中粒度並列化手法の評価

細井　聡　　小沢　年弘
APC 富士通

APC プロジェクトでは、マルチグレイン並列化コンパイラの研究開発を行っている。抽出する並列性としては、関数や基本ブロック間の粗粒度並列性と、中粒度レベルの並列性（ループレベルの並列性）とがある。中粒度レベルの並列性抽出の 1 方式として、スタンフォード大で提唱されている affine partitioning という手法をベースとして、いくつかの改良を加えた手法を開発している。今回、SPEC2000 の applu ベンチマークプログラムに本方式を適用したところ、DOALL 並列性とパイプライン並列性の双方を抽出することができるため、従来に比べて約 2.5 倍の性能が得られた。

The Evaluation of the medium grain level Parallelization Technique

Akira Hosoi　　Toshihiro Ozawa
APC Fujitsu

At the APC project, we are developing the multi grain parallelizer which extracts coarse grain level (between subroutines and/or basic blocks) parallelism and medium grain level (loop) parallelism. One of the medium grain parallelizers is the affine partitioning-based one which is proposed at the Stanford University, and we refined it. This method can extract both DOALL parallelism and pipeline parallelism. So in the case of ‘applu’(SPEC2000), it is about 2.5 times faster than that by the base compilers.

1. はじめに

Advanced Parallelizing Compiler(APC) プロジェクトでは、逐次 Fortran プログラムから様々なレベルの並列性を抽出しOpenMP 化したソースを出力する、マルチグレイン並列化コンパイラの研究開発を行っている。抽出する並列性としては、関数や基本ブロック間の粗粒度並列性、中粒度レベルの並列性（ループレベルの並列性）がある。中粒度レベルの並列性を抽出する方式の 1 つとして、スタンフォード大で提唱されている affine partitioning [2][3][4][5][6][7]という並列化手法をベースとした手法を用いている。

affine partitioning は、1)unimodular 変換や loop fusion/fission などの多くのループ変換を自動で行える 2)不完全入れ子ループから DOALL 並列性だけでなくパイプライン並列性を抽出することができる 3)並列性の抽出、データローカリティの改善、同期オーバヘッドの軽減を同時に行うことができるなどの特徴を持つ、最も強力な自動並列化手法の 1 つである。

しかしパイプライン並列性を抽出する場合、ループ本体が大きくなると、[2][3]で述べられている「アルゴリズム A」という手法では、質の良い解(充分に並列性が引き出され、かつ、データローカリティも充分に改善されているようなループ変換に相当する) を求めることは一般には難しい。質の悪い解が求まったり、たとえ質の良い解が求まっても、非常に効率が悪くコンパイルに多くの時間を要してしまうことが多いからである。そこで我々は、質の良い解を効率良く求める新アルゴリズムを考案した。

本稿では、まず 2 章で、affine partitioning ではどのようにしてパイプライン並列性を抽出するのか、また、より良いパイプライン並列性を抽出するために、実装する上でどのような工夫を行ったかについて述べる。そして 3 章で、OpenMP 上でどのようにしてパイプライン並列性を実現するかを説明し、4 章で SPEC2000 のうちの applu に対して我々の手法を適用した結果をいくつかのハードウェア上で評価した結果を述べる。5 章でまとめを行う。

2. affine partitioning によるパイプライン並列性の抽出とその改良

2.1. affine partitioning によるパイプライン並列性の抽出

affine paratitioning によりパイプライン並列性を抽出するには、まず、配列の添え字情報およびループ境界値(ループの初期値・終値)の情報から生成される制約条件、すなわち、連立不等式を構築する。そして、この連立不等式を満たす解のうち、できるだけ rank の大きい解を求めようとする。rank が 2 以上である解が求まれば、パイプライン並列実行可能となる[1]。解の rank が N であることは、変換後のループネストにおいて、最外の N 個のループが全ての代入文を囲むようなループネストに変換されることを意味する。そして、その最外の N 個のループは、任意の順でループ交換可能で、かつ、パイプライン並列実行可能となる。また、解の rank をできるだけ多くすることは、それだけ、tiling できる最外のループの数が増えることになり、ローカリティをより改善できる可能性が出てくる[2][3][5][6]。

しかし、[2][3]の「アルゴリズム A」という手法では、解を直接求めようとするが、これは、探索空間が広く効率的な方法ではない。我々の方式は、解の一部を最初に予想し、それをチェック・補完するという点が大きく異なる。以下に、我々のアルゴリズムの概略を述べる。

