self.CompileLocally()


    def CompileRemote(self):
        import time;s=time.clock()
        controller = net.FindController()
        while True:
            info = controller.GetAvaiableServer() # raises net.NoServerAvaiableError
            try:
                server = net.ConnectTo(info.uri)
                utils.log.debug('connecto to server: %.3f', time.clock()-s)
                s=time.clock()
                job = server.Compile(self.job)
                utils.log.debug('compile time: %.3f', time.clock()-s)
            except net.Error:
                continue # find a new server
            else:
                return self._FinishCompilation(job, info.name)


    def CompileLocally(self):
        lock = utils.GlobalLock('HydraLocalCompilation')
```

```
        lock.acquire()
        try:
            import time
            s = time.clock()
            job = self.job.Compile()
            print 'LocalCompile:', time.clock()-s
            return self._FinishCompilation(job, 'localhost')
        finally:
            lock.release()


    def _FinishCompilation(self, job, name):
        import time;s=time.clock()
        utils.Dashes(name)
        if job.CompilationSuccessful():
            job.WriteResult()
        res = job.result
        utils.log.debug('finish compilation: %.3s', time.clock()-s)
        return res.status, res.output
```

Arquivo: source/hydra/server.py

```
import socket
from hydra import net
from hydra import utils
import threading

#=============================================================================
# Server
#=============================================================================
class Server(object):
    '''Serves as a compilation server to Hydra clients.
    '''

    REGISTER_CONTROLLER_DELAY = 10.0 # register in the controller every 10s

    def __init__(self, uri=None, delay=REGISTER_CONTROLLER_DELAY):
        self.uri = uri
        self._compiling = False
        self._compiled_count = 0
        self._name = None
        self.register_delay = delay
        self.timer = utils.XTimer(delay, self.RegisterSelf)
```

```
        self.timer.start()
        self.RegisterSelf()


    def __del__(self):
        if self.timer is not None:
            self.timer.cancel()
            self.timer = None


    def IsCompiling(self):
        return self._compiling


    def Compile(self, job):
        '''Compiles the given job. If any communication error occurs,
        raise net.Error.
        '''
        if self._compiling:
            raise net.ServerBusyError('Already compiling, try again later')
        self._compiling = True
        owner = getattr(job, 'owner', '<unknown>')
        try:
            utils.log.info('compiling job for "%s"...', owner)
            job.Compile()
            utils.log.info('finished job for "%s".', owner)
            return job
        finally:
            self._compiling = False
            self._compiled_count += 1
            # call RegisterSelf() in another thread, to avoid a network error
            # to delay our return of the compiled job
            thread = threading.Thread(target=self.RegisterSelf)
            thread.start()



    def RegisterSelf(self):
        '''Register itself into the Controller.
        '''
        if not self._compiling and self.uri is not None:
            Error = net.Error
            utils.log.info('trying to register into the controller...')
```

```
            try:
                controller = net.FindController()
                controller.RegisterReadyServer(self.GetInfo())
                utils.log.info('registered in the controller.')
            except Error, e:
                utils.log.info('failed: %s' % e)
        else:
            utils.log.info('waiting compilation to complete...')


    def GetInfo(self):
        '''Returns a ServerInfo instance describing the server and its
        current state.
        '''
        info = ServerInfo(self.uri, self.GetName())
        # TODO: add info about CPU usage, memory, etc
        return info


    def GetName(self):
        '''get the name of the host.
        '''
        return socket.gethostname()


    def CompiledCount(self):
        '''Returns the total number of files compiled by this server since it's
        started.
        '''
        return self._compiled_count


#===============================================================================
# ServerInfo
#===============================================================================
class ServerInfo(object):
    '''Represents information about the server, its location and current state.
    '''

    def __init__(self, uri, name=None):
        self.name = name
        self.uri = uri
        self.cpu_usage = 0.0 # in percentage
```

```
    def __eq__(self, other):
        return self.uri == other.uri and \
               self.cpu_usage == other.cpu_usage

    def __ne__(self, other):
        return not self == other

    def __hash__(self):
        return hash(self.uri)

    def __repr__(self):
        return 'ServerInfo(uri="%s")'% self.uri
```

Arquivo: source/hydra/job.py

```
import zlib
import os
import tempfile
from hydra import utils
from hydra import settings

#============================================================================
# exceptions
#============================================================================
class Error(RuntimeError):

    PREFIX = None

    def __init__(self, msg):
        RuntimeError.__init__(self, self.PREFIX + msg)


class PreProcessError(Error):

    PREFIX = 'Error preprocessing source file: '


