細粒度並列計算機用最適化コンパイラ:OP.1

稲垣 達氏　松本 尚　平木 敬

東京大学理学部情報科学科

細粒度並列処理を行なう場合、要素プロセッサの高速化に伴い通信と同期のコストを考慮することが重要になる。本稿では手続き型プログラムの基本ブロック内の命令レベルの細粒度並列処理において通信と同期のオーバーヘッドを軽減するコンパイル手法を述べ、細粒度並列計算機用最適化コンパイラ OP.1 に実装した結果について報告する。OP.1 は通信の最適化手法として先行タスクの複製を行なうタスクスケジューリングアルゴリズムである DSH (Duplication Scheduling Heuristic) を採用し、同期コストの低い Elastic Barrier を用いた同期コードの生成を行なう。

OP.1:An Optimizing Parallelizer for Fine-Grain Multiprocessors

Tatsushi INAGAKI　Takashi MATSUMOTO　Kei HIRAKI

Department of Information Science, Faculty of Science, the University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113, Japan

On fine-grain parallel processing, as processor elements become faster, to consider communication and synchronization overheads becomes more important. This paper describes compilation techniques to reduce communication and synchronization costs on fine-grain parallel processing at the instruction level in basic blocks of procedural languages, and presents implementation results of these techniques on an optimizing parallelizer(OP.1) for fine-grain multiprocessors. OP.1 adopts DSH(Duplication Scheduling Heuristic), a scheduling heuristic which duplicates preceding tasks to optimize communication overheads. OP.1 generates low cost synchronization codes which use Elastic Barrier mechanism.

1 はじめに

機械語命令レベルの並列性を利用した細粒度並列処理を行なう場合、演算器の高速化に伴い細粒度での同期や通信のコストが相対的に大きくなる。これらに対してハードウェアによる高速なバリア同期機構 [10] やプロセッサ間通信機構 [8] を用いることで同期や通信のオーバーヘッドを削減もしくは隠蔽することが可能であるが、細粒度並列処理においてその効果を発揮させるには、静的なスケジューリングによって同期命令の発行やデータ転送のタイミングの最適化を行なうことが重要である。

本稿では、プログラムの基本ブロック及びトレース内の演算レベルの並列性を静的タスクスケジューリングによって利用する細粒度並列処理について述べる。我々はこのような細粒度並列処理を実現する最適化コンパイラ OP.1 (Optimizing Parallelizer 1) を製作している。以下では OP.1 の通信を考慮したタスクスケジューリング技法と、同期のオーバーヘッドを抑えるコード生成法について述べ、シミュレータ上での生成コードの評価を行なう。

2 システムの概要

2.1 目的機械

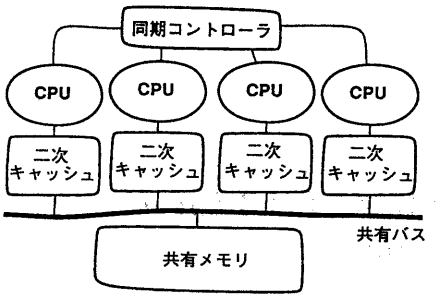


図 1: 目的機械のアーキテクチャ

OP.1 が対象とするアーキテクチャは密結合共有メモリ型の MIMD 型並列計算機である (図 1)。要素プロセッサは大容量のレジスタを持ったワークステーションクラスの RISC プロセッサを仮定しており、invalidate 及び update 系のプロトコルを備えた共有バスによるスヌープキャッシュで互いに高速に通信することが出来る。また、同期機構として elastic barrier [10] を持ち、バリア間の順序が狂わなければ殆んどオーバーヘッドのない同期が可能である。

OP.1 は、スヌープキャッシュを備えた MIMD 型並列計算機に低コストの同期機構を付加したアーキテクチャを用いて、VLIW 計算機で行なわれているような細粒度並列処理を実現することを目標としている。OSCAR[13] のように細粒度では全てのコストを静的に決定できるという前提をおき、同期を極力排除した並列処理を行なうことが可能なアーキテクチャもあるが、プロセッサの演算速度を向上させるとそれに従いデータの転送速度がボトルネックとなりやすく、キャッシュのミスヒットやバスのコンテンション等の動的な要素が細粒度にも現れてくると考えられる。OP.1 の目的機械は MIMD 計算機であるため VLIW 計算機より動的な外乱のオーバーヘッドを吸収する能力が高く、また、軽い同期機構を用いているためて同期命令のタイミングの乱れによって発生するオーバーヘッドも少ない。

