UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANELLENA ANDRADE SANTOS

Avaliação de desempenho do ambiente de programação X-Kaapi através do benchmark BOTS

Projeto de Diplomação

Prof. Dr. Nicolas Maillard
Orientador

Me. João Lima
Co-orientador

Porto Alegre, julho de 2013

“I’m a great believer in luck, and I find the harder I work the more I have of it.”

— Unknown Author

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BOTS	Barcelona OpenMP Task Suite
CUDA	Compute Unified Device Architecture
GPPD	Grupo de Processamento Paralelo e Distribuído
GPU	Graphics Processing Unit
HPF	High Performance Fortran
MPI	Message-Passing Interface
OpenMP	Open Multi-Processing
XKAAPI	Kernel for Adaptative, Asynchronous Parallel and Interactive Porgramming

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

No contexto de programação paralela, existem vários paradigmas e conceitos diferentes para expressar paralelismo. O caminho para se construir programas mais eficientes são ferramentas capazes de suportar mais de um desses paradigmas, para serem mais flexíveis e abrangentes.

X-Kaapi é uma ferramenta que provê APIs para paralelização de códigos com suporte a multíplos paradigmas de programação paralela. Entre eles, se destacam o paralelismo de tarefas e fluxo de dados. Ela também possui um sistema para escalonamento dinâmico que usa o algoritmo de roubo de tarefas.

Não existem, atualmente, muitos testes disponíveis para avaliar essa ferramenta. O objetivo deste trabalho é tentar suprir essa lacuna, ao realizar testes para avaliar o desempenho dela para diferentes tipos de aplicações. Para isso, é utilizado o *benchmark* BOTS, que foi desenvolvido para a implementação de paralelismo tarefas do OpenMP.

Palavras-chave: X-Kaapi, OpenMP, programação paralela, BOTS, recursão, tarefas, fluxo de dados.

Performance evaluation of X-Kaapi programming environment using BOTS benchmark

ABSTRACT

At the context of parallel programming, there are lots of different paradigms and concepts used to express parallelism. The path to build more efficient programs is to use tools that are able to support more than one of these paradigms, so they can be more flexible and embracing.

X-Kaapi is a tool that provides APIs for code parallelization that support a multiplicity of paradigms, the more important ones being task parallelism and data flow. It also has a runtime system for dynamic scheduling that uses a work stealing algorithm.

Currently, there are not many available tests to evaluate this tool. This study's goal is to fill this gap, by writing tests to evaluate its performance for different kinds of applications. The BOTS benchmark is used, which was developed for the OpenMP task parallelism implementation.

Keywords: X-Kaapi, OpenMP, parallel programming, BOTS, recursion, tasks, data flow.

1 INTRODUÇÃO

Limitações físicas, como consumo de energia e tamanho de transistores, sugerem o fim da lei de Moore (MOORE, 2006). Isso significa que não se pode mais contar apenas com o aumento da frequência de *clock* do processador para se obter melhores desempenhos. Como alternativa, a tendência tem sido a exploração da replicação de unidades de execução, com dois ou mais núcleos em um mesmo chip. De fato, a maioria dos computadores e notebooks vendidos atualmente possuem, no mínimo, dois cores.

Visando acompanhar essa evolução da arquitetura, é importante que as aplicações consigam aproveitar as unidades de processamento extra para melhorar seu desempenho. Para isso, existem diversas abordagens de programação paralela voltadas para diferentes tipos de *hardware* e *software*. Algumas, como o paradigma de troca de mensagens e memória compartilhada, definem maneiras de se usar o espaço de endereçamento para passar ou compartilhar informações entre partes do programa rodando em *threads* ou processos distintos. Outras definem a divisão do problema baseado em dados ou operações executadas através de fluxo de dados ou paralelismo de tarefas.

Existem várias APIs e ferramentas que provêem mecanismos para paralelização de código usando um ou outro paradigma. O desafio, no entanto, é desenvolver ambientes de programação e execução que ofereçam suporte eficiente a múltiplos paradigmas, para se obter melhores desempenhos. Nesse contexto, X-Kaapi oferece diferentes paradigmas como fluxo de dados, paralelismo de tarefas e memória compartilhada e um escalonador de baixa sobrecarga (GAUTIER et al., 2012).

1.1 Contribuição

Como a biblioteca X-Kaapi é recente e ainda está em desenvolvimento, não existem muitos testes disponíveis para medir seu desempenho. O objetivo desse trabalho é prover uma avaliação dessa ferramenta usando sua interface C, chamada kaapic. Para isso, foram utilizados quatros problemas diferentes do benchmark BOTS (DURAN et al., 2009). Ele foi escolhido por ser voltado para a implementação de paralelismo de tarefas de OpenMP, o que possibilita a comparação entre esta API e a ferramenta que é objeto de estudo deste trabalho.

Os problemas escolhidos foram: Sort, por ser uma implementação recursiva; SparseLU, por ser uma implementação iterativa para matrizes esparsas; Strassen, por ser uma implementação recursiva para matrizes densas; e Floorplan, por ser um algoritmo de otimização, com construções mais complexas.

Para a avaliação é utilizado o *speedup* das versões com X-Kaapi e OpenMP. Os dados de média e desvio padrão também são apresentados para justificar as curvas geradas. O objetivo é mostrar a diferença de desempenho entre as versões para diferentes tipos de

problemas, e qual o ganho que cada uma tem em relação aos algoritmos sequenciais.

1.2 Organização do texto

Neste trabalho, o Capítulo 2 explica o contexto científico de programação paralela. Ele introduz arquiteturas paralelas, e explica os principais paradigmas de programação. Por fim, ele apresenta algumas ferramentas que suportam o paralelismo de tarefas, que é o principal paradigma suportado por X-Kaapi.

O Capítulo 3 é dedicado à biblioteca que é o objeto de estudo deste trabalho, X-Kaapi. Seu objetivo é mostrar as principais características dessa ferramenta e prover o conhecimento necessário para que o leitor possa compreender o uso desta para a programação dos algoritmos testados.

No Capítulo 4 é apresentado o *benchmark* utilizado e os testes realizados. Para cada problema avaliado, é feita uma descrição do algoritmo, explicado como ele foi programado originalmente com OpenMP e como foi feita a implementação com kaapic, e, por fim, são apresentados os desempenhos obtidos.

O último capítulo conclui este trabalho, revisando o que foi apresentado e propondo sugestões para trabalhos futuros.

2 CONTEXTO CIENTÍFICO

Este capítulo tem por objetivo contextualizar o trabalho desenvolvido. Assim, nele são descritas algumas das tecnologias e ferramentas que existem atualmente na área de computação paralela. A Seção 2.1 apresenta uma introdução a arquiteturas de computadores paralelos. A Seção 2.2 explica os principais modelos de programação disponíveis para explorar o paralelismo oferecido pelo *hardware*. Por fim, a Seção 2.3 são apresentadas algumas ferramentas que usam modelo de tarefas, que está no cerne do assunto desse trabalho.

2.1 Arquiteturas Paralelas

Para alcançar melhor desempenho sem depender apenas da frequência de *clock*, uma solução é computadores paralelos. Dois modelos importantes são máquinas com mais de uma unidade de processamento no mesmo chip, e ter mais de um computador trabalhando juntos. O primeiro é representado por arquiteturas *multicore*, onde a memória física é compartilhada. O segundo descreve arquiteturas multicomputadores, que possuem memória distribuída.

Flynn propôs uma classificação que caracteriza os modelos de computadores paralelos (FLYNN, 1972). Ele os classifica quanto ao número de fluxo de dados e instruções, em quatro categorias:

- SISD: Representa arquiteturas puramente sequenciais, seguindo o modelo de Von Neuman. Uma instrução opera sobre um dado de cada vez.
- SIMD: Uma mesma instrução opera sobre vários dados.
- MISD: Múltiplas instruções operam sobre apenas um dado. Essa categoria é apenas teórica, não existem máquinas que satisfaçam essa classificação.
- MIMD: Múltiplas instruções operam sobre diferentes dados. É o modelo que representa arquiteturas multicore e multicomputadores. Quase todos os computadores paralelos são baseados nesse modelo (RAUBER; RUNGEL, 2010).

Arquiteturas paralelas não constituem apenas computadores voltados para computação de alto desempenho. Elas estão presentes em computadores pessoais e até mesmo dispositivos como celulares. Mas, para garantir que se obtenha o melhor desempenho com esse modelo de *hardware*, é preciso que os programas possam ser executados de maneira a aproveitar as unidades de processamento extra. E, para isso, é preciso que programadores e/ou compiladores sejam capazes de identificar operações que possam ser executadas em paralelo (QUINN, 2004).

2.2 Paradigmas de Programação Paralela

Usar compiladores para explorar paralelismo a nível de instruções não é suficiente para alcançar melhor desempenho. O mesmo é verdade quando se trata de identificar paralelismo a nível de threads. Então, para que os programas possam aproveitar o melhor das arquiteturas paralelas, é necessário que o programador determine o que pode ser paralelizável e como devem ser controlados os acessos aos dados e sincronizações. Para isso, existem várias bibliotecas e APIs que provêem funções e diretivas de compilação que estendem linguagens de programação sequenciais permitindo que o código seja paralelizado.

Essas ferramentas podem estar ligadas à arquitetura onde o programa vai ser executado. Alguns padrões, como OpenMP, são usados quando existe memória compartilhada. Outros, como MPI, são voltadas para modelos com memória distribuída. No entanto, os modelos de *hardware* e de programação são disjuntos; *hardware* com memória compartilhada provê excelente suporte a passagem de mensagens, assim como memória distribuída pode suportar modelo de programação de memória compartilhada (DONGARRA et al., 2003). Para facilitar a distinção entre ambos os tipos de modelos, a partir de agora o termo memória será usado como espaço de endereçamento.

A seguir são descritos alguns dos modelos mais importantes de programação paralela.

2.2.1 Troca de Mensagens

Nesse modelo, o espaço de endereçamento é distribuído. Cada unidade de processamento possui acesso direto apenas à sua memória local, e a comunicação com outras unidades é feita através de uma rede interconectada (QUINN, 2004). Basicamente, ele pode ser imaginado como uma coleção de processos executando programas sequenciais escritos em alguma linguagem sequencial aumentada com chamadas a uma biblioteca de funções para enviar e receber mensagens (FOSTER, 1995).

Esse modelo é natural a arquiteturas de multicomputadores, que possuem uma memória física distribuída. No entanto, ele também é usado em multiprocessadores, com variáveis compartilhadas como meio para a comunicação entre os processos (QUINN, 2004).

O padrão mais popular de troca de mensagens é o MPI, que surgiu com a versão 1.0 em 1994 (QUINN, 2004). Ele não é uma biblioteca, mas sim uma especificação para um grupo de funções que gerenciam o movimento de dados entre os processos. Apesar de ser muito abrangente, seis funções se destacam como básicas para implementar um programa paralelo com ele. São elas: inicialização e finalização (MPI_Init e MPI_Finalize); informação do número de processos e identificação do processo que está rodando (MPI_Comm_size e MPI_Comm_rank); e duas funções para mandar e receber mensagens (MPI_Send e MPI_Recv).

2.2.2 Memória compartilhada

Nesse caso, tarefas são referenciadas como *threads*, com a ideia de contexto de execução. Elas compartilham um mesmo espaço de endereçamento, onde podem ler e escrever assincronamente. Uma vantagem desse modelo para o programador é a falta de noção de possessão de dados. Assim, não há necessidade de especificar a comunicação explicitamente (FOSTER, 1995). Por outro lado, mecanismos como *locks* e semáforos precisam ser usados para evitar problemas de concorrência. Por exemplo, uma *thread* pode precisar impedir que outras leiam um local na memória até que ela acabe de modificá-lo.

Programação com memória compartilhada geralmente segue o paralelismo fork/join. Existe uma *thread* principal que executa a parte sequencial do código. Quando operações paralelas são necessárias, ela cria ou ativa outras *threads* (*fork*). Estas são destruídas ou desativadas depois que executam, resumindo o controle para a principal (*join*) (QUINN, 2004). O padrão POSIX *threads* — PThreads — representa uma implementação de baixo nível desse modelo. Ele provê funções para criar e destruir *threads* e coordenar o acesso à memória (DONGARRA et al., 2003).

Usando o padrão PThreads, um contexto de execução é criado através da função `pthread_create`. Ela recebe como parâmetros o endereço da função a ser executada e seus argumentos, e retorna um identificador da nova *thread*. Ao ser criada, ela também é escalonada para execução. Geralmente, a função `pthread_exit` é usada pela *thread* para sinalizar sua finalização (BUTENHOF, 1997).

2.2.3 Fluxo de dados

O paralelismo de fluxo de dados pode ser representado por tarefas independentes, aplicando uma mesma operação sobre elementos diferentes de um conjunto de dados (QUINN, 2004). Assim, a granularidade da computação é naturalmente pequena, e relativa à quantidade de dados distribuído para cada tarefa. Cabe ao programador prover informação para que seja feita a divisão dos dados.

Paralelismo de dados pode ser explorado tanto com memória compartilhada quanto distribuída. Para memória distribuída, o dados devem ser distribuídos entre os processadores de maneira que cada processador possa acessar a sua parte dos dados da sua memória local (RAUBER; RUNGEL, 2010).

CUDA é um exemplo de ferramenta que pode ser usada para expressar esse modelo de paralelismo. Ela possui uma API que expande a linguagem C, permitindo que o programador defina um novo tipo de função, que é chamado de *kernel*. Este, quando chamado, é executado N vezes em paralelo sobre um conjunto de dados, por N diferentes *threads* CUDA (CORPORATION, 2012).

Uma função *kernel* é definida através do idetificador `__global__`. O exemplo 2.1 mostra uma chamada a esse tipo de função. Nele, N é o número de blocos paralelos que irão executar a função e M é o número de *threads* por bloco. Entre parênteses, são passados os parâmetros da função *kernel*.