2.2. パイプライン並列性を抽出するための新アルゴリズム

1)解の一部を予想する

制約条件を直接解くことは効率が悪いので、解の一部を次のように予想する。affine partitioning は、ループネストの外側をできるだけ完全入れ子にして tiling 可能とすることにより、ローカリティを向上させようとする変換である。したがって、もし元のループネストの最外 N 個のループが全ての代入文を囲んでいるならば、変換後のループネストも最低 N 個のループが全ての代入文を囲む可能性が非常に大きい。よって、まず、変換後のループネストの形状をこのようであると予想する。

2)制約条件を分割して単純化する

これは、効率化のための工夫である。パイプライン並列性を抽出することは、affine partitioning では制約条件（連立不等式）を満たす解を求めることに相当する。その連立不等式は、ループ中の全ての代入文間の依存関係を結合したものである。しかし、ループ中の代入文の数が多くなると、不等式の次元数が非常に大きくなってしまい効率良く解くことが難しい。そこで、2 つの代入文ごとの制約条件（連立不等式）に分割する。それらの部分的な連立不等式は非常に簡単に解くことができることが多い。これらの解を求め、それらを合成することにより、全体の解を予想する。

3)ループ境界値を一部無視する

これは、できるだけ解の質を落とさずに解を効率良く求めるために行う。一般に、制約条件は、配列の添え字情報とループ境界値の情報とから構成される。データ依存解析がそうであるように、ループの境界値の情報を無視すれば、制約条件が単純となり、解くのが容易となる。しかし、全てのループ境界値を無視してしまうと、完全入れ子となるループの数が少なくなるなど、質の良い解が求まらないことが多い。そこで、パイプライン並列性を抽出するための制約条件を**fusion 制約条件**と**完全入れ子制約条件**の 2 つに分割して求めることにした。

fusion 制約条件と完全入れ子制約条件

一般に、不完全入れ子ループ内の任意の 2 つの配列参照においては、少なくとも 1 つのループが両方の配列参照を囲んでおり、どちらか一方だけの配列参照を囲むループが 0 個以上存在する[2]。

[1] rank が 1 である解は必ず存在する。それは、オリジナルの逐次プログラムである。

[2] 0 個の場合は、完全入れ子ループ

```
do  i = i0, i1
  do  j = LBj(i), UBj(i)
    A(f(i, j)) = …
  enddo
  do  k = LBk(i), UBk(i)
        … = A(g(i, k))
  enddo
enddo
```

図 1　不完全入れ子ループネスト

たとえば、図 1 では、i ループは 2 つの配列参照を囲んでいるが、j、k ループは一方の配列参照しか囲んでいない。図 1 の場合、fusion 制約条件と完全入れ子制約条件を以下のように定義する。

fusion 制約条件

i ループの内側（すなわち、i = 一定）を考える。図 1 における配列の参照順序を考えると、全ての A(f(i, j))へのストアを行った後に、A(g(i, k))の各要素をロードする。ところが実際は、ストアとロードを両方行う要素（A(f(i, j)) == A(g(i, k))を満たす要素）のみこの順序に従えば良く、それ以外の要素は全く任意の順に実行してかまわない。以上を fusion 制約条件と定義する。この制約条件を満たすようにすることは、j ループと k ループをループ融合することに相当する。

完全入れ子制約条件

図 1 において、 i が変化したときの 2 つの配列参照の依存関係から生成される制約条件[3]を完全入れ子制約条件と定義する。

そして、完全入れ子制約条件に対しては、ループの境界値を無視するようにした。その理由は、①ループ融合に相当する fusion 制約条件において、ループ境界値を無視することは、質の良い解が求まらない可能性が大きい ②完全入れ子制約条件では、両方の配列参照を必ず囲んでいる。すなわち、ループの境界値が常に等しいので、fusion 制約条件の場合よりはループ境界値を無視することの影響が少ないからである。