2.2 OP.1 の構成

OP.1 のコンパイル過程は大まかに次の五つに分けられる。

1. ソースプログラムの構文解析

   入力となる原始言語は構造として配列を持つ簡単な手続き型言語である。ポインタは持たないので、スカラ変数の依存関係は依存解析によって容易にデータフローグラフに変換することができる。

2. 中間コードの生成

   原始言語を single thread の中間言語に変換する。中間言語の基本演算子はスケジューリングの対象となるタスクに相当している。無限個のレジスタを持ったシングルプロセッサのアセンブリ言語にほぼ等しい。

3. フローグラフの生成

   中間言語で表されたプログラムを意味的に等価なフローグラフに変換する。フローグラフの構造は細粒度のデータフローグラフを制御を表すマクロブロックによってクラスタ化したものである。この変換は本質的に新たな情報を付け加えるものではないが、原始言語のインターフェースとしてのフロントエンドと、スケジューリング等の最適化を行なうバックエンドを切り離す意味を持つ。

4. スケジューリング

   アーキテクチャのパラメータを元にして通信、同期、資源制約を考慮した静的なタスクスケジューリングを行なう。

5. 同期コードの挿入及びレジスタの割り付け

   スケジューリングされたタスクグラフに対して、先行関係のクラスタ化、冗長な先行関係の除去等を後処理として行ない、バリア同期をのオーバーヘッドを出来るだけ少なくする。並行して各プロセッサ内のレジスタを生存期間解析によって割り付ける。

3 バックエンド部の最適化手法

コンパイラのバックエンド部の目的はタスクグラフを完全結合のプロセッサグラフにマッピングして dominant sequence [3] (スケジューリングされたタスクグラフの critical path) の長さを最小化することである。制約として次のようなものが考えられる。

1. プロセッサ間の通信コスト

   タスクグラフのエッジのコストが同一プロセッサ内で 0 と見做せることに対応。

2. 演算パイプラインの structural hazard

   プロセッサ内にマップされたノードに対する資源制約。

3. レジスタの個数による制約

   プロセッサ内にマップされたエッジに対する資源制約。

4. プロセッサ間の同期コスト

   プロセッサ間にマップされた複数のエッジに対応する制約。

他にもキャッシュ上の制約等が考えられるがここでは考慮に入れない。OP.1では制約1と制約2をタスクスケジューリング時に考慮し、制約3と制約4がなるべくスケジューリング結果に影響を与えないようにスケジューリング後のマッピングを行なう。

3.1 フローグラフ

バックエンド部の入力となるフローグラフについて説明する。フローグラフのノードは中間言語のニモニックに対応しており、次節のスケジューリングの対象となるタスクである。ノードには副作用を持つもの(loadとstoreに対応するノード)があり、このため先行関係を表すエッジにはデータ依存関係を表すものと制御依存関係を表すものとがある。フローグラフはこれらのノードとエッジをさらにループと条件分岐の二つの構造で階層化したものである。

3.2 通信のオーバーヘッドを考慮したスケジューリング

ヒューリスティックな静的タスクスケジューリングアルゴリズムに通信のコストを導入する試みは数多く成されているが[3, 6]、ここではB. Kruatrachueの DSH(Duplication Scheduling Heuristic)[5]を用いる。DSHは代表的なリストスケジューリングアルゴリズムであるCP/MISF法[4]に疑似的なバックトラッキングを導入し、プロセッサの空きスロットに先行タスクを複製する手法である(図2)。[3]等で議論されているクラスタリングの手法に比べて以下のような利点がある。

- スケジューリング中にタスクグラフの変更を行なうので、タスクグラフを固定したクラスタリングよりdominant sequence長をより短くできる可能性がある。
- タスクの配置をgreedyに行なうので同じタスクを何度も配置し直す手間が要らない。従って他のgreedyな最適化手法もあまり計算量を増加させずに導入することが出来る。