Código 2.1: CUDA kernel

```
1 int main() {
2   //[...]
3   kernelExample<<<N,M>>>(a, b, c);
4   //[...]
5 }
```

2.2.4 Paralelismo de Tarefas

Diferente do modelo de fluxo de dados, no paralelismo de tarefas existem tarefas independentes aplicando diferentes operações sobre elementos diferentes. Elas são basicamente partes do programa que podem ser executadas isoladamente, tais como blocos básicos, chamadas de funções ou laços. Esse modelo pode ser representado por um grafo, onde os nodos são as tarefas e as arestas mostram as dependências entre elas. Normal-

mente, uma tarefa não pode ser iniciada até que todas aquelas das quais ela depende terminem de executar (RAUBER; RUNGEL, 2010). Esse modelo facilita a paralelização de aplicações onde as unidades de trabalho são geradas dinamicamente, como em estruturas recursivas ou laços `while`.

Cilk-5 (FRIGO; LEISERSON; RANDALL, 1998) implementa o modelo de tarefas através de expansões para a linguagem C, usando palavras-chave. Essa ferramenta será discutida na Seção 2.3. X-Kaapi é outra biblioteca que explora o modelo de tarefas. Como Cilk, ela provê funções e mecanismos para criação de tarefas, mas também possui especificação do modo de acesso dos dados, visando identificar dependências, e evitando o uso de sincronização explícita. O Capítulo 2 discute detalhadamente essa ferramenta.

2.2.5 Anotações de código

Anotações de código, ou diretivas de compilação, são ferramentas de comunicação com o compilador normalmente usadas por várias APIs para expressar paralelismo. Elas são compreendidas apenas pelo compilador que as suporta. Possuem um formato que permite que elas sejam ignoradas caso o compilador não as conheça, para que o programa execute normalmente sem elas. Em C/C++ uma diretiva de compilação é chamada de `pragma`. Em Fortran, geralmente essas diretivas são expressadas através de comentários especiais de código.

HPF foi a primeira tentativa de definir diretivas de compilação para orientar o programador a especificar paralelismo. HPF introduz diretivas de distribuição de dados para prover controle sobre localidade. Essas diretivas são apenas recomendações ao compilador, não instruções. O compilador pode não obedecê-las se, por exemplo, ele determinar que o desempenho pode ser melhorado ao ignorá-las.

Essa ferramenta define as seguintes diretivas: `PROCESSORS` é usada para especificar a forma e tamanho de um *array* de processadores abstratos; `ALIGN` é usada para alinhar elementos de diferentes *arrays* entre si, indicando que eles devem ser mapeados para o mesmo processador; por fim, `DISTRIBUTE` é usada para distribuir um objeto em um array de processadores abstratos (FOSTER, 1995).

O exemplo 2.2 demonstra o uso de algumas das diretivas descritas. Nele, a computação do elemento A é distribuída em blocos entre o array de vinte processadores abstratos (p). Além disso, o uso da diretiva `ALIGN` define que o elementos C(i,j) devem ser mapeados para o mesmo processador que os A(j,i), assim como todos os elementos de B devem ser mapeados para o mesmo processador que todos os de A.

Código 2.2: Exemplo HPFortran

```
1 !HPF$ PROCESSORS p(20)
2       real A(100,100), B(100,100), C(100,100)
3 !HPF$ ALIGN B(:,:) WITH A(:,:)
4 !HPF$ ALIGN C(i,j) WITH A(j,i)
5 !HPF$ DISTRIBUTE A(BLOCK,*) ONTO p
```

OpenMP é uma API inicialmente voltada apenas para Fortran e mais tarde estendida para suportar também C e C++. Suas diretivas para essas duas linguagens são especificadas usando a diretiva de preprocessamento `pragma`, cuja sintaxe está representada no Código 2.3. Já para Fortran, todas as diretivas começam como um comentário especial que é definido como um `sentinel`, cuja sintaxe pode variar (BOARD, 2008). A Seção

2.3.2 descreve com mais detalhes o uso dessa interface.

Código 2.3: Diretiva OpenMP

```
1 #pragma omp directive-name [clause[ [,] clause]...]
```

Outra API que utiliza anotações de código semelhantes à sintaxe de OpenMP para expressar paralelismo é o OpenACC. De fato, a diferença do modelo de diretivas desta em relação a anterior é apenas o uso de `acc` no lugar de `omp`, como mostrado no Código 2.4. Essa interface de programação tem um modelo onde a execução é controlada por uma unidade de processamento hospedeira, com um disposito de aceleração anexo, como uma GPU. Ela tem ganhado espaço ultimamente entre as ferramentas voltadas para arquiteturas híbridas (OpenACC Working Group, 2013).

Código 2.4: Diretiva OpenACC

```
1 #pragma acc directive-name [clause[ [,] clause]...]
```

2.3 Ferramentas de Programação

2.3.1 Cilk

Cilk foi desenvolvida para programação paralela de propósitos gerais, mas é especialmente eficiente para explorar paralelismo dinâmico e assíncrono (GROUP, 2001).

Essa ferramenta provê um escalonador dinâmico que implementa o algoritmo de roubo de tarefas (BLUMOFE; LEISERSON, 1999), que é detalhado na Seção 3.3. Além disso, ela usa uma estratégia chamada dois clones, onde o compilador gera duas versões de cada procedimento Cilk: um rápido e um lento. O rápido corresponde à implementação sequencial e é executado sempre que a execução serial é suficiente. O lento é executado toda vez que há vantagem em utilizar a versão paralela (FRIGO; LEISERSON; RANDALL, 1998).

Cilk adiciona três palavras-chave à linguagem C para indicar paralelismo e sincronização. Para identificar uma função como uma tarefa Cilk, é usada a palavra-chave `cilk` antes da declaração. O paralelismo é criado pela geração dessas funções, através da palavra-chave `spawn`. A diferença entre o uso desta e uma chamada comum de função C é que, no caso mencionado, quem criou a tarefa continua executando. Ou seja, essa operação é não bloqueante. É usada sincronização explícita para garantir que a tarefa pai possa usar seguramente os valores retornados por todos os seus filhos, através da palavra-chave `sync`. O trecho de Código 2.5 mostra a implementação de Fibonacci usando essa ferramenta.

2.3.2 OpenMP

OpenMP é uma interface de programação voltada para aplicações de memória compartilhada. Ela é constituída por um conjunto de diretivas de compilação, rotinas de bibliotecas e variáveis de ambiente que podem ser usadas para especificar paralelismo em programas Fortran, C, e C++ (DONGARRA et al., 2003). A API não possui muitas diretivas diferentes, mas elas são suficientes para cobrir uma grande variedade de necessidades (CHAPMANM; JOST; PAS, 2008).

Código 2.5: Fibonacci com Cilk-5

```
1  cilk int fib (int n) {
2    int x, y;
3    if (n < 2) return n;
4
5    x = spawn fib (n-1);
6    y = spawn fib (n-2);
7
8    sync;
9    return x + y;
10 }
```

OpenMP suporta o modelo fork-join, já mencionado dentro do modelo de memória compartilhada. Novas threads são criadas sempre que uma construção paralela é encontrada. Para determinar as regiões paralelas, o programador simplesmente insere uma diretiva `parallel` imediatamente antes do início do trecho de código a ser paralelizado. No fim de uma região paralela existe uma barreira de sincronização implícita, ou seja, é necessário esperar que todas as threads executando esse bloco terminem para que o programa continue. Barreiras também podem ser definidas explicitamente através da diretiva `barrier` (CHAPMANM; JOST; PAS, 2008).

Se o programador quiser especificar a distribuição do trabalho entre as threads, existem diretivas especiais para isso. A abordagem mais comum é usar um laço `for`. Um exemplo do que foi descrito pode ser visto no Código 2.6. Nele, o pragma:

- Leva a disparar threads (`parallel`). No entanto, o número de threads não é especificado;
- Distribui as iterações do laço começado na linha seguinte entre as threads (`for`);
- Torna privada a variável de indução `i` implicitamente, e a variável `res`, explicitamente (`private`);
- Aloca as iterações entre as threads de forma **Round Robin**, por blocos de três iterações (`schedule`).

Código 2.6: OpenMP Exemplo 1

```
1 #pragma omp parallel for schedule(static,3) private(res)
2 for (i=0; i<100; ++i) {
3   res = calculo(i);
4 }
```

O segundo exemplo, do trecho de Código 2.7, introduz o uso da diretiva `default(none)`. Ela indica que o programador deverá especificar os atributos de compartilhamento de dados. Assim, é preciso explicitar para cada variável se ela será compartilhada entre todas as threads (`shared`) ou privada(`private`). Note o use da cláusula `firstprivate`, que possibilita que a variável seja pré inicializada (CHAPMANM; JOST; PAS, 2008).

Outro ponto interessante do segundo exemplo é o uso da diretiva `single`. Ela especifica que a construção associada, no caso, a impressão da mensagem, será executada por apenas uma *thread*. As outras *threads* associadas à mesma construção paralela esperam em uma barreira implícita até que ela termine, a não ser que uma cláusula `nowait` seja especificada.

Código 2.7: OpenMP Exemplo 2

```
1 #pragma omp parallel for default(none) shared(a,c)
2 firstprivate(res,i)
3 for (i=0; i<a; ++i) {
4   #pragma omp single
5   printf("Starting work...");
6
7   res = calculo(i) - c;
8 }
```

2.3.2.1 Suporte a tarefas

A partir da especificação 3.0, de 2008, OpenMP passou a prover suporte ao modelo de tarefas. Foi introduzida uma diretiva `task`, que pode ser usada no lugar de `parallel` para criar construções de tarefas. Toda vez que uma thread encontra uma região dessas, uma nova tarefa é criada e então atribuída para alguma thread para ser executada. A execução depende da disponibilidade da tarefa. Assim, ela pode ocorrer imediatamente ou ser postergada. Além disso, ao encontrar um ponto de escalonamento, a execução pode ser temporariamente suspensa(BOARD, 2008).

Se uma tarefa não for executada imediatamente ou for suspensa, ela pode ser recuperada posteriormente pela mesma thread que a encontrou, ou por outra. Se nada for definido, a thread que iniciou a execução é a única que pode retomar a execução (`tied`). Se a cláusula `untied` for usada na construção da tarefa, então qualquer thread pode executá-la. No exemplo 2.8, a cada nível de recursão, são criadas duas tarefas desse tipo(BOARD, 2008).

Os pontos de escalonamento de tarefas determinam lugares onde a implementação pode interromper a execução de uma tarefa e iniciar ou resumir outra. Existem quatro situações padrão onde isso pode acontecer: o ponto imediatamente depois da geração explícita de uma tarefa; depois da última instrução de uma região de tarefa; em regiões `taskwait`; e em regiões de barreira explícita ou implicita(BOARD, 2008).

Outra diretiva introduzida nessa especificação é `taskwait`. Ela determina que deve-se esperar pela finalização de todas as tarefas filhas geradas desde o início da tarefa corrente para que esta seja terminada. O exemplo 2.8 mostra o uso dessa diretiva na última linha. Ela é necessária, porque a última operação depende dos dados produzidos pelas duas tarefas anteriores.

Código 2.8: Fibonacci com OpenMP

```
1  long long fib (int n) {
2    long long x, y;
3    if (n < 2) return n;
4
5    #pragma omp task untied shared(x) firstprivate(n)
6    x = fib(n - 1);
7    #pragma omp task untied shared(y) firstprivate(n)
8    y = fib(n - 2);
9
10   #pragma omp taskwait
11   return x + y;
12 }
```

3 X-KAAPI

Este capítulo apresenta a biblioteca X-Kaapi, contextualizando o leitor com a ferramenta que foi objeto de estudo deste trabalho. Nele, são descritas as duas APIs atualmente disponíveis, kaapi++ e kaapic, os métodos de definição de acesso à memória e o algoritmo de escalonamento utilizado. Por fim, são comentados os testes disponíveis atualmente para avaliação.

3.1 Visão Geral

Alguns dos paradigmas de programação paralela descritos no capítulo anterior podem ser combinados em uma mesma ferramenta. X-Kaapi é uma biblioteca que provê suporte a mais de um desses paradigmas. São eles: paralelismo de tarefas, fluxo de dados e memória compartilhada. Isso a torna mais flexível, sendo eficiente para paralelizar diferentes tipos de aplicação.

A execução de um programa X-Kaapi gera uma sequência de tarefas que acessam dados em uma memória compartilhada. Uma tarefa é uma função cuja assinatura contém os dados compartilhados com outras tarefas. A assinatura também contém o modo de acesso de cada parâmetro. Uma tarefa requisitando acesso de leitura deve esperar por todos os escritores anteriores antes de iniciar sua execução (HERMANN; RAFFIN; FAURE, 2009).

O modelo de X-Kaapi, assim como Cilk(FRIGO; LEISERSON; RANDALL, 1998), IntelTBB (CORPORATION, 2007) e OpenMP-3.0 (BOARD, 2008), habilita a criação de tarefas não-bloqueantes: a tarefa é criada e o programa continua sua execução (LIMA et al., 2012).

Para algoritmos recursivos, tarefas são geradas a cada ramificação, e a execução é controlada pelo grafo de fluxo de dados (HERMANN; RAFFIN; FAURE, 2009). Esse grafo é construído dinamicamente a medida que as tarefas são geradas. Ele representa as dependências de dados entre as tarefas. Através disso, o sistema de tempo de execução da X-Kaapi pode detectar tarefas concorrentes assim que suas entradas são produzidas. Essa abordagem evita a necessidade de sincronização explícita.

Para que o programador possa gerar um programa X-Kaapi a partir de um código sequencial, a biblioteca possui APIs para as linguagens C e C++. Elas se baseiam no modelo de Athapascan (GALILEE et al., 1998) (ROCH; REVIRE; GAUTIER, 2003). Na próxima seção são descritas as principais características dessas duas interfaces.