4) 3)までに予想した解の一部が正しいことをチェックし、そして、解を補完することにより、解全体を求める。

[3] 2 つの配列参照の順序関係が、i が増加するにつれてどうなるかを求める。

2.3. 新アルゴリズムの適用例

新アルゴリズムを SPEC2000/applu のサブルーチン buts()内のループに適用する場合を考える。このループは、最外の kji ループが全ての代入文を囲む不完全入れ子ループネストである（図 2(a)）。

```
do k = nz-1, 2, -1          do k = 1, nz-2
  do j = ny-1, 2, -1          do j = 1, ny-2
    do i = nx-1, 2, -1          do i = 1, nx-2
      do m =1, 5                  do m =1, 5
        ...                         ...
          do l=1, 5                   do l=1, 5
            ...                         ...
          enddo                       enddo
      enddo                       enddo
      do m =1, 5                  do m =1, 5
        tmp=...                     tmp=...
        do l=1, 5                   do l=1, 5
          ...                         ...
        enddo                       enddo
      enddo                       enddo
        ...                         ...
    enddo                       enddo
  enddo                       enddo
enddo                       enddo
      (a)                         (b)
```

図 2　buts 内のループ

図 2(a)は、loop normalization（図 2(b)）や scalar expansion、array privatization などの前処理を行った後、**2.2.**を試みる。

①元のループの最外の 3 つのループが全ての代入文を囲んでいるので、変換後のループも同様であると予想する。

②kji ループは全ての配列参照を囲んでいるので、これらのループ境界値を全て無視する。

以上のようにして解を求めると、結局、kji ループが全ての代入文を囲み、kji ループが任意の順でループ交換可能で、かつ、パイプライン並列実行可能となるという結果が得られる。

3.　パイプライン並列性の実現

ここでは、buts 内のループから抽出したパイプライン並列性をどのように実現するかについて述べる。

まず、変換後のループの kj ループはループ交換可能となるので、kj ループを入れ換えたループネストで考える（図 3(a)）。図 3(a)を逐次実行した場合の実行順序を考えると、

図 3(b)の iteration space 中、太矢印で示したような順序となる。

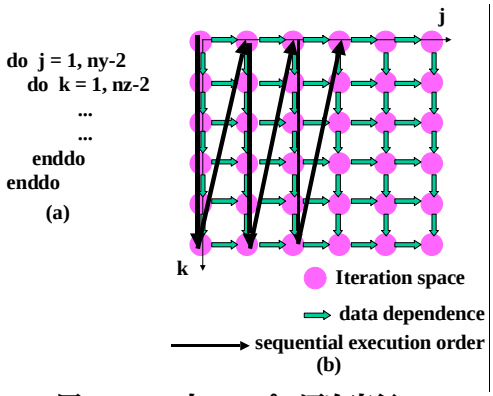

図 3　buts 内ループの逐次実行

次にこのループを並列実行することを考える。図 4(b)では、k ループを分割して複数のプロセッサに割り当てている。今、プロセッサ P1 の実行を考えると次のようになる。
1)P1 は P0 が j=1 の処理が終了するまで待つ。
2) P0 は j=1 の処理を終了すると、それを P1 に伝え j=2 の処理を開始する。そして、P1 が j=1 の処理を開始する。
3)P1 が j=1 の処理を終了すると、それを P2 に伝え、自分は j=2 の処理を開始する。

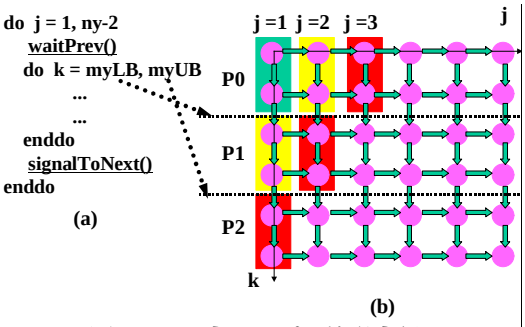

図 4 buts 内ループの並列実行

この場合の各プロセッサ P_n が実行する SPMD コードの概要を図 4(a)に示す。ここで、waitPrev()は P_{n-1} の処理が終了するまで待つ処理を行い、signalToNext()は自分の処理が終了したことを P_{n+1} に伝える処理を行う。我々のコンパイラは OpenMP ソースを出力するので、これら同期のためのサブルーチンも OpenMP で実現している。具体的には、[8]の FLUSH ディレクティブを用いた point-to-point 同期の実現をベースに、プロセッサ間の同期が正しく行えるように実装した。

4.　評価

本方式を SPEC2000 ベンチマークプログラムの applu に対して適用し、Sun SMP と Compaq Alpha Server 上で評価を行った。

4.1. SPEC2000/applu について

図 5 に示したループが applu の全実行時間のほとんど 100%を占める。サブルーチン jacld()、jacu()、 rhs()は容易に DOALL 化できるが、さらに、buts()と blts()から並列性を抽出しないと、全体として充分な性能向上が得られない。