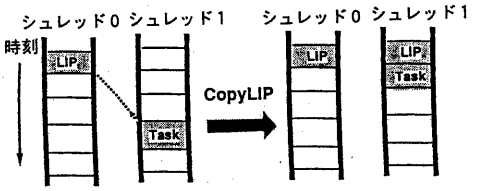


図2: DSHの概念図

DSHの手順を簡単に記述すると以下のようになる。

手続き名: TaskDuplicationProcess
入力: Task, Shred
出力: StartTime, CopyTaskList

1. CopyTaskListを空にする。
2. TaskStartTimeを用いてTaskのShredでの開始時刻を求め、StartTimeに代入する。
3. Taskがデータの依存関係を持っている先行タスクで、他のシュレッドに配置されているためTaskの開始時刻を遅らせているもの(LIP, last immediate predecessor)を探す。なければCopyTaskListの中のタスクについてLIPを探す。
4. CopyLIPを用いて3で探したLIPのShredへの複製を試みる。複製が成功したらTaskStartTimeを用いて複製したタスクとCopyTaskListとTaskをスケジュールし直し、Taskの開始時刻が早くなればStartTimeとCopyTaskListを更新する。
5. 複製が失敗したら終了。成功すれば3から再び繰り返す。

手続き名: CopyLIP
入力: LIP, Shred, StartTimeLimit
出力: Copy, CopyTaskList

1. LIPの開始時刻をTaskStartTimeを用いて求める。
2. StartTimeLimitより先に配置できれば成功。
   CopyTaskListを更新する
3. 配置できなかったら、CopyLIPを再帰的に用いてLIPのLIPを同じように複製する。複製が成功したらCopyTaskListを更新して返す。複製が失敗したらLIPの複製も失敗と見做す。

スケジューリングの手順はトレース内のタスクをCP/MISF法と同様にレベル付けして、優先度の高いタスクから順にTaskDuplicationProcessを用いて最も早い開始時刻を与えるシュレッドに配置していくことで実現される。DSHはDAGを扱うアルゴリズムであり、条件分岐のない基本ブロックにはナイーブに適用することができるが、基本ブロック内のDAGのサイズは一般にはあまり大きくないのでスケジューリングの効果が期待できない。OP.1ではVLIW計算機のコンパイラ等で採用されているtrace scheduling[1]の概念を援用して、ループ内のDAGのサイズを大きくしている。現在の実現ではトレースの方向はプログラマが文中に指示を入れることで決定している。

与えられたシュレッド内でのタスクの開始時刻を求める手続きTaskStartTimeはプロセッサ内のパイプラインによる制約と、スヌープキャッシュを通じた通信のコストを考慮して、与えられたタスクのできるだけ早い開始時刻を求める。

プロセッサの演算パイプラインによる制約はタスクの開始可能な時刻に反映させることによってナイーブに導入することが出来る。これは[5]で述べられているISH (Insertion Scheduling Heuristic)を、シングルプロセッサの演算パイプラインを一種の並列計算機と見做して適用したことに当たる。浮動小数点演算器のパイプラインステージのシーケンスは最

後のステージがボトルネックステージになっていることが多く、スケジューリングの対象となっている一つのタスクだけによる greedy なコストの評価は必ずしも適当ではない [2]。しかし、マルチプロセッサシステムにおける細粒度並列処理では、システム全体の演算器資源はフローグラフの並列度に比べれば充分あり、structural hazard はあまり深刻ではないと考えられるので、パイプライン制約はタスク開始時刻への反映だけに留めた。

DSH は通信コストをタスクの開始時刻に反映させているが、細粒度並列処理を当システムのような形で行なう場合、通信によるオーバーヘッドは単に後続タスクの開始時刻を遅らせるだけでなく、ロード及びストア命令の発行によってプロセッサ内の命令パイプラインにも影響を与える。そのため、スケジューリング時にロード及びストア命令もタスクとして考慮に入れなければならないし、また、冗長なロード及びストアを取り除き、通信によるコスト自体を低く抑える必要がある。先行タスクの値があるシュレッド内で必要である時、既にスケジューリング済みのタスクからの情報によって次のような場合が考えられる。