3.2 Interfaces de Programação

3.2.1 Kaapi++

Essa interface em C++ oferece mecanismos para a criação de tarefas e definição do modo de acesso das variáveis.

Cada tarefa, com exceção daquela criada na inicialização, deve possuir uma assinatura. Nela são definidos o número de parâmetros e o tipo e modo de acesso de cada um. Os principais modos suportados são leitura, escrita, leitura e escrita (acesso exclusivo), e escrita acumulativa. Essa sintaxe é usada para possibilitar que sejam identificadas as dependências de dados entre as tarefas. Dados são compartilhados entre duas tarefas se, e somente se, elas possuem o mesmo ponteiro como parâmetro efetivo. A implementação da tarefa reflete aquilo que foi definido na assinatura.

O trecho de Código 3.1 mostra um trecho da implementação do algoritmo fibonacci. Nele, é definida a tarefa `TaskFibo`, que possui dois parâmetros. O primeiro é um ponteiro com acesso de escrita, e o segundo um inteiro passado apenas como valor. Isso é declarado na assinatura da função, linhas 13 e 14. Os parâmetros da implementação devem ser consistentes com a assinatura.

Na implementação são criadas tarefas através de `ka::Spawn()`. Quando uma tarefa é criada, ela é colocada em uma pilha e o fluxo de controle continua a execução sem esperar pela sua finalização. É garantido que uma tarefa só vai iniciar sua execução quando todas as suas entradas foram produzidas, e que, ao final do programa, todas as tarefas criadas foram executadas.

Ainda no mesmo exemplo 3.1, temos a definição da tarefa `TaskSum`. Ela possui dois parâmetros com acesso de leitura, que são escritos pela duas tarefas criadas antes dela. Logo, existe uma dependência de leitura depois de escrita entre essas tarefas. Assim, existe uma sincronização implícita nesse ponto: a tarefa `TaskSum` só é executada depois que as tarefas anteriores terminaram, e isso é assegurado pelo sistema de execução.

3.2.2 Kaapic

Um código usando kaapic é delimitado por funções de inicialização (`kaapic_init`) e finalização (`kaapic_finalize`). A função de início deve ser chamada antes de qualquer outra rotina da biblioteca. Ela recebe como parâmetro uma *flag* que, quando diferente de zero, indica que só a *thread* principal deve ser iniciada. As demais ficarão suspensas até que o programa entre na região paralela (LEMENTEC; DANJEAN; GAUTIER, 2011).

Outras duas funções delimitam a região paralela do programa. Essas regiões são utilizadas para que as *threads* de sistema da X-Kaapi permaneçam ativas apenas durante a execução paralela. O trecho de Código 3.2 exemplifica o esqueleto de um programa com essas funções. Elas recebem como parâmtero a constante `KAAPIC_FLAG_DEFAULT`, que indica que o tipo de escalonamento usado para essa região é o dinâmico. Há também a possibilidade de determinar escalonamento estático, usando `KAAPIC_STATIC_SCHED`.

Finalmente, para criar uma tarefa, utiliza-se a função **kaapic_spawn**. Ela possui três parâmetros fixos: um atributo; o número de argumentos que a função a ser executa na tarefa recebe; e um ponteiro para ela. Além desses, ela recebe parâmetros opcionais, organizados em grupos de quatro. Eles identificam o modo de acesso, tipo, tamanho e valor de cada argumento da função que será executada. Assim como para **kaapi++**, a definição do modo de acesso é usado como meio para identificar dependências entre as tarefas. E como a outra API, a versão para C também suporta passagem de parâmetros por valor.

Código 3.1: Fibonacci com kaapi++

```
1  struct TaskSum : public ka::Task<3>::Signature
2      <ka::W<long>, ka::R<long>, ka::R<long> > {};
3
4  template<>
5  struct TaskBodyCPU<TaskSum> {
6    void operator() ( ka::pointer_w<long> r,
7                      ka::pointer_r<long> a,
8                      ka::pointer_r<long> b ) {
9      *r = *a + *b;
10   }
11 };
12
13 struct TaskFibo : public ka::Task<2>::Signature
14         <ka::W<long>, const long > {};
15
16 template<>
17 struct TaskBodyCPU<TaskFibo> {
18   void operator() ( ka::pointer_w<long> ptr,
19         const long n ) {
20     if (n < 2) {
21       *ptr = n;
22     } else {
23       ka::pointer<long> ptr1 = ka::Alloca<long>();
24       ka::pointer<long> ptr2 = ka::Alloca<long>();
25
26       ka::Spawn<TaskFibo>() ( ptr1, n-2);
27       ka::Spawn<TaskFibo>() ( ptr2, n-1);
28
29       ka::Spawn<TaskSum>() ( ptr, ptr1, ptr2 );
30     }
31   }
32 };
```

O trecho de Código 3.3 mostra a implementação de Fibonacci com a interface C. Note que os modos de acesso são os mesmos definidos no exemplo 3.1. Nesse caso também há um ponto de sincronização implícita, depois da criação das tuas tarefas `fibonacci`.

Apesar de usar fluxo de dados para calcular a dependência entre as tarefas, essa interface também permite que sejam criados pontos explícitos de sincronização. Para isso, existe a função `kaapic_sync`.

3.3 Escalonamento

X-Kaapi implementa escalonamento dinâmico a partir do algoritmo de roubo de tarefas. Essa técnica visa obter o balanceamento de carga entre os processos. Nessa abordagem, cada processador possui uma pilha de tarefas a serem executadas. Quando um deles se torna ocioso, ele se torna um ladrão e tenta roubar, através de uma requisição, uma tarefa da pilha de outro processador aleatório (vítima). Como resposta, a vítima manda

Código 3.2: Região paralela Kaapic

```
#include "kaapic.h"

int main() {
  kaapic_init(0);

  kaapic_begin_parallel(KAAPIC_FLAG_DEFAULT);
  //Parallel Region
  [...]

  kaapic_end_parallel(KAAPIC_FLAG_DEFAULT);

  kaapic_finalize();
}
```

Código 3.3: Fibonacci com kaapic

```
void sum(int* result1, int* result2, int* result) {
  *result = *result1 + *result2;
}

void fibonacci(int n, int* result) {
  if (n<2)
    *result = n;
  else {
    int* result1 = kaapic_alloca(sizeof(int));
    int* result2 = kaapic_alloca(sizeof(int));
    kaapic_spawn(0, 2, fibonacci,
        KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, n-1,
        KAAPIC_MODE_W, KAAPIC_TYPE_INT, 1, result1 );

    kaapic_spawn(0, 2, fibonacci,
        KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, n-2,
        KAAPIC_MODE_W, KAAPIC_TYPE_INT, 1, result2 );

    kaapic_spawn(0, 3, sum,
        KAAPIC_MODE_R, KAAPIC_TYPE_INT, 1, result1,
        KAAPIC_MODE_R, KAAPIC_TYPE_INT, 1, result2,
        KAAPIC_MODE_W, KAAPIC_TYPE_INT, 1, result );
  }
}
```

uma cópia de uma tarefa pronta, deixando a original marcada como roubada.

No caso da X-Kaapi, para encontrar uma tarefa pronta, o processador ocioso calcula as dependências de fluxo de dados verdadeiras, de acordo com os modos de acessos, para cada tarefa na pilha da vítima. Isso é feito através de uma iteração a partir da mais antiga até a mais recente, e termina assim que uma tarefa pronta é encontrada (GAUTIER et al., 2013). Quando as tarefas não estão sendo roubadas, são executadas seguindo a ordem FIFO, sem calcular as dependências. Isso é possível porque a execução sequencial é uma

ordem válida (GALILEE et al., 1998) (GAUTIER; BESSERON; PIGEON, 2007).

Essa abordagem é vantajosa para algoritmos recursivos, já que as tarefas são criadas dinamicamente. No entanto, para os não recursivos ela só é eficiente se a quantidade de roubos é alto. Caso contrário, a sobrecarga de calcular as dependências muitas vezes acaba prejudicando o desempenho. Para esses casos, a X-Kaapi também suporta escalonamento estático. E, além disso, é possível combinar os dois tipos de escalonamento, se necessário (HERMANN; RAFFIN; FAURE, 2009).

A X-Kaapi também possui uma otimização a implementação do roubo de tarefas. Ela introduz um protocolo de cooperação entre ladrões. Ele é utilizado da seguinte forma: se um ladrão tenta roubar uma vítima que já está sendo roubada, as tarefas são divididas entre os ladrões. Permitir requisições concorrentes diminui o número total de requisições, como demonstrado em (TCHIBOUKDJIAN; GAST; TRYSTRAM, 2012). Esse comportamento difere do que acontece na implementação de Cilk, onde o ladrão desiste de uma vítima que já está sendo roubada (FRIGO; LEISERSON; RANDALL, 1998).

3.3.1 Suporte a GPUs

Recentemente, X-Kaapi foi estendida para suportar arquiteturas híbridas (CPU e GPU). Trabalhos realizados com esse suporte mostram uma análise do desempenho usando dois problemas de algebra linear densa, produto de matrizes e fatorização de Cholesky. Os resultados obtidos superam estratégias estáticas. Isso devido a: um algoritmo de roubo de tarefas melhorado; implementação leve de tarefas na X-Kaapi; e uma busca otimizada por tarefas prontas (LIMA et al., 2012) (GAUTIER et al., 2013).

A nível de API, por enquanto, apenas kaapi++ suporta aplicações híbridas. Essa interface provê a possibilidade de definir duas implementações para uma mesma assinatura de tarefa, uma para CPU e outra para GPU. Para isso, existe uma especialização das classes `TaskBodyCPU` e `TaskBodyGPU` (GAUTIER et al., 2013). Através da assinatura, o sistema lida automaticamente com transferências de memória. Assim, detalhes de arquitetura são abstraídos do algoritmo. (LIMA et al., 2012)

Na implementação do escalonamento, X-Kaapi teve extensões para o uso do roubo de tarefas com GPUs (LIMA et al., 2012). Além disso, foram realizadas otimizações com o uso de localidade de dados (GAUTIER et al., 2013).

3.4 Conjunto de testes disponíveis

Para avaliar o modelo de programação da X-Kaapi, é importante que haja *benchmarks* e exemplos disponíveis. Atualmente, uma das principais fontes de testes para essa bibioteca está junto com o código dela. Existem exemplos para problemas como Fibonacci, N-queens e alguns outros algoritmos simples. No entanto, a maioria dos códigos disponíveis são para kaapi++.

Foram realizadas algumas avaliações usando algoritmos de álgebra linear densa, multiplicação de matrizes e fatoração de Cholesky, para kaapi++ (LIMA et al., 2012) (GAUTIER et al., 2013).

Algumas avaliações também foram realizadas com EUROPLEXUS, um código de simulação industrial para dinâmicas de rápida transição. Em (GAUTIER et al., 2012), foram realizados *benchmarks* comparando o desempenho de X-Kaapi com três modelos de programação: fork-join, laços paralelos e tarefas com fluxo de dados.

Como pode ser visto, não existem muitos testes disponíveis para avaliar o desempenho da X-Kaapi na interface kaapic. Visando suprir essa lacuna, esse trabalho propõe

uma extensão ao benchmark BOTS para acrescentar testes para X-Kaapi. A seguir, são mostrados os testes escolhidos e modificados para avaliação da biblioteca, e os resultados das medições de desempenho.

4 VALIDAÇÃO EXPERIMENTAL

Para validar o desempenho da implementação de tarefas da ferramenta X-Kaapi, foram usados quatro problemas do *benchmark* BOTS: Sort, Sparse LU, Strassen e Floorplan. Este capítulo explica a implementação de cada um desses algoritmos e apresenta os desempenhos obtido para cada um.

A Seção 4.1 justifica e introduz o *benchmark* escolhido. As Seções 4.2, 4.3, 4.4 e 4.5 explicam cada um dos problemas selecionados, comparam as implementações usando as duas ferramentas, e mostram os resultados obtidos. Por fim, é feita uma análise geral dos desempenhos observados.

A métrica utilizada para as avaliações foi o *speedup*, obtido através da média de 30 execuções. Todos os testes foram executados em na máquina BUGIO, do GPPD/UFRGS. Ela possui um processador Intel Core i7 930 2.8GHz, que possui oito cores, e 12GB de memória.

4.1 Benchmark BOTS

O objetivo do *benchmark* BOTS é testar a implementação de tarefas do OpenMP. A maioria dos *benchmarks* utilizados estão disponíveis publicamente no projeto Cilk, projeto Application Kernel Matrix, e o Olden Suite. Ele provê testes para diferentes aplicações, incluindo implementações recursivas, iterativas, e até mesmo não-determinísticas. Ele também oferece múltiplas versões de cada algoritmo, visando analisar diferentes possibilidades de implementações OpenMP (DURAN et al., 2009).

Outra característica desse *benchmark* é possuir um mecanismo de auto verificação, que garante uma semântica correta nas implementações. Os mecanismos utilizados para isso são três: aplicar um método de validação no resultado; comparar o resultado com dados de validação previamente estabelecidos; e executar uma versão sequencial do código para comparação (DURAN et al., 2009). Esse mecanismo ajuda a garantir que o programa continue correto depois de modificações e adaptações, o que facilita o porte para kaapic.

Por ser um *benchmark* que abrange diferentes tipos de problemas, ele é ideal para mostrar a flexibilidade da biblioteca X-Kaapi. Além disso, como foi desenvolvido para explorar o modelo de paralelismo de tarefas de OpenMP, isso facilita a adaptação a X-Kaapi, que também implementa esse modelo.

