Um Auto-Escalonamento para Sistemas Multiprocessadores

Hsueh Tsung Hsiang
Líria Matsumoto Sato

Laboratório de Sistemas Integráveis
Departamento de Engenharia Elétrônica
Escola Politécnica da USP
Av. Luciano Gualberto, trav. 3, nº 158
05508-900 - São Paulo - SP
tel/fax: (011) 211-4574
e-mail: {hsueh, liria}@lsi.usp.br, hthsiang@fox.cce.usp.br

Resumo

O particionamento e o escalonamento de tarefas são tópicos muito importantes para programas paralelos por exercerem grande influência sobre o balanceamento de carga e consequentemente sobre a eficiência no uso dos processadores de um sistema multiprocessador. Este trabalho apresenta uma nova técnica de escalonamento de tarefas baseada no esquema de auto-escalonamento. Esta nova técnica busca obter um bom balanceamento de carga e a exploração de paralelismo aninhado em blocos ou laços paralelos.

Abstract

The partitioning and the scheduling of tasks are very important for parallel programs, since they exert great influence on the load balancing and therefore on the efficiency in the use of the processors from a multiprocessor system. So this paper presents a new technique for task scheduling based on the auto-scheduling scheme. This new technique tries to achieve a good load balancing and also explores the nested parallelisms found in parallel blocks and loops.

I. Introdução

À medida que as diversas áreas da ciência avançam, há um crescimento proporcional da demanda por maior capacidade de processamento computacional. Novas gerações de computadores foram desenvolvidas e construídas para satisfazer essa ânsia por elevados desempenhos. Com o avanço da tecnologia na área da microeletrônica, os circuitos eletrônicos foram miniaturizados até atingirem dimensões microscópicas. Desta forma, o processamento convencional von-Neumann praticamente está atingindo o limite superior da sua capacidade e qualquer melhoria para transpassar tal barreira só poderia ser obtida através de gastos desproporcionais de recursos humanos, tecnológicos ou financeiros.

Entretanto, este não é o único caminho para a obtenção de alto desempenho. Tendo o alto desempenho por alvo, diversas arquiteturas não-convencionais foram propostas. Existe um ponto em comum entre as diversas soluções alternativas: o processamento paralelo, ou seja, a execução simultânea de múltiplas computações. Assim, hoje são disponíveis várias arquiteturas de computadores paralelos, havendo entretanto um aproveitamento incompleto de todo o poder de processamento disponível. A maioria dos programas e algoritmos disponíveis é seqüencial, impedindo o aproveitamento efetivo das novas máquinas paralelas. Existem duas abordagens para a solução desse problema, a primeira propõe a programação direta em novas linguagens paralelas e a segunda faz uso de ferramentas como os compiladores paralelizantes ou reestruturadores.

Em ambas as abordagens, existe uma dificuldade comum para a obtenção de alto desempenho: a distribuição da carga de processamento entre os processadores. A fim de se ter uma elevada eficiência, ou seja, uma utilização efetiva de toda a capacidade de processamento de uma máquina paralela, é vital que a carga de processamento seja distribuída o mais igualmente possível entre os processadores e ao mesmo tempo mantendo-os ocupados a maior parte do tempo. Qualquer desvio dessa meta implica necessariamente numa perda de eficiência.

O problema da distribuição de carga de processamento é tratado pelo escalonamento. Na programação direta em linguagens paralelas, quem se preocupa com o escalonamento é geralmente o próprio programador. Mas, quando se tratam de compiladores paralelizantes, o problema passa a ser do próprio compilador.

Este trabalho apresenta uma nova técnica de particionamento e escalonamento baseada na técnica de auto-escalonamento, proposta por Polychronopoulos[7][8]. A técnica proposta, denominada Auto-Escalonamento Uniforme, permite um melhor balanceamento de carga em sistemas multiprocessadores e também explorar diretamente o paralelismo aninhado presente em blocos ou laços aninhados.

Nas seções a seguir, inicialmente apresenta-se rapidamente a estrutura de um compilador paralelizante e as técnicas de paralelização e de escalonamento mais comuns. Em seguida, apresenta-se a técnica do Auto-Escalonamento Uniforme em detalhes, mostrando-se um exemplo com as estimativas teóricas do speedup e da eficiência para três esquemas de auto-escalonamento. São comparados os esquemas de Auto-Escalonamento, Auto-Escalonamento Uniforme sem Prioridades e o Auto-Escalonamento Uniforme com Prioridades.