1. 先行タスクが CopyTaskList の中に複製されている
2. 先行タスクが同じシュレッド内に割り付けられている
3. CopyTaskList の中の複製されたタスクが先行タスクの値を同じシュレッド内に既にロードしている
4. 以前に同じシュレッドに配置されたタスクが先行タスクの値を参照している
5. 上記の 1～4 を満たさない、即ち先行タスクが異なるシュレッド内にしか存在しない

1 と 2 の場合はプロセッサ内のレジスタによって値を参照することができ、DSH のモデルにほぼ一致する。5 の場合は通信全体のコストはバスやスヌープキャッシュを介した通信の遅延だけではなく、先行タスクの値のストア命令と、値を読み出すロード命令を配置可能な時刻に依存する。従って、タスクグラフのエッジの通信コストは厳密にはフローグラフに対して固定ではない。3 と 4 の場合は [5] ではコストは同じだが、実際のプロセッサ上ではロード命令を減らすことができる。先行タスクの値のロードを先行タスクのシュレッドへの複製と見做せばこれは DSH のモデルに一致する。共有メモリシステムを用いていることにより各タスクが値を他のシュレッドに送るためのストア命令は一つのタスクにつき 1 回で良い。同期命令も命令パイプラインに影響を与えるが、後に述べる冗長な同期の除去によって不要になる可能性もあるので、タスクスケジューリング時には配置しない。

3.3 同期オーバーヘッドの軽減

同期命令の挿入時には [13, 7] に述べられているように、先行関係の推移的閉包を取ることによって冗長な先行関係を取り除くことができる。バリア同期は多対多の先行関係であるから、各バリア同期を具体的に決定してしまえば、副産物として gantt chart 内での依存の距離の短い先行関係が生じ、これによって距離の長い先行関係がさらに除去される可能性がある。elastic barrier 同期命令を具体的に決定する手続きは次のようになる。

1. 一対一のタスク間先行関係を多対多の先行関係にクラスタ化する

   このとき、可能な限り多くの先行関係を一つのクラスタにまとめるほど、バリア命令が命令パイプラインに与えるオーバーヘッドは少なくなる。勿論余分な先行関係を導入しているのでキャッシュミスやバスのコンテンションなどの外乱には弱くなる。クラスタ化のアルゴリズムは次のようなものを使用した (図 3)。

   (a) 先行関係を後継タスクの開始時間の早い順に並べる。

   (b) 列の始めの先行関係をクラスタに加える。

   (c) 先行タスクの開始時間が前回のバリア線より早い場合は、バリア線が交わってしまうので、クラスタには加えず単独で扱う。このような先行関係は先行タスクと後継タスクの間に他の先行関係を含んでいるので、ステップ 2 で除去される可能性がある。

   (d) 先行タスクの開始時間がクラスタ内の先行関係のもっとも早い後継タスクの開始時間より遅い場合、クラスタ化してもオーバーヘッドが増えるだけなので、新しくクラスタを作る。

   (e) 上記以外の場合は先行関係をクラスタに加える。

   (f) (a) ～ (e) を先行関係の列が空になるまで繰り返す。

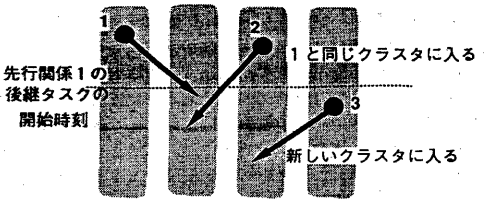


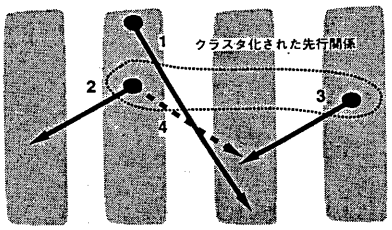
図 3: 先行関係のクラスタ化

2. 冗長な先行関係を除去する

   後継タスクのあるシュレッドから先行関係によって先行タスクの開始時間までに推移的にたどり着けるシュレッドに先行タスクが含まれていれば、その先行関係は除去できる (図 4)。