Para fazer a avaliação de desempenho, foram selecionados quatro problemas do BOTS. Sort, uma implementação de ordenação de inteiros que utiliza uma mistura de algoritmos recursivos. Sparse LU e Strassen, visando cobrir algoritmos de álgebra linear densa e esparsa. E Floorplan, um algoritmo de otimização. Nas próximas seções são detalhados cada um dos problemas utilizados, as implementações usando X-Kaapi e OpenMP, e os resultados obtidos com cada um.

4.2 Sort

4.2.1 Problema

Esse algoritmo utiliza uma versão modificada do *mergesort* para ordenar um vetor de números inteiros aleatórios. A abordagem implementada segue a seguinte lógica: primeiro, o vetor é ordenado recursivamente, com divisão sistemática em quatro subvetores. Depois, as partes são unidas novamente, usando um algoritmo paralelo de divisão e conquista. As tarefas são utilizadas para cada divisão e união. Quando o vetor chega a um tamanho suficientemente pequeno, uma versão sequencial do *quicksort* é usada. Esse tamanho pode ser definido em tempo de execução (DURAN et al., 2009).

Essa implementação permite a criação de tarefas dinamicamente, a cada nível da recursão. A granularidade das tarefas pode ser controlada configurando-se o valor de corte que define o uso do algoritmo sequencial.

4.2.2 Programação

O problema foi adaptado para a interface kaapic de duas maneiras diferentes. A primeira, mais semelhante com a abordagem utilizada em OpenMP, utiliza passagem de parâmetros como valor e por ponteiros. Essa abordagem precisa realizar sincronizações para garantir que os dados das tarefas estejam prontos quando elas são executadas. A segunda utiliza fluxo de dados para controlar as dependências entre as tarefas, através dos modos de acesso de parâmetros como leitura/escrita ou só leitura. Essa versão não precisa de sincronizações.

Os exemplos 4.1, 4.2 e 4.3 mostram a chamada da primeira tarefa da implementação, respectivamente para as versões kaapic com ponteiros, kaapic com fluxo de dados e OpenMP. Na primeira versão, os dois parâmetros que são passados como ponteiros são os mesmos que, na de fluxo de dados, possuem modo de acesso de leitura e escrita. Já o último parâmetro, como não precisa de referência pois a tarefa não vai modificá-lo, é passado como valor, o que equivale ao acesso de leitura. Nesse ponto, nenhuma das implementações precisam de sincronização, já que existe um ponto de sincronização implícito no final da região paralela. A versão OpenMP, por sua vez, não define o acesso às variáveis de forma explícita.

Código 4.1: Primeira Tarefa Versão Kaapic com ponteiros

```
1 kaapic_begin_parallel( KAAPIC_FLAG_DEFAULT );
2
3 kaapic_spawn(0,3, cilksort_par,
4   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, array,
5   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, tmp,
6   KAAPIC_MODE_V, KAAPIC_TYPE_LONG, 1, (long)bots_arg_size );
7
8 kaapic_end_parallel( KAAPIC_FLAG_DEFAULT );
```

Essa primeira tarefa chama uma função (cilksort_par), que divide o vetor a ser ordenado em quatro partes de tamanhos iguais, recursivamente. Ou seja, a cada execução são criadas quatro novas tarefas `cilksort_par`. A recursão encerra quando o tamanho do vetor atinge o ponto de corte, quando os elementos passam a ser ordenados de forma sequencial. Assim, são evitadas tarefas muito pequenas, que criam uma sobrecarga maior do que a vantagem de serem paralelizadas.

Código 4.2: Primeira Tarefa Versão Kaapic com fluxo de dados

```
1  kaapic_begin_parallel(KAAPIC_FLAG_DEFAULT);
2
3  kaapic_spawn(0, 3, cilksort_par,
4    KAAPIC_MODE_RW, KAAPIC_TYPE_LONG, bots_arg_size, array,
5    KAAPIC_MODE_RW, KAAPIC_TYPE_LONG, bots_arg_size, tmp,
6    KAAPIC_MODE_R, KAAPIC_TYPE_LONG, 1, bots_arg_size );
7
8  kaapic_end_parallel(KAAPIC_FLAG_DEFAULT);
```

Código 4.3: Primeira Tarefa Versão OpenMP

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  #pragma omp task untied
4  cilksort_par(array, tmp, bots_arg_size);
```

Depois disso, a função cria duas tarefas (mergesort_par) para unir novamente as partes, de duas em duas. E, por fim, é criada uma última tarefa para unir as duas metades resultantes. Os exemplos 4.4, 4.5 e 4.6 mostram as chamadas às tarefas que unem as partes do vetor. No caso da kaapic com ponteiros e OpenMP, há necessidade de um ponto de sincronização explícito antes das chamadas para `mergesort_par`, uma vez que ela depende dos dados das tarefas anteriores.

A tarefa `mergesort_par` une as partes do vetor recursivamente, criando duas novas tarefas a cada nível de recursão. Ela também possui um ponto de corte, onde a recursão encerra, e a união passa a ser executada sequencialmente.

Código 4.4: Merge Versão Kaapic com ponteiros

```
1   kaapic_sync();
2
3   kaapic_spawn(0,5, cilkmerge_par,
4     KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, A,
5     KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (A + quarter - 1),
6     KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, B,
7     KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (B + quarter - 1),
8     KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, tmpA );
9
10  kaapic_spawn(0,5, cilkmerge_par,
11    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, C,
12    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (C + quarter - 1),
13    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, D,
14    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (low + size - 1),
15    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, tmpC );
16
17  kaapic_sync();
18
19  cilkmerge_par(tmpA, tmpC - 1, tmpC, tmpA + size - 1, A);
```

Código 4.5: Merge Versão Kaapic com fluxo de dados

```
1  kaapic_spawn(0,5, cilkmerge_par,
2       KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,A,
3       KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,A+quarter-1,
4       KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,B,
5       KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,B+quarter-1,
6       KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpA );
7
8  kaapic_spawn(0,5, cilkmerge_par,
9       KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,C,
10      KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,C+quarter-1,
11      KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,D,
12      KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,low+size-1,
13      KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpC );
14
15 kaapic_spawn(0,5, cilkmerge_par,
16      KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpA,
17      KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,tmpC-1,
18      KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpC,
19      KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,tmpA+size-1,
20      KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,A );
```

Código 4.6: Merge Versão OpenMP

```
1  #pragma omp taskwait
2
3  #pragma omp task untied
4  cilkmerge_par(A, A + quarter - 1, B, B + quarter - 1, tmpA);
5  #pragma omp task untied
6  cilkmerge_par(C, C + quarter - 1, D, low + size - 1, tmpC);
7  #pragma omp taskwait
8  cilkmerge_par(tmpA, tmpC - 1, tmpC, tmpA + size - 1, A);
```

4.2.3 Desempenho

Para esse teste, a entrada foi um vetor de 10^8 elementos. Como ponto de corte, foi utilizado o valor 4096 para as funções *cilksort* e *mergesort*.

A Figura 4.1 mostra os *speedups* obtidos para cada uma das versões. Através das informações mostradas na Tabela 4.1 é possível observar que o desvio padrão das médias obtidas para o cálculo do *speedup* foi bem pequeno. Isso valida a curva obtida no gráfico de desempenho.

Nota-se que kaapic com fluxo de dados apresenta um desempenho melhor do que OpenMP a medida que o número de *threads* aumenta. Isso acontece porque o algoritmo de escalonamento da X-Kaapi consegue equilibrar melhor a carga entre as *threads*. Logo, ele lida melhor com a divisão do trabalho entre mais *threads*.

Mas, principalmente, a versão kaapic tem *speedup* melhor do que as duas outras versões analisadas, pela eliminação de sincronizações desnecessárias. A sobrecarga dessas

operações é maior do que aquela do cálculo da dependência de dados. Enquanto as sincronizações ocorrem sempre, o cálculo das dependências só acontece quando há roubo de tarefas.

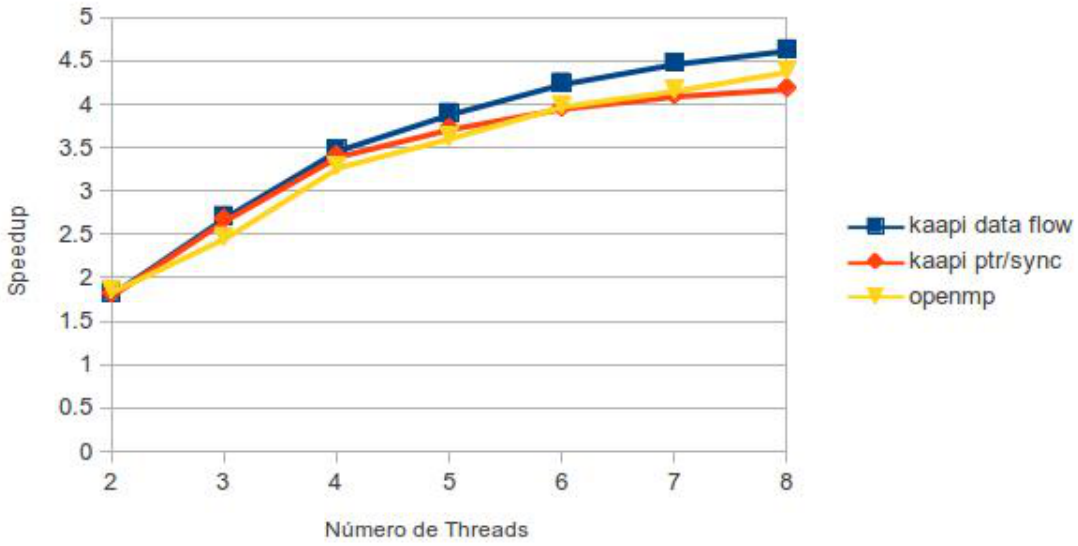

Figura 4.1: Gráfico de Speedup Sort

	Kaapic Fluxo de Dados		Kaapic Ptr/Sync		OpenMP	
Threads	Média	Desvio Padrão	Média	Desvio Padrão	Média	Desvio Padrão
2	6.299	0.015	6.334	0.015	6.207	0.02
3	4.258	0.008	4.312	0.011	4.666	0.237
4	3.319	0.03	3.386	0.008	3.515	0.104
5	2.962	0.013	3.099	0.023	3.188	0.127
6	2.72	0.017	2.914	0.037	2.896	0.106
7	2.58	0.031	2.814	0.035	2.771	0.082
8	2.495	0.035	2.758	0.035	2.631	0.072

Tabela 4.1: Desempenho Sort

4.3 Sparse LU

4.3.1 Problema

Esse algoritmo calcula a fatorização LU para uma matriz esparsa. A implementação divide a matriz em blocos. A quantidade e o tamanho dos blocos pode ser definida em tempo de execução. Como a matriz não é densa, alguns dos blocos são nulos. Em cada uma das fases da fatoração, uma tarefa é criada para cada bloco da matriz que não está vazio (DURAN et al., 2009).

4.3.2 Programação

O algoritmo utilizado para resolver esse problema é iterativo. Ele percorre os blocos da matriz aplicando as operações para montar o resultado da fatoração. Primeiro, para cada bloco da diagonal da matriz, ele aplica a função `lu0`. Depois ele modifica os elementos dos blocos das linhas, aplicando a função `fwd`. A próxima iteração modifica os elementos dos blocos das colunas da matriz, aplicando a função `bdiv`. Por fim, os blocos internos são modificados, usando os resultados calculados nas iterações anteriores, com a função `bmod`. Assim, no fim da execução o resultado da fatoração está no lugar dos valores originais da matriz.

Como a implementação é iterativa, foi usado escalonamento estático na versão com kaapic. Nesse caso, o número de roubos é muito grande, então a sobrecarga de usar escalonamento dinâmico é alto e acaba prejudicando o desempenho. As linhas 1 e 27 do Código 4.7 mostram o uso da flag `KAAPIC_FLAG_STATIC_SCHED` para configurar o escalonamento.

Na implementação utilizada, tarefas são criadas a cada iteração de cada laço. Os trechos de Código 4.7 e 4.8 mostram como foi implementada a criação das tarefas `lu0` e `bmod` para as interfaces kaapic e OpenMP, respectivamente. A demais tarefas foram omitidas para facilitar a visualização do código. Elas podem ser vistas no apêndice A, na Seção A.2.

Na versão com kaapic, os blocos da matriz que serão modificados são passados como parâmetros para as tarefas com modo de acesso de leitura e escrita. Os demais blocos recebem modo de acesso de leitura. A tarefa da linha 14 modifica um bloco que depende dos valores que são construídos pelas tarefas anteriores, logo existe um ponto de sincronização implícito. Como está sendo usado fluxo de dados, essa dependência é resolvida em tempo de execução pela biblioteca sem necessidade de uma chamada a `kaapic_sync()`. Note que os modos de acesso, nesses casos, são definidos apenas sobre os blocos da matriz.

No caso de OpenMP, é necessário explicitar que os índices usados para percorrer os blocos são privados para cada tarefa. Além disso, a matriz é declarada como compartilhada entre as tarefas, uma vez que todas elas acessam e modificam seus elementos. Como OpenMP não lida automaticamente com dependências, é necessário uma chamada explícita de sincronização na linha 7. Isso porque as entradas da próxima tarefa somente estarão prontas quando as tarefas anteriores acabarem sua execução.

4.3.3 Desempenho

O desempenho do algoritmo SparseLU foi medido com uma entrada de 75 blocos de tamanho 100x100, ou seja, uma matriz 7500x7500. A Figura 4.2 mostra o resultado dessas execuções.

O primeiro detalhe a se observar é que o desempenho de ambas as interfaces cresceu apenas até quatro *threads*. No caso de X-Kaapi, isso pode ter acontecido porque, usando o escalonamento estático, o balanceamento de carga não é eficiente.

Apesar disso, pode-se notar que, mesmo que o *speedup* de OpenMP tenha sido um pouco melhor, os resultados foram muito próximos. Isso fica evidente pela média dos tempos de cada execução, na Tabela 4.2. Considerando isso, é possível dizer que X-Kaapi, embora seja mais voltada para algoritmos recursivos, pode obter resultados tão bons quanto OpenMP para algoritmos iterativos. Isso comprova a flexibilidade dessa ferramenta, conforme mencionado na seção 3.1.

Código 4.7: SparseLU com Kaapic