II. Técnicas de Paralelização e Escalonamento

O trabalho de um compilador paralelizante pode ser representado esquematicamente pela figura abaixo.

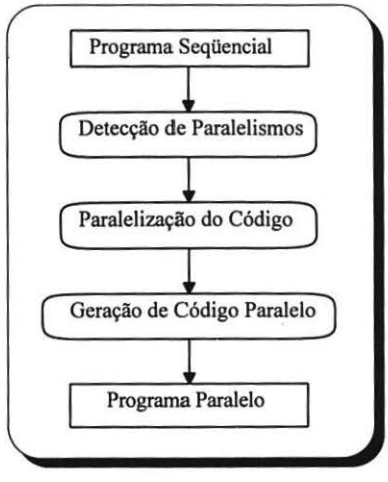

Fig. 1 - Esquema de um Compilador Paralelizante Genérico

A primeira fase de um compilador paralelizante é a de detecção de paralelismos. Em uma de suas etapas é realizada uma análise de fluxo de dados para a obtenção de informações a respeito das dependências de dados. Essa análise pode ser feita em cima do código fonte ou de alguma representação intermediária como no Parafrase-2[4]. Após a análise inicial para a obtenção das dependências de dados, o compilador pode aplicar uma série de transformações para eliminar algumas das dependências de dados, e conseqüentemente aumentar o grau de paralelismo máximo do programa[7].

A fase seguinte é a de paralelização do código, que pode ser subdividida em Particionamento e Escalonamento. A fase de particionamento é muito importante, pois ela está fortemente ligada à granularidade das tarefas distribuídas aos processadores. Na fase de escalonamento, decide-se para qual processador cada tarefa será atribuída.

Para colocar uma tarefa em execução, é necessário executar um código de inicialização e ao terminar a execução é executado um código de finalização. Estes trechos de código para inicialização e finalização constituem instruções que não fazem parte do programa do usuário. Assim, a sua execução representa um overhead no processamento do programa. Quanto menores as tarefas (granularidade menor), mais fácil é o trabalho do escalonador para a obtenção de um bom balanceamento de carga, mas, ao mesmo tempo tarefas menores implicam numa maior proporção de overhead para o corpo da tarefa. Se as tarefas forem maiores, melhora-se a proporção de overhead para código útil, mas, dificulta-se proporcionalmente a obtenção de um bom balanceamento de carga, pois como as tarefas são maiores, os seus tempos de execução tendem a ser mais desiguais.

Existem dois tipos básicos de esquemas de particionamento e escalonamento de um programa paralelo: estático ou dinâmico. Nos esquemas estáticos, o particionamento e o escalonamento são feitos em tempo de compilação, definindo-se a priori quais são as tarefas e também quais processadores executarão cada uma delas. Os esquemas estáticos possuem a grande vantagem da simplicidade de codificação e normalmente menor overhead em tempo de execução, sacrificando um grau maior de paralelismo ou um melhor balanceamento de carga, devido à falta de conhecimento pelo compilador de certas restrições que somente são determinadas em tempo de execução. Nos esquemas dinâmicos, as fases de particionamento e escalonamento são adiadas até o momento da execução, permitindo a otimização das mesmas. Este tipo de esquema permite aproveitar ao máximo o paralelismo existente, e também realizar

um balanceamento de carga sob medida para a situação sob a qual o programa está executando, porém, também possui a desvantagem de apresentar overhead muito elevado em tempo de execução, o que inviabiliza a sua aplicação na maioria dos casos, quando não há suporte em hardware.

A solução mais viável a curto prazo é utilizar esquemas mistos, como:

- códigos com múltiplas versões: algumas decisões mais críticas são deixadas para serem tomadas em tempo de execução, gerando-se código correspondente a cada uma das decisões possíveis[4].
- auto-escalonamento: o particionamento é feito parcialmente em tempo de compilação e o escalonamento ocorre em tempo de execução[7][8].