3. バリア命令を挿入しバリア線を張る

   gantt chart 上にスケジューリングされた先行タスクの下に elastic barrier の approve 命令、後継タスクの下に real request 命令を挿入する。各バリア線について、同期に参加しないシュレッドはダミーの approve 命令を発行する。また、後継タスクは real request の前に同期待ちを予告する pre request を発行する。ダミーの approve は前回のバリア線の直後まで上げることができる。pre request はシュレッド内のもっとも遅い先行タスクの下か、それがなければ前回のバリア線の直後まで上げることができる。



クラスタ化によって導入された先行関係4によって
1は取り除くことができる。

図 4: 冗長な先行関係の除去

3.4 各シュレッド内のレジスタ割り付け

OP.1 ではレジスタ割り付けはスケジューリング後に各シュレッド内で行なっている。これは要素プロセッサが充分多くのレジスタを備えているのでトレース内程度の並列性ではレジスタの溢れをそれほど気にしなくて良いことと、トレースの合流点でのレジスタのイメージを合わせる操作 (整数、浮動小数点の move 命令) がプロセッサ間通信に比べて非常に低コストであることから、後処理として行なってもスケジューリング結果にあまり影響を与えないと判断したためである。レジスタ割り付けはフローグラフの意味を保存すればよいので、生存期間解析を行なって一時変数の割り付けと同じように行なうことができる。

4 フロントエンド部の最適化手法

バックエンド部では DSH によってノードの複製によるローカルなフローグラフの変形を行なうが、入力となるフローグラフ自体も効率の良いものにし、同時に並列性をできるだけ抽出しておく必要がある。フローグラフの変形をコストを評価しながら行なう方法も検討されているが [6]、ここでは前処理として実現の容易なフロントエンド部での最適化方法を検討してみる。

フローグラフは中間コードと意味的に等価であるから、当然通常のシングルプロセッサに対する最適化手法は前処理として有効である。その他にも、並列性を拡大するための前処理としては次のようなものが挙げられる。

loop unrolling アンローリングはグローバルなフローグラフの変形を行なってトレース内の並列性を拡大する手法で、VLIW 計算機や RISC 計算機では良く利用されている。アンローリングによって生成されたコードには強い規則性があり、元のプログラムにない冗長さが見い出せる可能性がある。

disambiguation と copy propagation 配列のアクセスがあるループボディをアンローリングすると、do across 型のループでは確実に依存のあるメモリアクセスがトレース内に幾つも発生する。disambiguation によってこれらが依存することが確定できれば、copy-propagation によって制御依存のエッジをデータ依存のエッジに変換することができ、レイテンシを短くすることができる。逆に依存しないことが確定できればフローグラフの並列性を上げることができる。

common subexpression elimination 共通部分式の削除はフローグラフの並列性を低くする変換であるが、これは DSH における先行タスクの複製の逆変換である。共通部分式の削除を前処理として行なうことによって、プロセッサにある程度最適な並列度が抽出されることが期待できる。

5 性能評価

生成されたコードの評価をシミュレータ上で行なった。シミュレータは [11] で用いたものを使用した。細部のコストは、バリア命令の実行にプロセッサのパイプラインクロックで 2 clock、スヌープキャッシュの update プロトコルを用いたプロセッサ間通信に 6 clock とした。要素プロセッサのパイプライン構成は MIPS 社の R4000 に準拠している。サンプルプログラムの基本型のデータサイズは整数・浮動小数点数ともに 32bit とした。ISH と示されている線は、[5] 中で Insertion Scheduling Heuristic と呼ばれているタスクの複製を行なわないリストスケジューリング法で、今回のシステムに CP/MISF 法を適用したものにほぼ等しい。

5.1 例題 1

一つめの例は [13] で使用されているもので、級数計算を行なうプログラムである (図 5)。ループ内での演算レベルの並列性が高く、パイプラインコンフリクトを起こしやすい浮動小数点の除算が数多く含まれているため 8 台で 1 台の時に比べて約 2.5 倍のスピードアップが実現されている (図 6)。なお、グラフからはわかりにくいがプロセッサ数が 2 台の時は僅かであるが ISH が DSH を上回る性能を出している。このことは、プロセッサ数が少なく利用できる並列度の多さに対して資源制約が厳しい場合には、単純なヒューリスティックアルゴリズムで良い解を得ることが難しいことを物語っている。