```
kaapic_begin_parallel( KAAPIC_FLAG_STATIC_SCHED );
  for (kk=0; kk<blocks; kk++) {
  kaapic_spawn(0,1, lu0,
  /*diag*/
  KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,kk)
    );

  [...]

  for (ii=kk+1; ii<blocks; ii++)
      if (A(ii,kk) != NULL) {
  for (jj=kk+1; jj<blocks; jj++)
    if(A(kk,jj) != NULL) {
      kaapic_spawn(0,3, bmod,
    /*col*/
    KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(ii,kk),
    /*row*/
    KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,jj),
    /*inner*/
    KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(ii,jj)
      );
    }
      }

    } // for(kk...)
  }
kaapic_end_parallel( KAAPIC_FLAG_STATIC_SCHED );
```

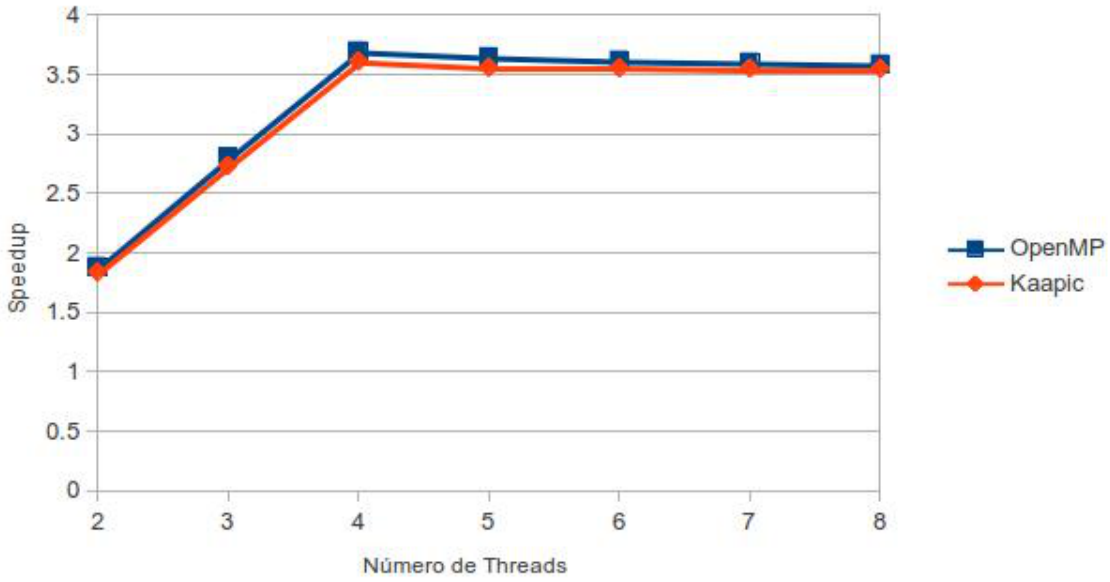

Figura 4.2: Gráfico de Speedup SparseLU

Código 4.8: SparseLU com OpenMP

```
#pragma omp parallel
#pragma omp single nowait
#pragma omp task untied
  for (kk=0; kk<bots_arg_size; kk++) {
    lu0(A(kk,kk));
[...]
  #pragma omp taskwait

  for (ii=kk+1; ii<num_blocks; ii++) {
      if (A(ii,kk) != NULL) {
      for (jj=kk+1; jj<num_blocks; jj++) {
  if(A(kk,jj) != NULL) {
#pragma omp task untied firstprivate(kk, jj, ii) shared(BENCH)
    if (A(ii,jj) == NULL)
      A(ii,jj) = allocate_clean_block();
    bmod(A(ii,kk), A(kk,jj), A(ii,jj));
  }
      } // for(jj...)
    } // if
  } //for(ii...)

  }
```

	Kaapic		OpenMP	
Threads	Média	Desvio Padrão	Média	Desvio Padrão
2	4.054	0.007	3.961	0.005
3	2.728	0.006	2.661	0.004
4	2.067	0.002	2.018	0.027
5	2.095	0.002	2.044	0.008
6	2.097	0.003	2.061	0.005
7	2.099	0.002	2.070	0.001
8	2.101	0.002	2.079	0.001

Tabela 4.2: Desempenho SparseLU

4.4 Strassen

4.4.1 Problema

Strassen é um algoritmo de multiplicação de matrizes que se baseia em recursão de um nível em blocos de tamanho 2x2. Ele visa reduzir a quantidade da operação de maior custo, que é a multiplicação. Em contraste com o algoritmo convencional, que utiliza 8 multiplicações e 4 adições, ele usa um esquema que precisa de 7 multiplicações e 18 adições (SONG; DONGARRA; MOORE, 2006). Considere a multiplicação de matrizes 2x2 mostrada na Figura 4.3. No caso do algoritmo Strassen, ela seria expressa conforme operações mostradas na Tabela 4.3.

$$M = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

Figura 4.3: Multiplicação de Matrizes

$p1 = (a11 + a22) * (b11 + b22)$	$p5 = (a11 + a12) * b22$
$p2 = (a21 + a22) * b11$	$p6 = (a21 - a11) * (b11 + b12)$
$p3 = a11 * (b12 - b22)$	$p7 = (a12 - a22) * (b21 + b22)$
$p4 = a22 * (b21 - b11)$	
$m11 = p1 + p4 - p5 + p7$	
$m12 = p3 + p5$	
$m21 = p2 + p4$	
$m22 = p1 + p3 - p2 + p6$	

Tabela 4.3: Operações Strassen

Para que o algoritmo possa ser aplicado diretamente, é necessário que a matriz seja quadrada, com dimensões 2^k. Como ele requer armazenamento temporário maior, e a divisão da matriz em blocos, ele é lento para matrizes pequenas. Uma melhoria para esse algoritmo é usar um ponto de corte na recursão, aplicando o método convencional para blocos pequenos (SONG; DONGARRA; MOORE, 2006).

O método usado pelo benchmark BOTS usa decomposição hierárquica. Ele divide cada dimensão da matriz em duas seções de tamanhos iguais. Para cada decomposição, uma nova tarefa é criada. Para evitar a criação de tarefas muito pequenas, é usado um ponto de corte (DURAN et al., 2009).

4.4.2 Programação

A implementação do algoritmo Strassen é dividida em quatro fases de operações que equivalem ao conjunto apresentado na subseção anterior. A Tabela 4.4 mostra as operações realizadas em cada uma dessas fases. A abordagem utilizada pelo BOTS consiste em executar sequencialmente a fase 1, criar uma tarefa para cada operação da fase 2, ou seja, uma para cada multiplicação, e então executar sequencialmente as fases seguintes. Cada tarefa executa as 4 fases para uma submatriz, divindo-a em submatrizes menores até que se chegue a um ponto de corte, quando a recursão para e a operação de multiplicação é executada através de um algoritmo sequencial de divisão e conquista. O apêndice A, Seção A.3 mostra essa implementação para kaapic e OpenMP.

Fase 1	$T1 = A11 + A22$	$T6 = B11 + B22$
	$T2 = A21 + A22$	$T7 = B12 - B22$
	$T3 = A11 + A12$	$T8 = B21 - B11$
	$T4 = A21 - A11$	$T9 = B11 + B12$
	$T5 = A12 - A22$	$T10 = B21 + B22$
Fase 2	$Q1 = T1 * T6$	$Q5 = T3 * B22$
	$Q2 = T2 * B11$	$Q6 = T4 * T9$
	$Q3 = A11 * T7$	$Q7 = T5 * T10$
	$Q4 = A22 * T8$	
Fase 3	$T1 = Q1 + Q4$	$T3 = Q3 + Q1$
	$T2 = Q5 - Q7$	$T4 = Q2 - Q6$
Fase 4	$C11 = T1?T2$	$C12 = Q3 + Q5$
	$C21 = Q2 + Q4$	$C22 = T3 - T4$

Tabela 4.4: Fases Strassen

A criação da tarefa com kaapic para uma das operações da fase 2 está exemplificada no trecho de Código 4.9. As demais chamadas e fases foram omitidas para facilitar a visualização. Os parâmetros passados para cada tarefa são: a submatriz que receberá o resultado da operação da tarefa, que, por isso, possui modo de acesso de leitura e escrita; as duas submatrizes usadas para chegar no resultado da primeira, que não são modificadas, então tem modo de acesso de apenas leitura; quatro inteiros com os tamanhos das submatrizes e o tamanho das linhas das submatrizes, que são passados como valor, já que não influenciam na dependência entre as tarefas. O último parâmetro indica a profundidade da recursão e não é usado nas versões testadas nesse trabalho.

Ao final da criação das tarefas existe um ponto de sincronização. Normalmente, não seria necessário uma chamada explícita a função `kaapic_sync`. No entanto, nesse algoritmo, a biblioteca não suporta apenas a ordem garantida pelo fluxo de dados.

O trecho de Código 4.10 mostra a versão da criação das tarefa com OpenMP, com a mesma simplificação. Nela são criadas tarefas que podem ser executadas por qualquer *thread*. Como o modo de acesso dos parâmetros não são explicitados, todos os parâmetros são tratados como privados. Vale notar que, nesse caso, em que cada tarefa tem muitos parâmetros, a implementação com OpenMP fica muito mais simples.

4.4.3 Desempenho

O desempenho do algoritmo Strassen implementado foi medido usando uma matriz de entrada de tamanho 4096x4096. O ponto de corte utilizado para o teste foi 64.

O Figura 4.4 mostra o *speedup* obtido com as duas implementações. Pode-se notar que as curvas são bem próximas. A diferença entre elas pode ser justificada como ruído estatístico, o que fica evidente pelas médias dos tempos e o desvio padrão mostrados na Tabela 4.5. Sendo assim, o desempenho de ambas foi bem próximo. A partir desse resultado, é importante observar que ao se usar sincronização explícita com X-Kaapi, o resultado obtido fica muito próximo daquele com OpenMP. Isso também se mostrou verdadeiro nos outros testes realizados.

Além disso, nesse caso, a implementação com OpenMP é mais simples. Como o desempenho das duas versões foi semelhante, a complexidade do uso de kaapic não é compensado pelo ganho da biblioteca. Isso sugere que trabalhos futuros integrem OpenMP com a biblioteca X-Kaapi, visando obter uma soma dos ganhos da interface do OpenMP e do escalonamento da X-Kaapi.

Código 4.9: Strassen com Kaapic

```
[...]

   kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
KAAPIC_MODE_RW,KAAPIC_TYPE_DBL,QuadrantSize*QuadrantSize,M2,
KAAPIC_MODE_R,KAAPIC_TYPE_DBL,QuadrantSize*QuadrantSize,A11,
KAAPIC_MODE_R,KAAPIC_TYPE_DBL,QuadrantSize*QuadrantSize,B11,
KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,(int)QuadrantSize,
KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,(int)QuadrantSize,
KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,(int)RowWidthA,
KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,(int)RowWidthB,
KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,(int)(Depth+1)
);

[...]

kaapic_sync();
```

Código 4.10: Strassen com OpenMP

```
[...]

#pragma omp task untied
OptimizedStrassenMultiply_par(M2, A11, B11,
   QuadrantSize, QuadrantSize, RowWidthA, RowWidthB, Depth+1);

[...]