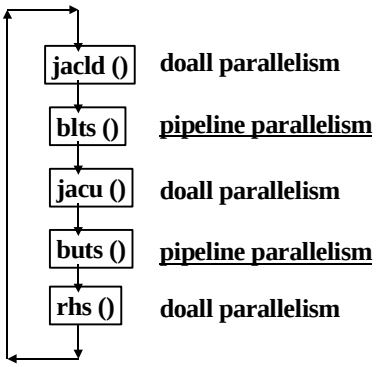

図 5　applu のメインループ

4.2.　Sun SMP 上での性能

図 6 に測定に使用した Sun SMP の構成と Forte Fortran コンパイラのコンパイルオプションを示す。コンパイルオプションは、SPEC2000 に対して Sun 社が Web 上で開示しているオプションを元に、当 SMP 用にチューニングした。

- Sun SMP
Ultra SPARCII(300MHz) ×12
Data Cache L1 16KB(direct-map)
L2 4MB(direct-map)
- Sun Forte 6.1 Fortran Native Compiler's options
sequential:
-fast -xcrossfile=1 -pad -dalign -Bstatic -xarch=v8plus -xcache=16/32/1:4096/64/1 -dn
parallel:
-fast -xcrossfile=1 -pad -dalign -xarch=v8plus -xcache=16/32/1:4096/64/1 -parallel -reduction -stackvar

図 6　Sun SMP の構成と Sun Forte のコンパイルオプション

CPU 数を変化させた場合の実行時間の変化を図 7 に示す。ここで、Sun(seq)は Sun Forte コンパイラにより生成した逐次用オブジェクトの実行時間を表わし、Sun(para)は CPU 数を変化させた場合の Sun Forte により生成した並列実行用オブジェクトの実行時間を示している。また、APC は、我々の中粒度並列性抽出手法による性能である。我々のコンパイラでは OpenMP ソースを出力するので、その OpenMP ソースを Sun Forte の OpenMP コンパイラによりコンパイルした。OpenMP コンパイラに与えたコンパイルオプションは、並列用のオプションに OpenMP ソースを認識できるように -mp=openmp を追加しただけである[4]。

我々の手法は、Sun(para)に対し 8 CPU で 2.7 倍の性能が得られている。その理由は、Sun(para)では DOALL 並列性は抽出しているが、パイプライン並列性を抽出していないからである。

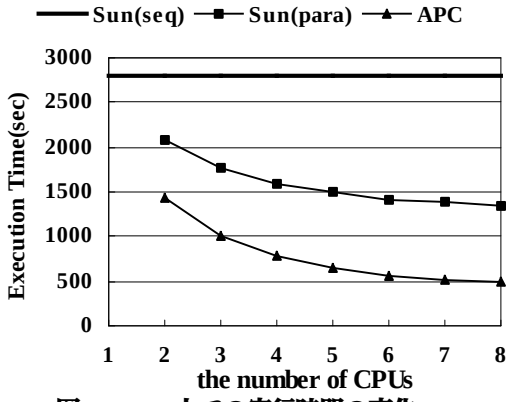

図 7　Sun 上での実行時間の変化

4.3. Alpha Server 上での性能

図 8 に測定に使用した Alpha Server の構成と Digital Fortran コンパイラのコンパイルオプションを示す。コンパイルオプションは、Compaq 社が Web 上で開示しているコンパイルオプションを元に、当マシン用にチューニングした。ただし、applu の場合、base と peak とでコンパイルオプションが異なること、また、並列実行用オブジェクトの性能が両オプションでかなり異なる傾向を示すことから、それぞれのオプションの場合について測定を行った。

- Compaq Alpha Server GS160 Model 6/73
Alpha 21264 (731MHz) × 8
Data Cache L1 64KB(2-way)
L2 4MB(direct-map)
- Compaq Digital Fortran Compiler's options

(1) base
sequential: -v -arch ev6 -O5 -fkapargs'-ur=1'
parallel: -v -arch ev6 -O5 -fkapargs' -conc -ur=1'

(2) peak
sequential: -v -g3 -arch ev6 -fast -O5 -transform_loops -unroll 14 -fkapargs='ur=1'
parallel: -v -g3 -arch ev6 -fast -O5 -transform_loops -unroll 14 -fkapargs='-conc ur=1'