#============================================================================
# CompilationResult
#============================================================================
class CompilationResult(object):
```

```
    def __init__(self, status, output):
        self.status = status
        self.output = output
        self.compilation = None


    def SetCompilation(self, contents):
        if settings.COMPRESS and contents:
            orig_size = len(contents)
            contents = zlib.compress(contents)
            comp_size = len(contents)
            print 'compilation compression gain: %.2f' % (comp_size*100.0/orig_size)
        self._compilation = contents

    def GetCompilation(self):
        # de-compress the internal contents
        if settings.COMPRESS:
            return zlib.decompress(self._compilation)
        else:
            return self._compilation

    #compilation = property(GetCompilation, SetCompilation)


#============================================================================
# Job
#============================================================================
class Job(object):
    '''
    Represents a file to be built in another computer.

    Handles all the details of pre-processing the file, compiling it and writing
    it to the destination.

    _ __init__(params, compiler)
    - PreProcess()
    - Compile()
    - WriteFile()
    '''

    def __init__(self, compiler, params):
        '''
        @param params: A list of the parameters that was passed to the
```

```
            compilation process.
        @param compiler: a Compiler instance describing the target compiler.
        '''
        self.params = params
        self.compiler = compiler
        self.cmdline = compiler.Parse(params)
        self.preprocess = None
        self.result = None
        self.owner = '<undefined>'


    def IsPreProcessed(self):
        return self._preprocess is not None


    def PreProcess(self):
        #lock = utils.GlobalLock('HydraPreProcessing')
        #lock.acquire()
        import time;s=time.clock()
        status, out, err = utils.Execute2(self.cmdline.GetPreProcessParams())
        if status != 0:
            raise PreProcessError('compiler returned status %d. Stderr:\n%s' % \
                                  (status, err))
        # save the preprocessing output
        self.preprocess = out
        utils.log.debug('preprocessing time: %.3f', time.clock()-s)
        #lock.release()


    def SetPreProcess(self, contents):
        # compress the contents internally, if apply
        if settings.COMPRESS and contents:
            orig_size = len(contents)
            contents = zlib.compress(contents)
            comp_size = len(contents)
            print 'preprocess compression gain: %.2f' % (comp_size*100.0/orig_size)
        self._preprocess = contents

    def GetPreProcess(self):
        # de-compress the internal contents
        if settings.COMPRESS:
            return zlib.decompress(self._preprocess)
        else:
```

```
        return self._preprocess

#preprocess = property(GetPreProcess, SetPreProcess)


def IsCompiled(self):
    return self.result is not None


def Compile(self):
    '''Compiles the preprocessed job, and puts the resulting object file into the
    attribute compilation.
    '''
    cmdline = self.compiler.Parse(self.params)
    utils.log.debug('Parameters parsed successfully (%s)' % cmdline.source_file)
    tempdir = tempfile.gettempdir()
    # write the preprocessed source to the temp dir with the same original name
    name = os.path.basename(cmdline.source_file)
    cmdline.source_file = os.path.join(tempdir, name)
    f = file(cmdline.source_file, 'w')
    try:
        f.write(self.preprocess)
    finally:
        f.close()
    utils.log.debug('preprocessing written')

    # change the output filename
    name = os.path.basename(cmdline.output_file)
    cmdline.output_file = os.path.join(tempdir, name)

    # execute the compiler
    utils.log.debug('executting the compiler...')
    status, out = utils.Execute(cmdline.params)
    utils.log.debug('done. status = %d' % status)
    result = CompilationResult(status, out)

    # read the object file if the compilation was successful
    if status == 0 and os.path.isfile(cmdline.output_file):
        f = file(cmdline.output_file, 'rb')
        try:
            result.compilation = f.read()
        finally:
            f.close()
```

```
                os.remove(cmdline.output_file)
        utils.log.debug('read compilation results')
        self.result = result
        os.remove(cmdline.source_file)
        return self


    def CompilationSuccessful(self):
        return self.result is not None and self.result.status == 0 and \
            self.result.compilation is not None



    def WriteResult(self):
        '''Writes the result of the compilation to the original destiny.
        '''
        assert self.CompilationSuccessful(), "trying to write a failed compilation"
        # write to the output file
        f = file(self.cmdline.output_file, 'wb')
        try:
            f.write(self.result.compilation)
        finally:
            f.close()
```

Arquivo: source/hydra/schedulers.py

```
'''
Module that implements schedulers, which are objects that choose the "best"
server for a compilation at the momment.
Those classes defined in this module are used by the Controller.
'''