#pragma omp taskwait
```

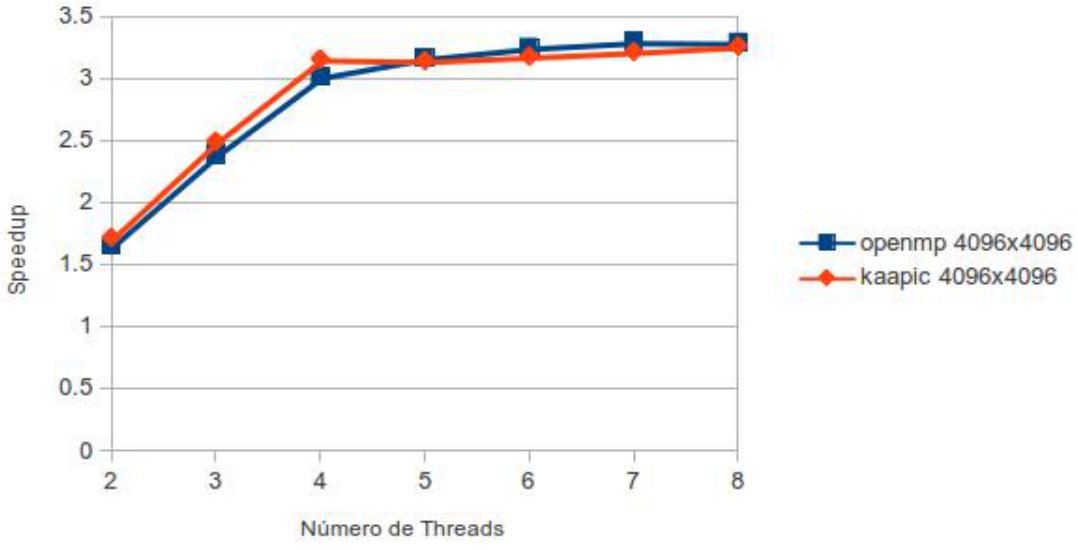

Figura 4.4: Gráfico de Speedup Strassen

	Kaapic		OpenMP	
Threads	Média	Desvio Padrão	Média	Desvio Padrão
2	10.502	0.159	10.901	0.203
3	7.232	0.119	7.554	0.241
4	5.712	0.071	5.979	0.307
5	5.736	0.062	5.695	0.197
6	5.672	0.041	5.549	0.047
7	5.605	0.041	5.471	0.026
8	5.529	0.029	5.481	0.047

Tabela 4.5: Desempenho Strassen

4.5 Floorplan

4.5.1 Problema

O problema de otimização Floorplan consiste em uma implementação para otimizar a área total ocupada por um grupo de células. Estas são definidas como um conjunto de blocos retangulares indivisíveis. Um mesmo tipo de célula possui apenas uma área, mas pode ter formas diferentes. Faz parte do algoritmo selecionar a forma que melhor se encaixa para minimizar a área. Essa solução pode ser aplicada na construção de componentes eletrônicos, como microprocessadores e *chips* de memória (FOSTER, 1995).

A Figura 4.5 mostra, em A, um conjunto de três células, com, respectivamente uma, três e duas formas possíveis. Em B, são mostradas todas as opções para distribuir as células. Em C, é apresentada a árvore que representa todas as possibilidades para essa distribuição. O nodo em destaque representa a melhor solução (menor área) para esse exemplo.

A entrada do algoritmo é um conjunto com as descrições de cada célula, incluindo área e formas. A saída é o tamanho de área mínimo que inclue todas as células. A abordagem utilizada pelo benchmark BOTS para encontrar a solução é uma busca recursiva *Branch and Bound*, onde tarefas são criadas para cada ramo da solução. O estado de cada algoritmo precisa ser copiado para cada tarefa recém criada. Isso implica na necessidade de sincronizações adicionais para manter o estado das tarefas pais (DURAN et al., 2009).

A ideia básica da implementação é manter a informação da melhor solução até o momento. Antes de expandir um nodo da árvore, é verificado se a área parcial da distribuição que ele representa é maior do que a melhor solução conhecida. Se for, a busca por esse caminho é abandonada (FOSTER, 1995).

4.5.2 Programação

A implementação utilizada inicia com a criação de uma tarefa que chama uma função recursiva que fará o cálculo da solução ótima. Ela também soma os nodos visitados para que seja possível medir a quantidade de nodos que foram visitados. Os trechos de Código 4.11 e 4.12 mostram essa primeira operação com cada uma das versões, kaapic e OpenMP, respectivamente.

A versão com kaapic possui um parâmetro a mais, *ntasks*, para armazenar o resultado da soma dos nodos. Isso é necessário porque as tarefas dessa interface não retornam nenhum valor, a não ser através de seus parâmetros. Na versão com OpenMP, esse valor é o retorno da função. Note que o grupo de parâmetros que descrevem esse último ar-

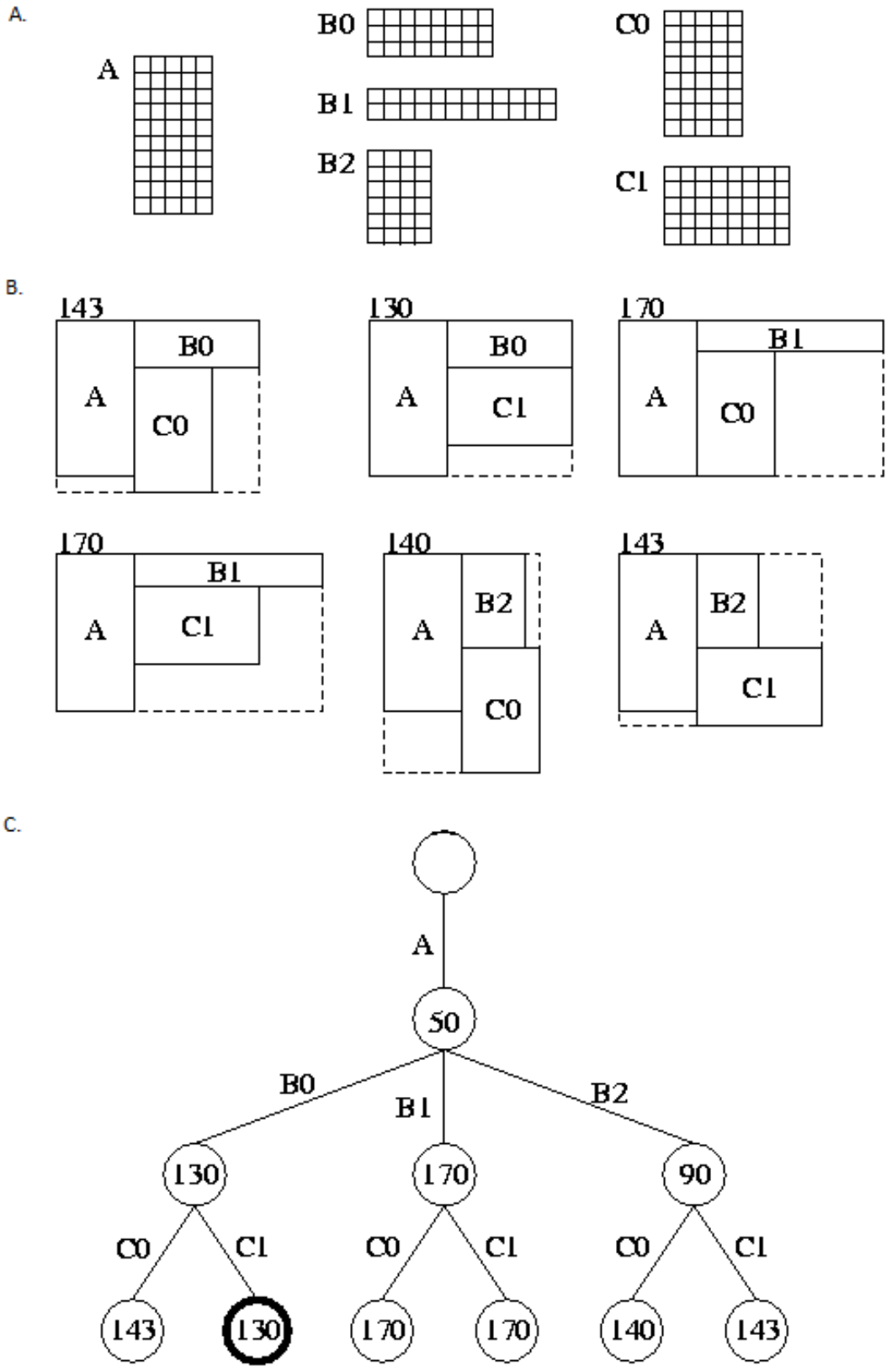

Figura 4.5: Passos Floorplan

gumento da tarefa kaapic possui um parâmetro a mais do que os outros. Isso acontece

Código 4.11: Primeira Tarefa com Kaapic

```
1  static kaapi_lock_t kaapi_lock;
2
3  int ntasks = 0;
4  kaapic_begin_parallel( KAAPIC_FLAG_DEFAULT );
5  kaapi_atomic_initlock(&kaapi_lock);
6  kaapic_spawn(0,5,add_cell,
7  KAAPIC_MODE_V,KAAPIC_TYPE_INT,1,1,
8  KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,footprint,
9  KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,board,
10 KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,gcells,
11 KAAPIC_MODE_CW,KAAPIC_REDOP_PLUS,KAAPIC_TYPE_INT,1,&ntasks
12 );
13 kaapic_sync();
14 kaapi_atomic_destroylock(&kaapi_lock);
15 kaapic_end_parallel( KAAPIC_FLAG_DEFAULT );
16
17 bots_number_of_tasks = ntasks;
```

Código 4.12: Primeira Tarefa com OpenMP

```
1 #pragma omp parallel
2 #pragma omp single
3 bots_number_of_tasks = add_cell(1, footprint, board, gcells);
```

porque o modo de acesso utilizado é escrita acumulativa. Por isso, é necessário que se informe, além do tipo, tamanho e do valor, a operação que será aplicada. Nesse caso, essa operação é a soma, indicada por `KAAPIC_REDOP_PLUS`.

Os demais parâmetros kaapic são passados como valor, uma vez que eles não são necessários para o cálculo de dependências, já que está sendo usado sincronismo explícito. Além disso, essa implementação introduz o uso de *locks* para garantir acesso exclusivo às variaveis dentro da tarefa, uma vez que são usadas variáveis globais para armazenar os resultados parciais do algoritmos. Para isso, são introduzidas duas novas funções da biblioteca X-Kaapi: `kaapi_atomic_initlock`, que inicia o `lock` que será utilizado dentro da tarefa, e `kaapi_atomic_destroylock`, que finaliza o uso desse.

No apêndice A, Seção A.4 estão os códigos com a implementação das versões kaapic e OpenMP. A lógica da implementação pode ser descrita de seguinte maneira: cada tarefa atua sobre um nodo da árvore. Dois laços são utilizados para percorrer todas as formas da célula naquele nó, e, para cada uma, todas as possibilidades de localização. Para cada uma dessas combinações, se a célula não pode ser encaixada, então essa combinação é abandonada. Se, por outro lado, a célula puder ser utilizada, sua área é somada ao contador da área. Se essa célula for a última (nodo folha) e a área somada for menor do que o último resultado armazenado, então o melhor resultado (armazenado em uma variável global) é atualizado. Se ela não for a última, mas a área encontrada até agora é menor do que o último resultado, então a função é chamada novamente para a próxima célula, criando uma nova tarefa. Caso contrário, esse caminho de busca é abandonada.

No momento do acesso à variável com o melhor resultado conhecido, existe uma

região crítica, já que tarefa escreve na variável. Por isso, para garantir que nenhuma outra tarefa acesse essa variável nesse momento, é usado o *lock* comentado anteriormente. O *lock* é criado através da função `kaapi_atomic_lock`, e liberado através de `kaapi_atomic_unlock`. O trecho de Código 4.13 mostra essa implementação. Para ter a mesma garantia de acesso exclusivo à variável `MIN_AREA`, a implementação OpenMP indica a região crítica através da diretiva `critical`, usada na linha 2 do Código 4.14.

Código 4.13: Região Crítica Kaapic

```
1 kaapi_atomic_lock(&kaapi_lock);
2 if (area < MIN_AREA) {
3   MIN_AREA = area;
4   [...]
5 }
6 kaapi_atomic_unlock(&kaapi_lock);
```

Código 4.14: Região Crítica OpenMP