5.2 例題 2

二つめの例は [9] で使用されている誤差拡散法のプログラムの一部である (図 7)。この例は演算レベルの並列性があまり高くなく、クリティカルパスに loop carried な依存がある難しい例である。[9] ではパイプラインステージがオーバーラップできない CISC 計算機を要素プロセッサに用いて評価しており、ループアンローリング無しで 1.67 倍のスピードアップを得ている。しかし、今回の条件では 1 台でコンパイルした場合にトレース内の並列性が殆んど要素プロセッサのパイプラインによる並列性に吸収されてしまい、1 台で既に dominant sequence 長が critical path 長に近くなっている。そのため台数が増えるにつれスケジューリング時に吸収できなかった通信のオーバーヘッドや同期のオーバーヘッドが現れ、1 台の時より性能が悪くなってしまう (図 8)。DSH はランダムに生成されたタスクグラフに対しては負の台数効果を示すことが殆んどない [5] 優れたアルゴリズムであるが、現実のコードから生成される並列度の低いタスクグラフにナイーブに適用した場合にはタスクの大きさやエッジに表された通信のコスト以外のオーバーヘッドが支配的になりうることを、図 8 は示している。

```
      integer v9, v10
      real v1, v2, v3, v4, v5, v6, v7, v8

      v1 = 0.0
      v6 = 1.0
      v7 = 1.0
      v8 = 4.0
      v3 = 0.0
      v5 = 0.0
      do 10 v9 = 1, 1000
         v10 = v9 - v9 / 100 * 100
         v2  = 1.0 / v6
         v2  = v2 - 1.0 / (v6 + 2.0)
         v2  = v2 + 1.0 / (v6 + 4.0)
         v2  = v2 - 1.0 / (v6 + 6.0)
         v3  = v3 + v7 / (v6 + 8.0)
         v3  = v3 - v7 / (v6 + 10.0)
         v3  = v3 + v7 / (v6 + 12.0)
         v3  = v3 - v7 / (v6 + 14.0)
         v4  = v7 / (v6 + 16.0)
         v5  = v5 + v7 / (v6 + 18.0)
         v4  = v4 + v7 / (v6 + 20.0)
         v5  = v5 + v7 / (v6 + 22.0)
         v4  = v4 + v7 / (v6 + 24.0)
         v5  = v5 + v7 / (v6 + 26.0)
         v4  = v4 + v7 / (v6 + 28.0)
         v5  = v5 + v7 / (v6 + 30.0)
         v1  = v1 + v8 / (v6 + 32.0)
         v1  = v1 - v8 / (v6 + 34.0)
         v1  = v1 + v8 / (v6 + 36.0)
         v1  = v1 - v8 / (v6 + 38.0)
         v2  = v2 + v4
         v1  = v1 + 4.0 * v2
         v5  = v6 + 40.0
10    continue
      v1 = v1 + 4.0 * v3 - 4.0 * v5
      end
```

図 5: 例題 1 のプログラム

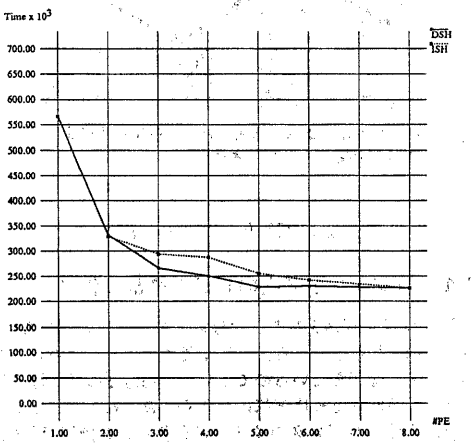


図 6: 例題 1 の実行結果

```
int     i, j;
float   a0, a1, a2, a3, a4, tmp,
        p, w, e[], d[], t[], g[];

for (i = 0; i <= N; i++) {
    j = (i & 0x3ff) + 1;
    w = (a0 * e[j-1] + a1 * e[j] + a2 * e[j+1]
        + a3 * tmp + a4 * d[i]) / 32.0;
    p = t[floor (w)];
    if (j != 1)
        e[j-1] = tmp;
    else
        e[1024] = tmp;
    tmp = w - p;
    g[i] = p;
}
```

