ハイパフォーマンス
コンピューティング 77－6
（1999．8．2）

ループの部分実行に基づく並列化コンパイラの実装

横田 大輔† 千葉 滋†‡ 板野 肯三†

†筑波大学工学研究科 ‡科学技術振興事業団 さきがけ研究21

自動並列化コンパイラは、並列化のためにプログラムの静的な解析をおこなうが、プログラムの挙動によっては解析が非常に難しくなりがちである。本論文では、静的な解析をおこなうのではなく、プログラムの一部を実際に実行し、実行プロファイルを収集、解析をおこなうことでプログラムの並列化をおこなう方法について述べる。この方法では、いちど部分実行を終えた後は、プロファイルを元に静的に並列化をおこなうので、すべてを動的におこなう方法よりも実行時のオーバヘッドが少なくなる。本論文ではまた、この部分実行による並列化のアイデアに基づいて、我々が現在実装しているコンパイラについて構成を述べる。

The implimentation of compiler based on partial execution of loops

DAISUKE YOKOTA †, SHIGERU CHIBA †‡ and KOUZOU ITANO †

†Institute of Information Science and Electronics, University of Tsukuba
‡PREST,Japan Science Technology Corp.

This paper describes a new paralellizing method, partial execution we named. Parallelizing compilers analize source programs by static analysis. But flows in programs sometimes makes this analysis very difficult. This paper describes our method which inspects information for paralellization not by static analysis but by executiong codes partially. This method demands less overhead in execution time than methods which process all in execution, because our method paralellizes a program statically with a profile after once partial execution. This paper describes the compiler which we are implementing based on our idea, partial execution.

1. はじめに

プログラマによる簡単なヒントをもとにループを自動並列化するコンパイラが数多く開発されているが、そのようなコンパイラは非常に緻密なプログラムの静的解析を必要とする。このためプログラムの挙動によっては、解析が困難で効率良く並列化できなかったり、また新しい並列化の技術を開発しても、コンパイラの開発が壁になって実用化に非常な時間がかかることがある。とくに巨大な配列をプロセッサに分割にして配置し、並列計算をおこなう場合、ループの反復ごとに、配列のどの部分を他のプロセッサに送信しなければならないか解析しなければならないが、このような解析はときに非常に困難である。

本論文では、プログラムの一部を実際に実行して、配列参照の挙動に関する情報を収集し、それに基づいてループの並列化をおこなう方法について述べる。この方法の利点は、複雑な挙動を解析できるコンパイラでも、その実装がやさしくなることである。また、実行時に実際の挙動を観察しながら、動的にプロセッサ間の通信をおこなう方法とくらべても、実行時オーバーヘッドの点で有利である。我々の方法では、配列参照の挙動の解析をコンパイル時におこなうので、実行時のオーバヘッドは非常に小さくなるからである。

以下では、まず静的な解析が困難なケースについて述べ、次に我々の方法について述べる。さらに現在、我々が実装しているプロトタイプ・コンパイラについて述べ、最後に関連研究とまとめ、そして今後の課題について述べる。

2. 静的な解析が困難なケース

ループを並列化する際、静的な解析が困難になる原因は、大きく二つある。制御構造による問題と、配列へのアクセスに関する問題である。これらの問題は、フローグラフやそれに類する表現を用いて、繁雑な解析を行なえば、ある程度解決可能である。

しかし、これらを利用しても通信の挙動の解析が困難なケースがある。例えば、配列の添字が他の配列の内容、関数の戻り値、非線形な計算であったり、配列の添字が線形でも正規化が難しい場合には、解析が困難になる。

このような問題を持ったループを並列化する場合、大きく二通りの方法がある。

一つ目は、動的にデータを所有しているプロセッサを探し、そのデータへのアクセスがローカルアクセスになるのか、通信になるのか特定し、通信が必要な場合は、それを所有しているプロセッサがデータを送信し、必要としているプロセッサがデータを受信する方法である。