#=============================================================================
# Dummy
#=============================================================================
class Dummy(object):
    '''Very basic implementation of a scheduler, just returns the first server
    in the given list.
    '''



    def Choose(self, servers):
        '''Chooses the most appropriate ServerInfo from the given list.
        @param servers: a list of ServerInfo instances.
        @raise ValueError: if the given list is empty.
```

```
        @raise TypeError: if the parameter is not a list.
        '''
        if not isinstance(servers, list):
            msg = 'servers parameter must be a list; %s received' % type(servers)
            raise TypeError(msg)
        if not servers:
            raise ValueError('empty list given to Choose()')
        return servers[0]
```

Arquivo: source/hydra/pyronet/baseserver.py

```
import time
from threading import Thread
import Pyro.core
import Pyro.errors

#============================================================================
# BaseServer
#============================================================================
class BaseServer(Pyro.core.ObjBase):

    ID = None

    def __init__(self):
        Pyro.core.ObjBase.__init__(self)
        self._shutdown = False
        if self.ID is None:
            raise NotImplementedError('ID class attribute must be defined')


    def StartLoop(self, host='', port=0):
        Pyro.core.initServer(banner=False)
        self.mydaemon = daemon = Pyro.core.Daemon(host=host, port=port)
        daemon.connect(self, self.ID)
        self.uri = self.getProxy().URI
        kwargs = dict(timeout=3.0, condition=lambda: not self.IsShutdown())
        self._thread = Thread(target=daemon.requestLoop, kwargs=kwargs)
        self._thread.start()


    def IsShutdown(self):
        return self._shutdown


    def Shutdown(self):
```

```
        self._shutdown = True
        self._thread.join()


    def BusyLoop(self, host='', port=0):
        self.StartLoop(host=host, port=port)
        try:
            while not self.IsShutdown():
                time.sleep(0.1)
        except KeyboardInterrupt:
            self.Shutdown()
```

Arquivo: source/hydra/pyronet/controller.py

```
from baseserver import BaseServer
from hydra import controller
from exceptions import *

#============================================================================
# Controller
#============================================================================
class Controller(controller.Controller, BaseServer):

    ID = 'HydraController'
    USES_NS = False

    def __init__(self):
        BaseServer.__init__(self)
        controller.Controller.__init__(self)



#============================================================================
# FindController
#============================================================================
TIMEOUT = 3.0 # timeout for pinging the controller

def FindController():
    try:
        Pyro.core.initClient(0)
        controller = Pyro.core.getProxyForURI('PYROLOC://frodo/HydraController')
        controller._setTimeout(TIMEOUT)
        controller.Ping()
    except Error, e:
        raise ControllerNotFoundError(str(e))
```

```
    else:
        return controller
```

Arquivo: source/hydra/pyronet/server.py

```
from threading import Thread
from baseserver import BaseServer
from hydra import server

#==============================================================================
# Server
#==============================================================================
class Server(server.Server, BaseServer):

    ID = 'HydraServer'
    USES_NS = False

    def __init__(self, *args, **kwargs):
        BaseServer.__init__(self)
        server.Server.__init__(self, *args, **kwargs)



    def Shutdown(self):
        BaseServer.Shutdown(self)
        if self.timer is not None:
            self.timer.cancel()
            self.timer = None



    def StartLoop(self, *args, **kwargs):
        BaseServer.StartLoop(self, *args, **kwargs)
        thread = Thread(target=self.RegisterSelf)
        thread.start()
```

Abaixo, dois dos Unit-Tests do sistema são aprestandos.

Arquivo: source/hydra/tests/test_client.py

```
import os
import unittest
import testutils
import hydra
from hydra import dummynet
```

```
from hydra.client import Client


#============================================================================
# MyClient
#============================================================================
class MyClient(Client):
    '''Overwrite Client so we can be sure which of its methods were actually
    called.
    '''
    def __init__(self, *args):
        self._compiled_status = None
        Client.__init__(self, *args)

    def CompileRemote(self):
        result = Client.CompileRemote(self)
        self._compiled_status = 'remote'
        return result

    def CompileLocally(self):
        result = Client.CompileLocally(self)
        self._compiled_status = 'locally'
        return result