O esquema de múltiplas versões permite melhorar o aproveitamento da máquina paralela, deixando para o tempo de execução algumas decisões, a respeito da melhor forma de se particionar ou escalonar determinados trechos do programa. Desta forma, torna-se possível a aplicação de versões de códigos bastante otimizadas para cada uma das situações, e com overhead reduzido. Mas, o esquema apresenta a desvantagem de necessitar da duplicação de trechos de código, pois para cada decisão adiada haverá vários trechos correspondentes aos caminhos a serem tomados.

O esquema de auto-escalonamento, proposto por Polychronopoulos[7][8], tenta resolver dois problemas importantes:

- o controle dinâmico do particionamento de programas
- a exploração de paralelismo não estruturado

O esquema é apresentado em mais detalhes na subseção B. A seguir são apresentadas técnicas de particionamento dinâmico para laços e para tarefas.

A. Técnicas de Particionamento e Escalonamento para Laços

As técnicas tradicionais são geralmente direcionadas para a execução de laços paralelos. São apresentadas, a seguir, de forma sucinta alguns dos esquemas mais conhecidos.

1. O Algoritmo Self-Scheduling - SS

É o esquema mais simples de escalonamento[7]. Cada processador recebe uma única iteração do laço, decrementando o contador de iterações até que todo o laço tenha sido executado. O esquema oferece o melhor balanceamento de carga, pois garante-se que todos os processadores terminarão o laço com uma defasagem máxima igual ao tempo da última iteração. O esquema também resulta em grande overhead, pois para cada vez que um processador realiza uma busca de iteração, o contador de iterações precisa ser decrementado usando acesso exclusivo à memória.

2. O Algoritmo Chunk Self-Scheduling - CSS(k)

O esquema CSS(k) é similar ao SS, com a diferença que cada processador recebe k iterações ao invés de uma iteração[7]. O esquema tenta reduzir assim o número de acessos exclusivos à memória, pois para um laço com n iterações haverá $\lceil \frac{n}{k} \rceil$ acessos exclusivos à memória contra n no caso do SS. Mas o CSS(k) piora significativamente o balanceamento de carga, pois agora a defasagem máxima entre os tempos de execução dos processadores pode aumentar até o tempo de execução do último bloco de iterações.

3. O Algoritmo Guided Self-Scheduling - GSS(k)

O GSS(k) combina as vantagens de ambos os algoritmos anteriores. Cada processador recebe um conjunto de iterações (chunk), cujo tamanho é determinado dinamicamente proporcionalmente ao número de iterações por executar e inversamente proporcional ao número de processadores que executam o laço[7]. Na i-ésima iteração, cada processador recebe $\lceil \frac{R_i}{p} \rceil$ iterações, onde R_i é o número de iterações que restam na i-ésima iteração e p é o número de processadores. Como no CSS(k), o parâmetro k representa o limite inferior para o número de iterações em um conjunto de iterações.

O GSS(k) oferece overhead reduzido no início da execução de um laço através de conjuntos de iterações maiores, ou seja, são economizados acessos exclusivos à memória. E, ao contrário do CSS(k), o tamanho dos conjuntos de iterações é reduzido exponencialmente até atingir k iterações. Os conjuntos de iterações menores no final da execução do laço contribuem para o balanceamento de carga. O valor ideal para k é

dependente do sistema e deve ser tal que haja um compromisso entre o overhead correspondente a conjuntos de iterações com tamanho k e a qualidade do balanceamento de carga.

4. O Algoritmo Trapezoid Self-Scheduling - TSS(f,l)

O algoritmo de TSS(f,l), assim como o GSS(k) descrito anteriormente, também utiliza conjuntos de iterações com tamanho variável. O primeiro conjunto de iterações atribuido a um processador tem tamanho f e o último, tamanho l. Ao contrário do GSS(k), cujos tamanhos de conjuntos de iterações são reduzidos exponencialmente, o TSS(f,l) reduz o tamanho dos conjuntos de forma linear. Isso reduz o número de conjuntos de iterações com tamanho reduzido no final da execução do laço, mas, mantendo o bom balanceamento de carga[10].

B. Técnicas de Particionamento e Escalonamento de Tarefas

As técnicas de particionamento e escalonamento de tarefas tentam explorar o paralelismo que não se apresenta na forma de laços paralelos, chamado de paralelismo não-estruturado ou funcional[3][5].