二つ目は、目的別の並列化の手法を用いる方法である。この方法は、コードの目的に合わせた並列コードのテンプレート等を用意しておき、目的に合わせてテンプレートを選択して並列化する方法である。この方法は、一般的な並列化が困難なケースでも、多くのプログラマが頻繁に記述するコードを並列化できる。例えば、二重ループで、内側の境界が、外側のループの制御変数で要素が指定される配列変数の値で作られるループを並列化する手法[7]、リダクションループ等を、リダクション用のコードに置き換える手法などがある。

一つ目の方法の問題点は、静的な解析を用いて生成されたコードにはないオーバーヘッドがあり、これが大きい点である。このオーバーヘッドはデータを所有しているプロセッサを探すために発生する。このオーバーヘッドは、実際に行なわれる通信が単純な規則性にしたがっていた場合でも発生する。この場合、データを所有するプロセッサを探すためのコードは、成果に対して割に合わないコストになる。

二つ目の方法の問題点は、目的毎に並列コードを準備しなくてはならない点である。並列コードが用意されていないコードは、他の方法が必要になる。

3. 部分実行を用いた並列化

ソースコードが複雑であっても、結果として簡単で規則的な通信パターンを持つ場合がある。この場合、静的な解析が困難になり、出力に通信先を探すコードを組み込み、動的に通信先を探す必要が生じる。しかし、部分実行を用いた並列化は、このような方法と違い、生成されるコードのすべての通信が通信先を特定されている。このため、実行時に通信を特定するためのオーバーヘッドはない。

部分実行を用いた並列化は図1の流れで行なわれる。

ソースプログラムをログを取るためのコードに変換し、それを実行する。次にその実行結果の情報を用いて、実際の並列化を行なう。

本論文の手法では、ログを取るフェーズと利用する(並列化する)フェーズに完全に分離して並列化を行なう。これは、今後、部分実行を用いた並列化のアプローチの改良を容易にする。

3.1 部分実行

本論文の並列化コンパイラは、並列化をする前に、ソースコードの一部分を実際に実行する。本論文ではこれを部分実行と呼ぶ。部分実行は並列化に有用な情報をログとして記録する。部分実行のためのプログラムは、ソースプログラムのループの境界、配列の範囲を変更して、

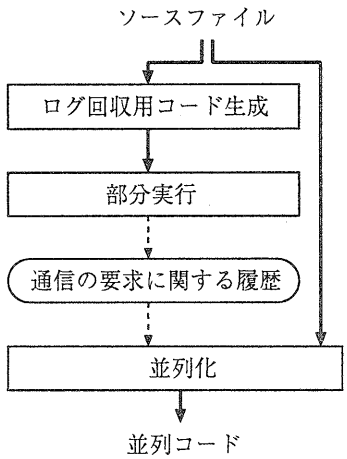


図1 部分実行を用いた並列化の流れ

分散処理される配列へのアクセスを監視するコードを埋め込んだものである。

部分実行は、実行する量と得られる情報のバランスが重要になる。配列、ループの反復に関する全ての情報を得るのは、ソースプログラムの全てを実行しなくてはいけない。このため、ソースプログラムのなかで、実行のコストに比べ、得られる情報が多い部分を選択する必要が生じる。

本論文では、部分実行する範囲を、並列に実行されるコードの1プロセッサ分に限定した。一般に、SPMDの並列コードは各プロセッサの振舞が互いに酷使しているので、1プロセッサ分の実行はコード全体の実行の代表とみなせる。したがって、本論文における部分実行する範囲は妥当である。

取られたログは、並列化の際、コンパイラによってアクセスされる。

この手法による並列化は、動的に通信先を決定する簡便さを、動的に通信先を決定することによるオーバーヘッドなしに実行できる。なぜなら、通信先を決定するための動的な試験(部分実行)をコンパイル時に行なうからである。これを実現するために、ログを記録するフェーズ(部分実行)およびコードは、並列化するフェーズおよび得られたコードとは完全に別個のものにした。