図 8　Alpha Server の構成と Digital Fortran のコンパイルオプション

4.3.1 base オプションの場合の性能

base オプションの場合の実行時間の変化を図 9 に示す。Sun SMP の場合と同様、Compaq(seq)は Digital Fortran により生成された逐次用のオブジェクトの実行時間を、Compaq(para)は並列実行用のオブジェクトの実行時間を表わしている。

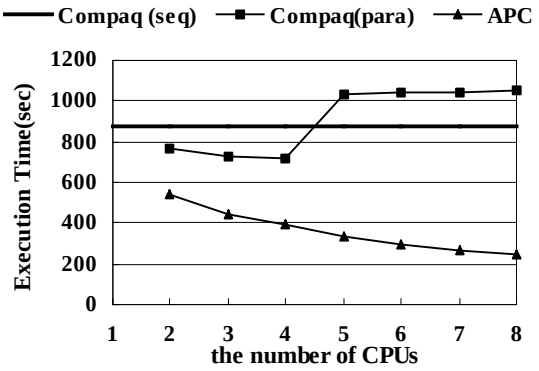

図 9　Alpha 上での実行時間の変化 (base)

Compaq(para)は 5CPU を超えるとかえって遅くなり、逐次版より性能が悪くなっている。これは、Alpha Server が 4CPU を単位とする cc-NUMA 構成であることが一因であると考えられるが、詳細は調査中である。これに対し我々の手法では、CPU 数の増加につれて実行時間が短くなっている。

4.3.2. peak オプションの場合の性能

[4] Sun Forte の最適化に我々の並列化が追加されたことを意味する。

peak オプションの場合の実行時間の変化を図 10 に示す。peak オプションでは base に対して、さらに unrolling などの最適化が施されている。

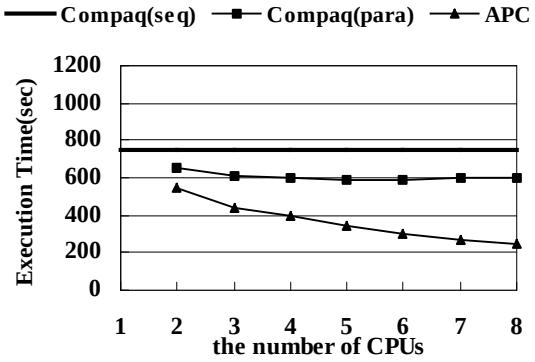

図 10 Alpha 上での実行時間の変化 (peak)

図 9 と比較すると、まず、逐次版の性能が base に対して 2 割弱向上していることがわかる。また、Compaq(para)は、5CPU 以上で逐次より遅くなることはないが、そこで性能が飽和してしまっている。一方、我々の手法では、base の場合とほとんど同じ性能が得られている。そして peak の場合、Compaq(para)に対し、8CPU で 2.4 倍の性能が得られている。

5. まとめ

我々の中粒度並列性抽出方式では、不完全入れ子ループから DOALL 並列性とパイプライン並列性の双方を抽出することができる。SPEC2000/applu の場合、従来と比較して 2.5 倍程度の性能向上が得られた。今後、他のベンチマークに対しても評価を行っていく予定である。

【参考文献】

[1] A.W.Lim and M.S.Lam: Communication-Free Parallelization via Affine Transformations, Lecture Notes in Computer Science, Vol. 892, 1995.

[2] Amy W. Lim and Monica S. Lam: Maximising Parallelism and Minimizing Synchronization with Affine Transforms, Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Jan., 1997.

[3] Amy W. Lim and Monica S. Lam: Maximizing parallelism and minimizing synchronization with affine partitions, Parallel Computing, Vol. 24, No. 3-4, pp. 445-475, May, 1998.

[4] Amy W. Lim, Gerald I. Cheong and Monica S. Lam: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication, ICS'99, Jun., 1999.

[5] A. W. Lim and M. S. Lam: Cache Optimizations with Affine Partitioning, proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, Mar. 2001.

[6] A. W. Lim, S-W. Liao and M. S. Lam: Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning, proceedings of PPoPP , Jun. 2001.

[7] A. W. Lim: Improving Parallelism and data Locality with Affine Partitioning, PhD Thesis, Stanford University, Aug. 2001.

[8] OpenMP Fortran Application Program Interface Version 1.1 Nov. 1999.