1. O Auto-Escalonamento (AS - Auto-Scheduling)

O Auto-Escalonamento, proposto por Polychronopoulos[5], é um esquema de esquema que aproveita os conceitos do modelo de fluxo de dados (Dataflow), mas adiciona outros conceitos que viabilizam a exploração do paralelismo funcional[7][8][9].

O Auto-Escalonamento utiliza o Grafo Hierárquico de Tarefas[2][5][8] (HTG - Hierarchical Task Graph) de um programa como controle para a execução das tarefas de que o programa é composto. O AS permite aproveitar o paralelismo existente em todos os níveis, tornando possível executar o programa eficientemente em ambientes multiprogramados e com quantidade variável de processadores.

Em tempo de compilação, o programa é dividido em tarefas que serão as unidades básicas de execução para o esquema AS. No início e no final de cada tarefa são

embutidos os códigos de inicialização e de finalização, respectivamente. Estes códigos são responsáveis por ações que normalmente cabem ao sistema operacional, mas a sua implementação a nível de programa-usuário reduz bastante o overhead associado ao gerenciamento das tarefas. Com a redução do overhead, torna-se viável o escalonamento de tarefas tão pequenas quanto um bloco básico (seqüência de instruções cuja execução sempre inicia na primeira instrução e se contiver um desvio, este deverá ser a última instrução [1]).

Um programa auto-escalonado é composto basicamente de uma Fila de Tarefas Habilitadas (ETQ - Enabled Task Queue) e de um HTG instrumentado, isto é, os seus nós contém imagens executáveis das tarefas já com os códigos de inicialização e de finalização.

Os processadores que executam um programa são os disponíveis na máquina no instante da execução. Assim, o número de processadores que participam da execução pode variar durante a execução, como é típico em ambientes multiprogramados, onde o término da execução de um outro programa aumenta o número de processadores livres na máquina.

A ETQ é uma fila onde estão os blocos identificadores das tarefas prontas para execução. Os blocos identificadores das tarefas contêm informação suficiente a respeito do contexto e do código da tarefa, de modo que qualquer processador livre possa executá-la[5].

Cada um dos processadores que alocados para o programa executa um laço como o abaixo:

```
do forever
   busca um bloco identificador de tarefa na ETQ
   executa a tarefa correspondente
```

A pilha do sistema usada normalmente para programas seqüenciais não é suficiente para a implementação das estruturas para os frames de ativação de um programa paralelo. Uma estrutura mais adequada é a cactus-stack[5].

Os laços paralelos possuem uma política diferenciada de escalonamento, sendo que as tarefas a eles correspondentes podem conter uma única iteração, se o escalonamento para laços é do tipo SS, ou podem conter um conjunto de iterações, se o escalonamento para laços é do tipo CSS(k), GSS(k) ou TSS(f,l). Nos casos de escalonamento do tipo GSS(k) ou TSS(f,l), as tarefas possuem granularidade decrescente à medida que o laço é executado, sendo exponencialmente para o GSS(k) e linearmente para o TSS(f,l).

O HTG instrumentado de um programa paralelo, antes da geração de código, normalmente passa por diversas etapas de otimização de modo a reduzir o número de arcos de dependência e também diminuir o número de sincronizações e acessos exclusivos à memória contidos nos códigos de inicialização e de finalização das tarefas.

2. O Auto-Escalonamento Uniforme (UAS - Uniform Auto-Scheduling)

O Auto-Escalonamento Uniforme, proposto neste artigo, é baseado no Auto-Escalonamento descrito no item anterior. As principais diferenças estão na forma de tratamento dado às unidades básicas de execução: tarefas no AS e macrotarefas no UAS, e na granularidade das mesmas.

No esquema proposto por Polychronopoulos, um processador ao realizar uma busca na ETQ recebe uma tarefa que pode conter um trecho de código seqüencial ou um conjunto de iterações de um mesmo laço paralelo. Neste caso, há uma distinção entre tarefas correspondentes a conjuntos de iterações de laços e tarefas genéricas.

No Auto-Escalonamento Uniforme cada unidade básica de execução aqui especificada como macrotarefa, é um conjunto de tarefas, onde cada uma pode ser um trcho seqüencial ou uma iteração de um laço. Quando um processador livre busca um macrotarfea na ETV, não faz distinção entre os seus elementos, havendo portanto uma uniformidade no tratamento dado a tarefas genéricas e iterações de dados. As macrotarefas recebidas pelos processadores podem conter código de diferentes nós do HTG.