```
1 if (area < MIN_AREA) {
2 #pragma omp critical
3   if (area < MIN_AREA)
4     MIN_AREA = area;
5     [...]
6 }
```

4.5.3 Desempenho

O desempenho do Floorplan foi medido através de uma entrada com 20 células. O Gráfico 4.6 mostra um desempenho muito ruim para OpenMP, mas algum *speedup* para kaapic.

Provavelmente, a razão da falta de *speedup* para OpenMP é porque são criadas muitas tarefas, e o escalonador não consegue distribuir bem a carga delas. Uma versão com cortes baseados na profundidade da árvore poderia melhorar esse resultado.

Já para kaapic, mais uma vez, o bom desempenho se dá por possuir um escalonador mais eficiente e por não usar sincronizações desnecessárias.

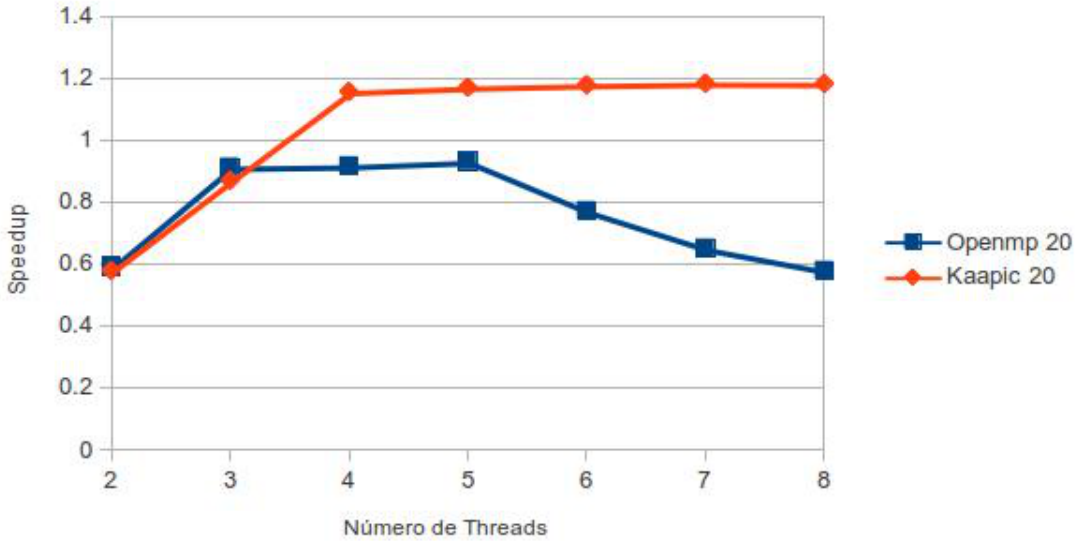

Figura 4.6: Gráfico de Speedup Floorplan

	Kaapic		OpenMP	
Threads	Média	Desvio Padrão	Média	Desvio Padrão
2	18.895	0.054	18.484	0.037
3	12.572	0.030	11.990	0.071
4	9.457	0.069	11.931	0.286
5	9.343	0.193	11.742	0.583
6	9.279	0.280	14.174	1.185
7	9.240	0.353	16.811	0.825
8	9.248	0.469	18.903	0.964

Tabela 4.6: Desempenho Floorplan

4.6 Conclusão

Neste capítulo foram usados quatro algoritmos diferentes para medir o desempenho da biblioteca X-Kaapi e sua interface kaapic em relação a implementação de tarefas do OpenMP. Os testes foram realizados com entradas grandes, cenário em que o paralelismo é melhor aproveitado.

A partir dos resultados apresentados, pode-se concluir que o escalonamento dinâmico usado pela X-Kaapi contribui significativamente para um melhor desempenho. Já o uso de sincronizações explícitas prejudica a eficiência. No entanto, mesmo nos casos em que elas foram utilizadas, os *speedups* obtidos ainda foram bons, embora não melhores do que os de OpenMP.

Assim, os resultados obtidos mostram que o uso de kaapic pode ser tão bom quanto OpenMP, ou até mesmo melhor. E isso se mostrou verdade em todos os testes, o que indica que X-Kaapi pode ser eficiente para diferentes tipos de implementações e algoritmos. No entanto, isso acontece muito mais por causa dos benefícios do escalonamento usado pelo sistema da X-Kaapi, do que pela interface C. Sem a sobrecarga dessa interface, é possível

que o desempenho para os algoritmos testados tivesse sido muito melhor. Considerando trabalhos anteriores (LIMA et al., 2012) (GAUTIER et al., 2013), acredita-se que a API kaapi++ poderia ter resultados melhores.

5 CONCLUSÕES

Neste trabalho foi usado o *benchmark* BOTS para avaliar o desempenho da biblioteca X-Kaapi e sua interface C, kaapic. A implementação de tarefas do OpenMP, para qual esse benchmark foi originalmente construído, foi utilizada para comparação. Foram escolhidos quatro testes, com o objetivo de testar kaapic com diferentes tipos de aplicações.

Foi executado um teste com um algoritmo de ordenação, Sort, com uma implementação recursiva. A implementação kaapic foi avaliada com duas versões, uma com fluxos de dados e escalonamento dinâmico, e outra com sincronizações explícitas e escalonamento estático. Os resultados mostraram que a primeira versão ganha em desempenho da implementação com OpenMP, enquanto a outra perde.

O segundo teste foi feito com um algoritmo de algebra linear esparsa, SparseLU, com implementação iterativa. Nesse caso, foi implementado apenas uma versão kaapic usando fluxo de dados e escalonamento estático. Os resultados mostraram que X-Kaapi pode ser tão eficiente quanto OpenMP para esse algoritmo iterativo.

O terceiro teste foi executado com um algoritmo de algebra linear densa, Strassen, que implementa uma abordagem recursiva. Ele foi implementado usando fluxo de dados, escalonamento dinâmico e sincronizações explícitas. Nesse caso, devido à utilização das sincronizações, o desempenho da X-Kaapi foi bem semelhante ao do OpenMP.

O último teste realizado utilizou o algoritmo Floorplan, um algoritmo de otimização recursivo, e introduziu o uso de *locks* pela X-Kaapi. Os resultados mostraram um *speedup* com essa biblioteca, enquanto a versão com OpenMP teve um desempenho abaixo do sequencial.

Os resultados dos testes mostraram que X-Kaapi é uma biblioteca flexível, tendo sido eficiente para diferente tipos de aplicações. No entanto, pode-se notar que seu desempenho é melhor para implementações recursivas. Isso se deve, principalmente, ao uso de escalonamento dinâmico com o algoritmo de roubo de tarefas, que é eficiente no balanceamento de carga entre os processos.

5.0.1 Trabalhos futuros

Baseado nos resultados obtidos, é possível perceber que o desempenho melhor da X-Kaapi se deve, principalmente, ao algoritmo de escalonamento utilizado. Uma abordagem para melhorar o desempenho, seria usar uma interface com uma sobrecarga menor do que kaapic. Como já foram obtidos bons resultados com *kaapi++*, seria interessante repetir os testes deste trabalho para essa interface.

Além disso, outra abordagem que poderia ser explorada, embora exija modificações para a própria biblioteca X-Kaapi, seria portar a interface OpenMP para ela. Assim, seria possível unir a vantagem da otimização no escalonamento com uma interface mais leve e

estável.

APÊNDICE A CÓDIGOS

Esse apêncide contém trechos de códigos com a implementação de cada um dos problemas testados. O objetivo é mostrar as criações de tarefas e usos das APIs, sendo assim, alguns trechos foram omitidos para facilitar a visualização do código.

A.1 Sort

Código A.1: Sort Versão Kaapic com fluxo de dados

```
1  void cilkmerge_par
2  (ELM *low1,ELM *high1,ELM *low2,ELM *high2,ELM *lowdest){
3     [...]
4     //Ponto de corte
5     if (high2 - low2 < bots_app_cutoff_value ) {
6     seqmerge(low1, high1, low2, high2, lowdest);
7     return;
8     }
9     [...]
10 kaapic_spawn(0,5, cilkmerge_par,
11    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,low1,
12    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,lowsize,split1-1,
13    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,low2,
14    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,lowsize,split2,
15    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,lowdest
16 );
17
18 kaapic_spawn(0,5, cilkmerge_par,
19    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,split1+1,
20    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,lowsize,high1,
21    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,split2+1,
22    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,lowsize,high2,
23    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,lowsize,lowdest+lowsize+2
24 );
25
26    return;
27 }
28
29 void cilksort_par(ELM *low, ELM *tmp, long size) {
30 [...]
31    //Ponto de corte
```

```
32    if (size < bots_app_cutoff_value_1 ) {
33        seqquick(low, low + size - 1);
34        return;
35    }
36  [...]
37  kaapic_spawn(0, 3, cilksort_par,
38    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,A,
39    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpA,
40    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,quarter );
41
42  kaapic_spawn(0,3, cilksort_par,
43    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,B,
44    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpB,
45    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,quarter );
46
47  kaapic_spawn(0,3, cilksort_par,
48    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,C,
49    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpC,
50    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,quarter );
51
52  kaapic_spawn(0,3, cilksort_par,
53    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,D,
54    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpD,
55    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,(size - 3 * quarter) );
56
57  kaapic_spawn(0,5, cilkmerge_par,
58    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,A,
59    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,A+quarter-1,
60    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,B,
61    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,B+quarter-1,
62    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpA );
63
64  kaapic_spawn(0,5, cilkmerge_par,
65    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,C,
66    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,C+quarter-1,
67    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,D,
68    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,low+size-1,
69    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpC );
70
71  kaapic_spawn(0,5, cilkmerge_par,
72    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpA,
73    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,tmpC-1,
74    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,tmpC,
75    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,tmpA+size-1,
76    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,quarter,A );
77  }
78
79  void sort_par ( void ) {
80  kaapic_begin_parallel(KAAPIC_FLAG_DEFAULT);
81  kaapic_spawn(0, 3, cilksort_par,
82    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,bots_arg_size,array,
```

```
83    KAAPIC_MODE_RW,KAAPIC_TYPE_LONG,bots_arg_size,tmp,
84    KAAPIC_MODE_R,KAAPIC_TYPE_LONG,1,bots_arg_size
85          );
86  kaapic_end_parallel(KAAPIC_FLAG_DEFAULT);
87  }
```

Código A.2: Sort Versão Kaapic com ponteiros

```
1   void cilkmerge_par
2     (ELM *low1,ELM *high1,ELM *low2,ELM *high2,ELM *lowdest){
3   [...]
4     if (high2 - low2 < bots_app_cutoff_value ) {
5       seqmerge(low1, high1, low2, high2, lowdest);
6       return;
7     }
8   [...]
9   kaapic_spawn(0,5, cilkmerge_par,
10    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, low1,
11    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (split1 -1),
12    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, low2,
13    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, split2,
14    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, lowdest
15      );
16  kaapic_spawn(0,5, cilkmerge_par,
17    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (split1+1),
18    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, high1,
19    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (split2+1),
20    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, high2,
21    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, (lowdest + lowsize + 2)
22      );
23    kaapic_sync();
24
25    return;
26  }
27
28  void cilksort_par(ELM *low, ELM *tmp, long size)
29  {
30  [...]
31    if (size < bots_app_cutoff_value_1 ) {
32      seqquick(low, low + size - 1);
33      return;
34    }
35  [...]
36  kaapic_spawn(0,3, cilksort_par,
37    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,A,
38    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpA,
39    KAAPIC_MODE_V,KAAPIC_TYPE_LONG,1,(long)quarter
40      );
41
42  kaapic_spawn(0,3, cilksort_par,
43    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,B,
44    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpB,
```

```
45    KAAPIC_MODE_V,KAAPIC_TYPE_LONG,1,(long)quarter
46       );
47
48 kaapic_spawn(0,3, cilksort_par,
49    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,C,
50    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpC,
51    KAAPIC_MODE_V,KAAPIC_TYPE_LONG,1,(long)quarter
52       );
53
54 kaapic_spawn(0,3, cilksort_par,
55    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,D,
56    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpD,
57    KAAPIC_MODE_V,KAAPIC_TYPE_LONG,1,(long)(size-3*quarter)
58       );
59 kaapic_sync();
60
61 kaapic_spawn(0,5, cilkmerge_par,
62    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,A,
63    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,(A + quarter - 1),
64    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,B,
65    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,(B + quarter - 1),
66    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpA
67       );
68
69 kaapic_spawn(0,5, cilkmerge_par,
70    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,C,
71    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,(C + quarter - 1),
72    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,D,
73    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,(low + size - 1),
74    KAAPIC_MODE_V,KAAPIC_TYPE_PTR,1,tmpC
75       );
76
77 kaapic_sync();
78
79 cilkmerge_par(tmpA, tmpC - 1, tmpC, tmpA + size - 1, A);
80 }
81
82
83 void sort_par ( void ) {
84 kaapic_begin_parallel( KAAPIC_FLAG_DEFAULT );
85 kaapic_spawn(0,3, cilksort_par,
86    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, array,
87    KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, tmp,
88    KAAPIC_MODE_V, KAAPIC_TYPE_LONG, 1, (long)bots_arg_size
89       );
90 kaapic_end_parallel( KAAPIC_FLAG_DEFAULT );
91 }
```

Código A.3: Sort Versão OpenMP

```
1  void cilkmerge_par
2    (ELM *low1,ELM *high1,ELM *low2,ELM *high2,ELM *lowdest) {
```

```
3  [...]
4    if (high2 - low2 < bots_app_cutoff_value ) {
5        seqmerge(low1, high1, low2, high2, lowdest);
6        return;
7    }
8  [...]
9  #pragma omp task untied
10   cilkmerge_par(low1, split1 - 1, low2, split2, lowdest);
11 #pragma omp task untied
12   cilkmerge_par(split1 + 1, high1, split2 + 1, high2,
13       lowdest + lowsize + 2);
14 #pragma omp taskwait
15
16
17   return;
18 }
19
20 void cilksort_par(ELM *low, ELM *tmp, long size) {
21 [...]
22   if (size < bots_app_cutoff_value_1 ) {
23       seqquick(low, low + size - 1);
24       return;
25   }
26 [...]
27 #pragma omp task untied
28   cilksort_par(A,tmpA,quarter);
29 #pragma omp task untied
30   cilksort_par(B,tmpB,quarter);
31 #pragma omp task untied
32   cilksort_par(C,tmpC,quarter);
33 #pragma omp task untied
34   cilksort_par(D,tmpD,size - 3 * quarter);
35
36 #pragma omp taskwait
37
38 #pragma omp task untied
39   cilkmerge_par(A,A + quarter - 1,B,B + quarter - 1,tmpA);
40 #pragma omp task untied
41   cilkmerge_par(C,C + quarter - 1,D,low + size - 1,tmpC);
42 #pragma omp taskwait
43
44   cilkmerge_par(tmpA,tmpC - 1,tmpC,tmpA + size - 1,A);
45 }
46
47 void sort_par ( void ) {
48 #pragma omp parallel
49 #pragma omp single nowait
50 #pragma omp task untied
51       cilksort_par(array, tmp, bots_arg_size);
52 }
```

A.2 SparseLU

Código A.4: SparseLU Versão Kaapic

```
1  void sparselu_par_call(float **BENCH) {
2    kaapic_begin_parallel( KAAPIC_FLAG_STATIC_SCHED );
3    for (kk=0; kk<blocks; kk++) {
4    kaapic_spawn(0,1, lu0,
5    /*diag*/
6    KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,kk)
7       );
8
9    for (jj=kk+1; jj<blocks; jj++)
10     if (A(kk,jj) != NULL) {
11       kaapic_spawn(0,2, fwd,
12   /*diag*/
13   KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,kk),
14   /*col*/
15   KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,jj)
16        );
17     }
18
19   for (ii=kk+1; ii<blocks; ii++)
20     if (A(ii,kk) != NULL) {
21     kaapic_spawn(0,2, bdiv,
22   /*diag*/
23   KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,kk),
24   /*row*/
25   KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(ii,kk)
26      );
27      }
28
29   for (ii=kk+1; ii<blocks; ii++)
30     if (A(ii,kk) != NULL) {
31       for (jj=kk+1; jj<blocks; jj++)
32   if(A(kk,jj) != NULL) {
33     kaapic_spawn(0,3, bmod,
34   /*col*/
35   KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(ii,kk),
36   /*row*/
37   KAAPIC_MODE_R,KAAPIC_TYPE_FLT,blck_size*blck_size,A(kk,jj),
38   /*inner*/
39   KAAPIC_MODE_RW,KAAPIC_TYPE_FLT,blck_size*blck_size,A(ii,jj)
40      );
41   }
42      }
43
44   } // for(kk...)
45
46   kaapic_end_parallel( KAAPIC_FLAG_STATIC_SCHED );
47 }
```

Código A.5: SparseLU Versão OpenMP