#============================================================================
# TestClient
#============================================================================
class TestClient(unittest.TestCase):


    def setUp(self):
        self.compiler = testutils.GetNativeCompiler()
        # install dummy net as our network layer
        hydra.InstallNetworkLayer(dummynet)
        # tmp dir
        os.mkdir('_tmp_test')


    def tearDown(self):
        testutils.delete('_tmp_test')
```

```
    def F(self, filename):
        return os.path.join(os.path.dirname(__file__), filename)


    def O(self, filename):
        return os.path.join('_tmp_test', filename)


    def CompileParams(self, source, target=None):
        includes = ['_tmp_test']
        if target:
            target = self.O(target)
        params = testutils.GetObjectParams(self.F(source), target, includes)
        return params


    def testNoServersAvaiable(self):
        '''local compilation by lack of servers.
        '''
        dummynet.Controller.SERVERS[:] = []
        params = self.CompileParams('circle.cpp', 'circle.o')
        client = MyClient(self.compiler, params)
        status, output = client.Go()
        self.assertEqual(status, 0)
        self.assertEqual(client._compiled_status, 'locally')
        self.assert_(os.path.isfile(self.O('circle.o')))


    def testSingleRemoteCompiled(self):
        '''simple compilation using one server.
        '''
        server = dummynet.Server('dummy')
        dummynet.Controller.SERVERS[:] = [server]

        params = self.CompileParams('circle.cpp', 'circle.o')
        client = MyClient(self.compiler, params)
        status, output = client.Go()
        self.assertEqual(status, 0)
        self.assertEqual(client._compiled_status, 'remote')
        self.assert_(os.path.isfile(self.O('circle.o')))
        self.assertEqual(server.compile_count, 1)
```

```
    def testFailedRemoteCompilation(self):
        ''' make the client compile because the server failed for some reason.
        '''
        server = dummynet.Server('dummy')
        def fails(*args):
            raise dummynet.Error
        server.Compile = fails
        dummynet.Controller.SERVERS.append(server)

        params = self.CompileParams('circle.cpp', 'circle.o')
        client = MyClient(self.compiler, params)
        status, output = client.Go()
        self.assertEqual(status, 0)
        self.assertEqual(client._compiled_status, 'locally')
        self.assert_(os.path.isfile(self.O('circle.o')))
        self.assertEqual(server.compile_count, 0)


#============================================================================
# main
#============================================================================
if __name__ == '__main__':
    unittest.main()
```

Arquivo: source/hydra/tests/test_server.py

```
import unittest
import time
import threading
import hydra
from hydra import net
from hydra import dummynet
from hydra import server


#============================================================================
# TestServer
#============================================================================
class TestServer(unittest.TestCase):


    def setUp(self):
        hydra.InstallNetworkLayer(dummynet)
        self.controller = net.FindController()
```

```
        self.controller.SERVERS[:] = []
        self.controller.EXISTING_SERVERS.clear()
        self.server = server.Server('server')
        self.server.register_delay = 0.1 # register in the controller every .1s
        self.controller.EXISTING_SERVERS[self.server.uri] = self.server


    def tearDown(self):
        self.controller.SERVERS[:] = []


    def testServer(self):
        # make sure our server connected to the Controller
        self.assertEqual(len(self.controller.SERVERS), 1)
        self.assertEqual(self.controller.SERVERS[0].uri, 'server')
        self.assert_(not self.server.IsCompiling())

        # fake job: just wait a little and pretends it is compiling something
        class Job:

            def Compile(self):
                time.sleep(1.0)

        # get the server from the controller
        info = self.controller.GetAvaiableServer()
        server = net.ConnectTo(info.uri)
        self.assertEqual(info.uri, 'server')
        self.assertEqual(self.controller.SERVERS, [])

        # tell the server to compile the job in another thread
        job = Job()
        thread = threading.Thread(target=server.Compile, args=(job,))
        thread.start()
        time.sleep(0.2)

        # check compiler state while compiling
        self.assert_(self.server.IsCompiling())
        self.assertEqual(len(self.controller.SERVERS), 0)
        # try to compiler another job and check for the error
        self.assertRaises(net.ServerBusyError, server.Compile, Job())

        # wait for the "job" to finish
        thread.join()
```

```
        time.sleep(0.1)
        self.assertEqual(len(self.controller.SERVERS), 1)
        self.assertEqual(self.controller.SERVERS[0].uri, 'server')
        self.assert_(not self.server.IsCompiling())


    def testServerInfo(self):
        # test hashability and equality
        info1 = server.ServerInfo('server1')
        info1b = server.ServerInfo('server1')
        info2 = server.ServerInfo('server2')
        self.assertEqual(info1, info1b)
        self.assertNotEqual(info1, info2)
        self.assertEqual(hash(info1), hash(info1b))
        self.assertNotEqual(hash(info1), hash(info2))

        info1b.cpu_usage = 50.0
        self.assertNotEqual(info1, info1b)

        # test stringfying
        self.assertEqual(repr(info1), 'ServerInfo(uri="%s")'% info1.uri)


#============================================================================
# main
#============================================================================
if __name__ == '__main__':
    unittest.main()
```