Uma descrição mais detalhada da técnica é dada na seção III.

III.O Auto-Escalonamento Uniforme

A. Descrição

O esquema de Auto-Escalonamento Uniforme (UAS) é muito similar ao esquema de Auto-Escalonamento proposto por Polychronopoulos. O UAS tenta resolver algumas deficiências referentes ao balanceamento de carga que restringem o desempenho de um programa paralelo com AS.

Um programa paralelo com UAS é composto basicamente de um HTG instrumentado e de um Vetor de Tarefas Habilitadas (ETV - Enabled Task Vector), no lugar da ETQ no Auto-Escalonamento.

O ETV é uma estrutura de dados similar a uma fila circular mas acessível randomicamente como um vetor. O ETV contém os blocos identificadores das tarefas habilitadas para execução em ordem decrescente de prioridade. A prioridade de uma tarefa habilitada é determinada através de heurísticas que dependem de diversos fatores que podem ser estimados em tempo de compilação. Exemplos:

- tarefas possuem prioridade proporcional ao maior dos tempos de execução de seus sucessores.
- tarefas com maior tempo de execução estimado possuem maior prioridade.
- tarefas que não fazem parte de blocos paralelos possuem maior prioridade do que outras que fazem parte de blocos paralelos.

Enfatiza-se que os exemplos dados são heurísticas e podem não resultar na melhor escolha da prioridade em todos as situações possíveis. Isso ocorre porque a análise feita para se obter a prioridade só leva em conta o nó a ela correspondente no HTG e seus sucessores.

Como no esquema AS, os processadores executam um laço como o mostrado abaixo:

```
do forever
   busca um macrotarefa no ETV
   executa a macrotarefa
```

O conteúdo da macrotarefa recebida por um processador é definido através da aplicação de algoritmos equivalentes àqueles aplicados a laços: SS, CSS(k), GSS(k) ou TSS(f,l). O conteúdo do ETV é considerado como se fossem iterações de um laço paralelo.

Os algoritmos SS e CSS(k) aplicados a laços podem ser utilizados sem modificações. Assim, se o algoritmo adotado for o SS, então cada macrotarefa conterá a tarefa mais prioritária no ETV e se for adotado o CSS(k), cada macrotarefa conterá as k tarefas mais prioritárias.

Tanto no caso do algoritmo GSS(k) como no caso do TSS(f,l), são necessárias adaptações dos algoritmos originais aplicados a laços. No caso do algoritmo GSS(k) modificado, cada macrotarefa conterá $\lceil \frac{n_i}{p_i} \rceil$ tarefas, onde n_i é o número de tarefas no ETV no instante t_i e p_i o número de processadores que participam da execução do programa no mesmo instante t_i. No caso do algoritmo TSS(f,l) modificado, o algoritmo comporta-se como o TSS(f,l) enquanto não há novas tarefas habilitadas e cada vez que uma nova tarefa é habilitada, o escalonamento é reiniciado, de modo que a próxima macrotarefa alocada conterá f tarefas.

B. Exemplo de Análise de Desempenho

O exemplo a ser analisado corresponde ao trecho de código em linguagem C abaixo:

```
a=f1();                          /* tarefa A */
b=f2(a);                         /* tarefa B */
c=f3( a )*f3(a*a);               /* tarefa C */
for(i=0;i<45;i++ )               /* tarefa D */
   d[i]=f4(b,i);
e=f5(b,c);                       /* tarefa E */
for(j=0;j<20;j++)                /* tarefa F */
   f[j]=f6(e,j);
for(k=0;k<3;k++)                 /* tarefa G */
   g[k]=f7(d[k]);
h = f8(f[2],g[3]);               /* tarefa H */
```

Através dos comentários está indicado qual o particionamento adotado. Os três laços são paralelizáveis. Considere-se que as funções utilizadas f1 a f8 não utilizam variáveis

globais e que possuam aproximadamente o mesmo tempo de execução para facilitar os cálculos. Considere-se também que o sistema multiprocessador possua 10 processadores disponíveis para a execução do programa. Abaixo está mostrado o grafo de tarefas para o trecho de código em questão.