```
1  void sparselu_par_call(float **BENCH) {
2  #pragma omp parallel
3  #pragma omp single nowait
4  #pragma omp task untied
5    for (kk=0; kk<num_blocks; kk++) {
6      lu0(A(kk,kk));
7
8      for (jj=kk+1; jj<num_blocks; jj++) {
9        if (A(kk,jj) != NULL) {
10 #pragma omp task untied firstprivate(kk, jj) shared(BENCH)
11         fwd(A(kk,kk), A(kk,jj));
12       }
13     }
14
15     for (ii=kk+1; ii<num_blocks; ii++)
16       if (A(ii,kk) != NULL) {
17 #pragma omp task untied firstprivate(kk, ii) shared(BENCH)
18         bdiv(A(kk,kk), A(ii,kk));
19       }
20
21   #pragma omp taskwait
22
23     for (ii=kk+1; ii<num_blocks; ii++) {
24   if (A(ii,kk) != NULL) {
25         for (jj=kk+1; jj<num_blocks; jj++) {
26           if(A(kk,jj) != NULL) {
27 #pragma omp task untied firstprivate(kk, jj, ii) shared(BENCH)
28             if (A(ii,jj) == NULL)
29               A(ii,jj) = allocate_clean_block();
30             bmod(A(ii,kk), A(kk,jj), A(ii,jj));
31           }
32         } // for(jj...)
33       } // if
34     } //for(ii...)
35
36   #pragma omp taskwait
37   } // for(kk...)
38 }
```

A.3 Strassen

Código A.6: Strassen Versão OpenMP

```
1  void OptimizedStrassenMultiply_par(REAL *C, REAL *A, REAL *B,
     unsigned MatrixSize,
2      unsigned RowWidthC, unsigned RowWidthA, unsigned RowWidthB,
         int Depth)
3  [...] //Fase 1
4    //Ponto de corte
```

```
5   if (MatrixSize <= bots_app_cutoff_value) {
6     MultiplyByDivideAndConquer(C, A, B, MatrixSize, RowWidthC,
          RowWidthA, RowWidthB, 0);
7     return;
8   }
9  [...]
10    //Fase 2
11    kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
12 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, M2,
13 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, A11,
14 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, B11,
15 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
16 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
17 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthA,
18 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthB,
19 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
20       );
21    kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
22 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, M5,
23 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S1,
24 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S5,
25 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
26 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
27 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
28 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
29 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
30       );
31    kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
32 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize,
       T1sMULT,
33 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S2,
34 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S6,
35 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
36 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
37 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
38 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
39 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
40       );
41    kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
42 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, C22,
43 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S3,
44 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S7,
45 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
46 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthC,
47 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
48 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
49 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
50       );
51    kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
52 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, C11,
53 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, A12,
```

```
54 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, B21,
55 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
56 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthC,
57 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthA,
58 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthB,
59 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
60       );
61   kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
62 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, C12,
63 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S4,
64 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, B22,
65 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
66 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthC,
67 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
68 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthB,
69 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
70       );
71   kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
72 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, C21,
73 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, A22,
74 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, QuadrantSize*QuadrantSize, S8,
75 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
76 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthC,
77 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)RowWidthA,
78 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)QuadrantSize,
79 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)(Depth+1)
80       );
81
82   kaapic_sync();
83 [...] //Fases 3 e 4
84 }
85
86 void strassen_main_par(REAL *A, REAL *B, REAL *C, int n) {
87   kaapic_begin_parallel( KAAPIC_FLAG_DEFAULT );
88   kaapic_spawn(0, 8, OptimizedStrassenMultiply_par,
89 KAAPIC_MODE_RW, KAAPIC_TYPE_DBL, n*n, C,
90 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, n*n, A,
91 KAAPIC_MODE_R, KAAPIC_TYPE_DBL, n*n, B,
92 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)n,
93 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)n,
94 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)n,
95 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)n,
96 KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, (int)1
97   );
98   kaapic_sync();
99   kaapic_end_parallel( KAAPIC_FLAG_DEFAULT );
100 }
```

Código A.7: Strassen Versão OpenMP

```
1 void OptimizedStrassenMultiply_par(REAL *C, REAL *A, REAL *B,
    unsigned MatrixSize,
```

```
2       unsigned RowWidthC, unsigned RowWidthA, unsigned RowWidthB,
            int Depth)
3  [...]
4   if (MatrixSize <= bots_app_cutoff_value) {
5      MultiplyByDivideAndConquer(C, A, B, MatrixSize, RowWidthC,
           RowWidthA, RowWidthB, 0);
6      return;
7    }
8  [...]
9    #pragma omp task untied
10   OptimizedStrassenMultiply_par(M2,A11,B11,QuadrantSize,
         QuadrantSize,RowWidthA,RowWidthB,Depth+1);
11
12   #pragma omp task untied
13   OptimizedStrassenMultiply_par(M5,S1,S5,QuadrantSize,
         QuadrantSize,QuadrantSize,QuadrantSize,Depth+1);
14
15   #pragma omp task untied
16   OptimizedStrassenMultiply_par(T1sMULT, S2, S6,  QuadrantSize,
         QuadrantSize, QuadrantSize, QuadrantSize, Depth+1);
17
18   #pragma omp task untied
19   OptimizedStrassenMultiply_par(C22,S3,S7,QuadrantSize,RowWidthC
         ,QuadrantSize,QuadrantSize,Depth+1);
20
21   #pragma omp task untied
22   OptimizedStrassenMultiply_par(C11,A12,B21,QuadrantSize,
         RowWidthC,RowWidthA,RowWidthB,Depth+1);
23
24   #pragma omp task untied
25   OptimizedStrassenMultiply_par(C12,S4,B22,QuadrantSize,
         RowWidthC,QuadrantSize,RowWidthB,Depth+1);
26
27   #pragma omp task untied
28   OptimizedStrassenMultiply_par(C21,A22,S8,QuadrantSize,
         RowWidthC,RowWidthA,QuadrantSize,Depth+1);
29
30   #pragma omp taskwait
31 [...]
32 }
33
34 void strassen_main_par(REAL *A, REAL *B, REAL *C, int n) {
35 #pragma omp parallel
36 #pragma omp single
37 #pragma omp task untied
38   OptimizedStrassenMultiply_par(C, A, B, n, n, n, n, 1);
39 }
```

A.4 Floorplan

Código A.8: Floorplan Versão Kaapic

```
1  void add_cell_task(coor* NWS,int i,int j,int id,int nn,coor
       FOOTPRINT,ibrd BOARD,struct cell *CELLS,int* nnc ) {
2    [...]
3    if (! lay_down(id, board, cells)) {
4      goto _end;
5    }
6
7    [...]
8    /* Se última célula */
9    if (cells[id].next == 0) {
10     if (area < MIN_AREA) {
11       kaapi_atomic_lock(&kaapi_lock);
12       if (area < MIN_AREA) {
13         MIN_AREA          = area;
14         MIN_FOOTPRINT[0] = footprint[0];
15         MIN_FOOTPRINT[1] = footprint[1];
16         memcpy(BEST_BOARD, board, sizeof(ibrd));
17       }
18       kaapi_atomic_unlock(&kaapi_lock);
19     }
20   } else if (area < MIN_AREA) {
21     add_cell(cells[id].next, footprint, board,cells, nnc);
22   } else {}
23
24 _end:;
25 }
26
27 static void add_cell(int id,coor FOOTPRINT,ibrd BOARD,struct
       cell *CELLS,int *ntasks) {
28 [...]
29   /* Para cada forma */
30   for (i = 0; i < CELLS[id].n; i++) {
31     [...]
32     /* Para cada localização */
33     for (j = 0; j < nn; j++) {
34       kaapic_spawn(0,9, add_cell_task,
35   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, NWS,
36   KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, i,
37   KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, j,
38   KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, id,
39   KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, nn,
40   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, FOOTPRINT,
41   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, BOARD,
42   KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, CELLS,
43   KAAPIC_MODE_CW, KAAPIC_REDOP_PLUS, KAAPIC_TYPE_INT, 1, &nnc
44     );
45     }
46   }
47
48   kaapic_sync();
```

```
49
50    *ntasks += nnc+nnl;
51  }
52
53  void compute_floorplan (void)
54  {
55  [...]
56    int ntasks = 0;
57    kaapic_begin_parallel( KAAPIC_FLAG_DEFAULT );
58    kaapi_atomic_initlock(&kaapi_lock);
59    kaapic_spawn(0,5, add_cell,
60      KAAPIC_MODE_V, KAAPIC_TYPE_INT, 1, 1,
61      KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, footprint,
62      KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, board,
63      KAAPIC_MODE_V, KAAPIC_TYPE_PTR, 1, gcells,
64      KAAPIC_MODE_CW, KAAPIC_REDOP_PLUS, KAAPIC_TYPE_INT, 1, &
          ntasks
65        );
66    kaapic_sync();
67    kaapi_atomic_destroylock(&kaapi_lock);
68    kaapic_end_parallel( KAAPIC_FLAG_DEFAULT );
69  }
```

Código A.9: Floorplan Versão OpenMP

```
1  static int add_cell(int id, coor FOOTPRINT, ibrd BOARD, struct
       cell *CELLS) {
2  [...]
3
4    /* Para cada forma */
5    for (i = 0; i < CELLS[id].n; i++) {
6  [...]
7      /* Para cada localização */
8      for (j = 0; j < nn; j++) {
9  #pragma omp task untied private(board, footprint,area) \
10 firstprivate(NWS,i,j,id,nn) \
11 shared(FOOTPRINT,BOARD,CELLS,MIN_AREA,MIN_FOOTPRINT,N,BEST_BOARD
       ,nnc,bots_verbose_mode)
12 {
13         [...]
14         if (! lay_down(id, board, cells)) {
15    goto _end;
16         }
17
18 [...]
19         /* Se última célula */
20         if (cells[id].next == 0) {
21    if (area < MIN_AREA) {
22 #pragma omp critical
23      if (area < MIN_AREA) {
24        MIN_AREA          = area;
25        MIN_FOOTPRINT[0] = footprint[0];
```

```
26          MIN_FOOTPRINT[1] = footprint[1];
27          memcpy(BEST_BOARD, board, sizeof(ibrd));
28          bots_debug("N  %d\n", MIN_AREA);
29        }
30      }
31          } else if (area < MIN_AREA) {
32  #pragma omp atomic
33        nnc += add_cell(cells[id].next, footprint, board,cells);
34        } else { }
35  _end:;
36  }
37        }
38      }
39
40  #pragma omp taskwait
41  return nnc+nnl;
42  }
43
44  void compute_floorplan (void) {
45  [...]
46  #pragma omp parallel
47  #pragma omp single
48    bots_number_of_tasks = add_cell(1, footprint, board, gcells);
49  }
```

REFERÊNCIAS

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **J. ACM**, New York, NY, USA, v.46, p.720–748, 1999.

BOARD, O. A. R. **OpenMP Application Program Interface Version 3.0**. 2008.

BUTENHOF, D. R. **Programming with Posix Threads**. [S.l.]: Addison-Wesley, 1997.

CHAPMANM, B.; JOST, G.; PAS, R. v. d. **Using OpenMP**: portable shared memory parallel programming. [S.l.]: The MIT Press, 2008.

CORPORATION, I. **Intel threading building blocks**. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

CORPORATION, N. **NVIDIA CUDA C Programming Guide**. [S.l.: s.n.], 2012.

DONGARRA, J. et al. **Sourcebook of Parallel Computing**. 1.ed. [S.l.]: Morgan Kaufmann Publishers, 2003.

DURAN, A. et al. Barcelona OpenMP Tasks Suite: a set of benchmarks targeting the exploitation of task parallelism in openmp. In: PARALLEL PROCESSING, 2009. ICPP '09. INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2009. p.124–131.

FLYNN, M. Some Computer Organizations and Their Effectiveness. **Computers, IEEE Transactions on**, [S.l.], 1972.

FOSTER, I. **Designing and Building Parallel Programs**. USA: Addison-Wesley Publishing Company, 1995.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. **SIGPLAN Not.**, New York, NY, USA, p.212–223, 1998.

GALILEE, F. et al. Athapascan-1: on-line building data flow graph in a parallel language. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 1998. PROCEEDINGS. 1998 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 1998. p.88–95.

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, 2007., New York, NY, USA. **Proceedings...** ACM, 2007. p.15–23. (PASCO '07).

GAUTIER, T. et al. **X-Kaapi**: a multi paradigm runtime for multicore architectures. [S.l.]: INRIA, 2012. Rapport de recherche.

GAUTIER, T. et al. XKaapi: a runtime system for data-flow task programming on heterogeneous architectures. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS), 27., Boston, Massachusetts, États-Unis. **Anais...** [S.l.: s.n.], 2013.

GROUP, S. T. **Cilk 5.4.6 Reference Manual**. [S.l.]: Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 2001.

HERMANN, E.; RAFFIN, B.; FAURE, F. Interactive physical simulation on multicore architectures. In: EUROGRAPHICS CONFERENCE ON PARALLEL GRAPHICS AND VISUALIZATION, 9., Aire-la-Ville, Switzerland, Switzerland. **Proceedings...** Eurographics Association, 2009. p.1–8. (EG PGV'09).

LEMENTEC, F.; DANJEAN, V.; GAUTIER, T. **X-Kaapi C programming interface**. [S.l.]: INRIA, 2011. Rapport Technique. (RT-0417).

LIMA, J. V. F. et al. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD), New York, USA. **Anais...** IEEE, 2012.

MOORE, G. E. Cramming more components onto integrated circuits, Reprinted from Electronics. **Solid-State Circuits Society Newsletter, IEEE**, [S.l.], 2006.

OpenACC Working Group. **The OpenACC Application Programming Interface, Version 2.0**. 2013.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill, 2004.

RAUBER, T.; RUNGEL, G. **Parallel Programming**: for multicore and cluster systenms. [S.l.]: Springer-Verlag Berlin Heidelberg, 2010.

ROCH, J.-L.; REVIRE, R.; GAUTIER, T. **Athapascan : an api for asynchronous parallel programmingÂ user's guide**. [S.l.]: INRIA, 2003. Rapport de recherche.

SONG, F.; DONGARRA, J.; MOORE, S. Experiments with Strassen's Algorithm: from sequential to parallel. In: PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS - 2006. **Anais...** [S.l.: s.n.], 2006.

TCHIBOUKDJIAN, M.; GAST, N.; TRYSTRAM, D. Decentralized List Scheduling. **Annals of Operations Research**, [S.l.], 2012.