図 7: 例題 2 のプログラム

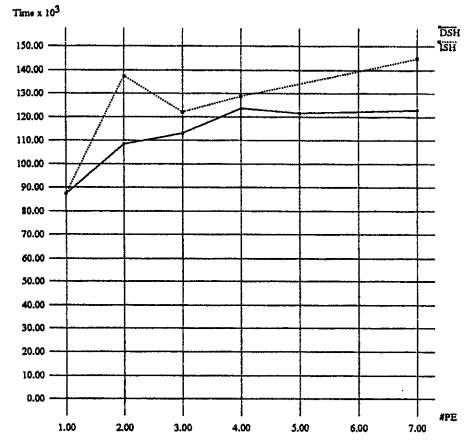


図 8: 例題 2 の実行結果

6 まとめ

基本ブロック内の命令レベルの細粒度並列処理を行なうために、通信のオーバーヘッドを考慮した静的タスクスケジューリングアルゴリズム DSH と同期オーバーヘッドの軽減を図るコード生成の手法を用いたコンパイラを作成し、シミュレータ上でその性能を評価した。本論文で示した手法ではまだあらゆるプログラムに対して正の台数効果を示すことはできないが、細粒度並列処理において、定量的にコストを考慮するヒューリスティックな最適化アルゴリズムと、定性的にオーバーヘッドを軽減する前処理や後処理を組合せた手法は様々な場面において有効であると思われる。

今後の課題としては、現在フロントエンド部の最適化手法はまだ実装されていないものが多いので、それらの実装並びにバックエンド部の最適化に与える影響についての考察が残されている。また、当研究室で製作中の汎用細粒度並列計算機お茶の水 1 号 [12] 上に実装し、性能を評価する予定である。

参考文献

[1] Ellis, J. R., *Bulldog: A Compiler for VLIW Architectures.* Cambridge: The MIT Press, 1986.

[2] Ertl, M. A. and A. Krall, "Instruction Scheduling for Complex Pipelines," in *Proc. of 4th Int. Conf. of Compiler Construction '92*, pp. 207–218, Oct. 1992.

[3] Gerasoulis, A. and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *Journal of Parallel and Distributed Computing, Special Issue on Scheduling and Load Balancing*, vol. 16, pp. 276–291, Dec. 1992.

[4] Kasahara, H. and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.

[5] Kruatrachue, B., *Static Task Scheduling and Grain Packing in Parallel Processing Systems.* PhD thesis, Electrical and Computer Eng. Dept., Oregon State Univ., Corvallis, 1987.

[6] Sarkar, V. and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," *SIGPLAN*, vol. 21, no. 7, pp. 17–26, July 1986.

[7] 笠原 博徳, 藤井 稔久, 本多 弘樹, 成田 誠之助, "スタティック・マルチプロセッサ・スケジューリング・アルゴリズムを用いた常微分方程式求解の並列処理," 情報処理学会論文誌, vol. 28, no. 10, pp. 1060–1069, Oct. 1987.

[8] 松本 尚, "細粒度並列実行支援マルチプロセッサの検討," 信学技法 *CPSY89-37*, pp. 37–42, Aug. 1989.

[9] 松本 尚, "細粒度並列実行支援マルチプロセッサの検討," 情報処理学会論文誌, vol. 31, no. 12, pp. 1840–1851, Dec. 1990.

[10] 松本 尚, "Elastic Barrier: 一般化されたバリア型同期機構," 情報処理学会論文誌, vol. 32, no. 7, pp. 886–896, July 1991.

[11] 松本 尚, "スヌープキャッシュ制御機構の DOACROSS ループへの適用," in 並列処理シンポジウム *JSPP'92* 論文集, pp. 297–304, June 1992.

[12] 中里 学, 大津 金光, 戸塚 米太郎, 松本 尚, 平木 敬, "お茶の水 1 号の構成と評価," 情報処理学会計算機アーキテクチャ研究会報告 *SWoPP'93*, Aug. 1993.

[13] 尾形 航, 吉田 明正, 合田 憲人, 岡本 雅巳, 笠原 博徳, "スタティックスケジューリングを用いたマルチプロセッサシステム上の無同期細粒度並列処理," in *Proc. Joint Symposium on Parallel Processing 1993*, pp. 111–118, May 1993.