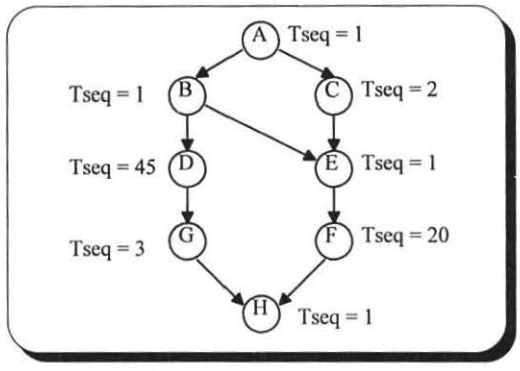

Fig. 2 - Grafo de Tarefas

Foi feita uma simulação da execução do trecho sob três esquemas diferentes: o Auto-Escalonamento (AS), o Auto-Escalonamento Uniforme sem Prioridades (UAS-NP) e o Auto-Escalonamento Uniforme com Prioridades (UAS-P). Os resultados são apresentados a seguir na forma de tabelas.

Notação:

- X(i,j): representa as iterações de i até j de um laço correspondente à tarefa X
- t: tempo em unidade arbitrária
- P_i: representa o i-ésimo processador do sistema $i = 0, 1, \ldots 9$.
- ETQ: Fila de Tarefas Habilitadas(Tam. ETQ é o número de elementos na fila)
- ETV: Vetor de Tarefas Habilitadas (Tam. ETV é o número de tarefas no vetor)
- Proc. Ocup.: número de processadores ocupados
- Proc. Livres: número de processadores livres.

Tabela 1 - Auto-Escalonamento

t	P_8	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	ETQ	Tam. ETQ	Proc. Ocup.	Proc. Livres
0											A	1	0	10
1	A										B C	2	1	9
2	B	C									D(0,44)	45	2	8
3	D(0,4)	C	D(5,9)	D(10,13)	D(14,17)	D(18,20)	D(21,23)	D(24,26)	D(27,28)	D(29,30)	D(31,44) E	15	10	0
4	D(1,4)	E	D(6,9)	D(11,13)	D(15,17)	D(19,20)	D(22,23)	D(25,26)	D(28,28)	D(30,30)	D(31,44) F(0,19)	34	10	0
5	D(2,4)	F(0,0)	D(7,9)	D(12,13)	D(16,17)	D(20,20)	D(23,23)	D(26,26)	D(31,32)	D(33,34)	D(35,44) F(1,19)	29	10	0
6	D(3,4)	F(1,1)	D(8,9)	D(13,13)	D(17,17)	D(35,36)	D(37,37)	D(38,38)	D(32,32)	D(34,34)	D(40,44) F(2,19)	23	10	0
7	D(4,4)	F(2,2)	D(9,9)	D(39,39)	D(40,40)	D(36,36)	D(43,43)	D(44,44)	D(41,41)	D(42,42)	F(3,19) G(0,2)	17	10	0
8	G(0,0)	F(3,3)	G(1,1)	G(2,2)							F(4,19)	16	1	9
9		F(4,4)									F(5,19)	15	1	9
23		F(18,18)									F(19,19)	1	1	9
24		F(19,19)									H	1	1	9
25	H											0	1	9

Tabela 2 - Auto-Escalonamento Uniforme sem Prioridades

t	P_8	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	ETV	Tam. ETV	Proc. Ocup.	Proc. Livres
0											A	1	0	10
1	A										B C	2	1	9
2	B	C									D(0,44)	45	2	8
3	D(0,4)	C	D(5,8)	D(9,12)	D(13,16)	D(17,19)	D(20,22)	D(23,25)	D(26,27)	D(28,29)	D(30,44) E	16	10	0
4	D(1,4)	D(30,31)	D(6,8)	D(10,12)	D(14,16)	D(18,19)	D(21,22)	D(24,25)	D(27,27)	D(29,29)	D(32,44) E	14	10	0
5	D(2,4)	D(31,31)	D(7,8)	D(11,12)	D(15,16)	D(19,19)	D(22,22)	D(25,25)	D(32,33)	D(34,35)	D(36,44) E	10	10	0
6	D(3,4)	F(39,39)	D(8,8)	D(12,12)	D(16,16)	D(36,36)	D(37,37)	D(38,38)	D(33,33)	D(35,35)	D(40,44) E	6	10	0
7	D(4,4)	F(40,40)	D(9,9)	D(42,42)	D(43,43)	D(44,44)	E				F(0,19) G(0,2)	23	7	3
8	F(0,2)	F(3,4)	F(5,6)	F(7,8)	F(9,10)	F(11,12)	F(13,13)	F(14,14)	F(15,15)	F(16,16)	F(17,19) G(0,2)	6	10	0
9	F(1,2)	F(4,4)	F(6,6)	F(8,8)	F(10,10)	F(12,12)	F(17,17)	F(18,18)	F(19,19)	G(0,0)	G(1,2)	2	10	0
10	F(2,2)	G(1,1)	G(2,2)								H	1	3	7
11	H											0	1	9

Tabela 3 - Auto-Escalonamento Uniforme com Prioridades

t	P_8	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	ETV	Tam. ETV	Proc. Ocup.	Proc. Livres
0											A	1	0	10
1	A										B C	2	1	9
2	B	C									D(0,44)	45	2	8
3	D(0,4)	C	D(5,8)	D(9,12)	D(13,16)	D(17,19)	D(20,22)	D(23,25)	D(26,27)	D(28,29)	E D(30,44)	16	10	0
4	D(1,4)	E D(30,30)	D(6,8)	D(10,12)	D(14,16)	D(18,19)	D(21,22)	D(24,25)	D(27,27)	D(29,29)	D(31,44) F(0,19)	34	10	0
5	D(2,4)	D(30,30)	D(7,8)	D(11,12)	D(15,16)	D(19,19)	D(22,22)	D(25,25)	D(31,34)	D(35,37)	D(38,44) F(0,19)	27	10	0
6	D(3,4)	F(38,40)	D(8,8)	D(12,12)	D(16,16)	D(41,43)	D(44,44) F(0,1)	F(2,3)	D(32,34)	D(36,37)	F(4,19)	16	10	0
7	D(4,4)	F(39,40)	F(4,5)	F(6,7)	F(8,9)	D(42,43)	F(0,1)	F(3,3)	D(33,34)	D(37,37)	F(10,19)	10	10	0
8	F(10,10)	F(40,40)	F(5,5)	F(7,7)	F(9,10)	F(43,43)	F(1,1)	F(14,14)	D(34,34)	F(12,12)	F(13,19) G(0,2)	10	10	0
9	F(13,13)	F(14,14)	F(15,15)	F(16,16)	F(17,17)	F(18,18)	F(19,19)	G(0,0)	G(1,1)	G(2,2)	H	1	10	0
10	H											0	1	9

As tabelas mostram qual tarefa terminou de ser executada em cada instante de tempo. Além disso, as tabelas mostram também o número de processadores envolvidos na execução do trecho de código. No esquema AS, o algoritmo usado para escalonar os laços foi o GSS(1), e nos esquemas UAS-NP e UAS-P, utilizou-se o GSS(1) modificado.

Observando-se a primeira tabela correspondente ao esquema AS, nota-se que no intervalo de 2 a 7 unidades de tempo, todos os 10 processadores da máquina estavam ocupados, demonstrando um bom balanceamento. Porém, a partir do instante 8, o laço paralelo da tarefa F manteve um único processador para o qual havia sido escalonado. Isso ocorreu porque no instante em que a tarefa F foi escalonada, todos os outros processadores estavam executado o laço da tarefa D. Assim, 9 processadores não participaram da execução do programa durante 17 unidades de tempo. O balanceamento de carga é, então, péssimo durante a maior parte do tempo de execução do programa.

No esquema UAS-NP, observa-se que o balanceamento de carga é nitidamente melhor , sendo fraco apenas no início e no final da execução do programa, situações em que não há ainda tarefas habilitadas o suficiente ou não há mais tarefas a serem habilitadas. Nota-se também que a não utilização de um esquema com prioridades criou uma situação de insuficiência de tarefas habilitadas por volta do instante 7. Este problema teria sido evitado se a tarefa E tivesse sido executada antes. E é exatamente isso que ocorre no caso do esquema UAS-P, como se pode ver na tabela 3. A tarefa E já estava pronta para execução no final do instante 3 e como era mais prioritária que a tarefa D, ela é escalonada na macrotarefa imediatamente seguinte.

Pode-se calcular o speedup e o índice de eficiência para se ter uma comparação quantitativa das 3 técnicas de auto-escalonamento:

Seqüencial: $T_{seq} = \Sigma_{k=A}^{H} T_k = 1 + 1 + 2 + 45 \times 1 + 1 + 20 \times 1 + 3 \times 1 + 1 = 74$

AS: $T_{AS} = 25 \Rightarrow speedup = \frac{T_{seq}}{T_{AS}} = \frac{74}{25} = 2,96 \Rightarrow eficiência = \frac{speedup}{p} = \frac{2,96}{10} = 29,6\%$

UAS-NP: $T_{UAS\text{-}NP} = 11 \Rightarrow speedup = \frac{74}{11} = 6,73 \Rightarrow eficiência = \frac{6,73}{10} = 67,3\%$

UAS-P: $T_{UAS\text{-}P} = 10 \Rightarrow speedup = \frac{74}{10} = 7,4 \Rightarrow eficiência = \frac{7,4}{10} = 74\%$

Observe-se que as técnicas UAS-NP e UAS-P fornecem resultados próximos, e realmente o uso do esquema com prioridades poderia ser substituído por uma análise mais rigorosa no momento de particionamento do programa em tarefas, detectando-se esses problemas e aplicando-se técnicas nas quais se adia o processamento de algumas tarefas. No exemplo dado, a execução da tarefa D deveria ser atrasada até que se inicie a execução de E. Entretanto, o esquema de prioridade é mais simples de ser implementado e não eleva significativamente o overhead já representado pelos códigos de inicialização e de finalização das tarefas.

IV.Conclusão

Este trabalho apresentou um novo esquema de auto-escalonamento, o Auto-Escalonamento Uniforme, no qual tenta-se resolver os problemas que impedem um programa com auto-escalonamento tradicional[8] de atingir melhores índices de eficiência ou melhores valores para o speedup em sistemas multiprocessadores.

A base da solução consiste na melhoria do balanceamento de carga nos processadores. Outra vantagem da técnica é a exploração do paralelismo aninhado, como aquele presente em laços ou blocos paralelos aninhados.

V. Bibliografia

1. AHO, A.V., SETHI, R., ULLMAN, J. D. **Compiler-principles, techniques and tools.** Addison Wesley, 1986.

2. GIRKAR, M., POLYCHRONOPOULOS, C. D. **The HTG: an intermediate representation for programs based on Control and Data Dependences.** Technical Report 1046, CSRD, University of Illinois at Urbana-Champaign, Maio 1991.

3. GIRKAR, M., POLYCHRONOPOULOS, C. D. **Automatic extraction of funcional paralelism from ordinary programas,** IEEE Transaction on Parallel and Distributed Systems, vol. 3, no. 2, p. 166-178, Mar. 1992.

4. LEUNG, B. P. **Issues on the design of parallelizing compilers.** Master Thesis, University of Illinois at Urbana-Champaign, 1990.

5. MOREIRA, J. D., **Parallel processing: architecture and programming. Em: Anais da II Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho.** vol. 2, 1992.

6. POLYCHRONOPOULOS, C. D., GIRKAR, M. HAGHIGHAT, M. R., LEE, C. L., LEUNG, B. P., SCHOUTEN, D. A., **The structure of parafrase-2: an advanced parallelizing compiler for C and Fortran.** In: GELERNTER, D. et al., eds. **Languages and compilers for parallel computing.** MIT Press; 1990, p. 423-453.

7. POLYCHRONOPOULOS, C. D. **Parallel programming and compilers.** Kluwer Academic Publishers, 240 p., 1988.

8. POLYCHRONOPOULOS, C. D. **Auto-scheduling: control flow and data flow come together.** Technical Report 1088, CSRD, University of Illinois at Urbana-Champaign, 1991.

9. POLYCHROPOULOS, C. D., KUCK, D., **Guided self-scheduling: a practical scheduling scheme for parallel supercomputers.** IEEE Transaction on Computers, vol. C-36, no. 12, p. 1425-1439, Dez. 1987.

10. TZEN, J. TEN, NI, L. M., **Trapezoid self-scheduling: a practical scheduling for paralle compilers.** IEEE Transaction on Parallel and Distributed Systems, vol. 4, no. 1, p. 87-98, 1993.