3.2 プログラマが持つ責任

本論文の手法を用いた並列化は、大きく分けて二つの責任をプログラマに要求する。部分実行が得る情報に関する責任と、並列化に発生する責任である。

3.2.1 部分実行が得る情報に関する責任

本論文の並列化コンパイラは、ある1台のプロセッサの通信のパターンを、全てのプロセッサの通信パターンとして利用している。これにより、コンパイラはループの反復および配列の全ての要素を検証する必要がなくなる。

このため、部分実行を用いた並列化では、プログラマ

が保証しなければならない点が生じる。

ソースプログラムが以下の三点を満たしている事を、プログラマが保証しなければならない。

・どのプロセッサも同じ通信パターンを持つ。

・ループの境界が定数。

・プロセッサ、配列の境界がトーラス状につながっている。

また、部分実行によってとられた1プロセッサ分のログが、全体を代表できると保証できる場合でも、ログの利用の方法によっては、さらに責任が発生する場合がある。

3.2.2 並列化に発生する責任

部分実行を用いて並列化した際、並列化のアプローチによっては、上記の三点以外に、プログラマが保証しなくてはいけない点がある場合がある。部分実行によって取られた通信の履歴、ループの反復毎の変数の依存距離等を利用する際、これらの情報のうち、全ての反復の分の情報を用いない場合、情報の一部を切り捨てる事になる。

例えば、本論文で実装に用いているアプローチは、全ての通信履歴を参照しないで並列化するアプローチである。このアプローチは次のような仮定をもとに並列化を行なう。配列の境界付近では同一の方向、距離の通信を持ち、配列の内部では通信を持たない、これらの区分けはループ制御変数と定数の不等式で表せられる。

ログの全ての情報を用いないで並列化した場合は、部分実行によって得られた情報の一部を丸めた事になるので、特別な保証をプログラマに要求する必要が生じる。このような保証は並列化のアプローチに依存する。

4. 実　　装

本論文では、この手法が有効である事を確認するために、現在、実装を行ない予備実験をしている。コンパイラは FreeBSD 上で動作し、入力は Fortran、出力されるコードは FreeBSD で構成された LAN 上の複数のマシンで動作する C のコードである。出力されるコード(図2)は各マシンにコードの実行をさせるランチャーと、SPMD 形式で並列に実行される本体のコードである。

本論文の並列化コンパイラは Fortran を入力とし、C を出力にした。Fortran は並列化コンパイラを利用する非情報系のからの要望が大きく、現在でも、C や C++ よりも利用されている。このため、入力を Fortran にした。また、実行される環境が、FreeBSD であるので、利用の簡便さから C を出力にした。

4.1 処理の流れ

本論文で論じられた並列化コンパイラは、図3に示される流れで、並列化を行なう。Fortran から C に変換するフェーズでシンボルテーブルを作る。このシンボルテーブルは並列化のフェーズで再び利用される。この時、出力される C のソースには、後のフェーズのため、部分実行、並列化に関係した部分が分かるように、目印となるコードが埋め込まれている。

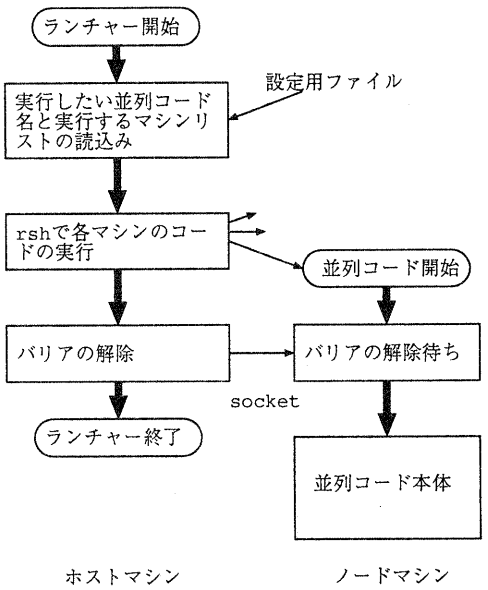


図2　出力されるコードの動作

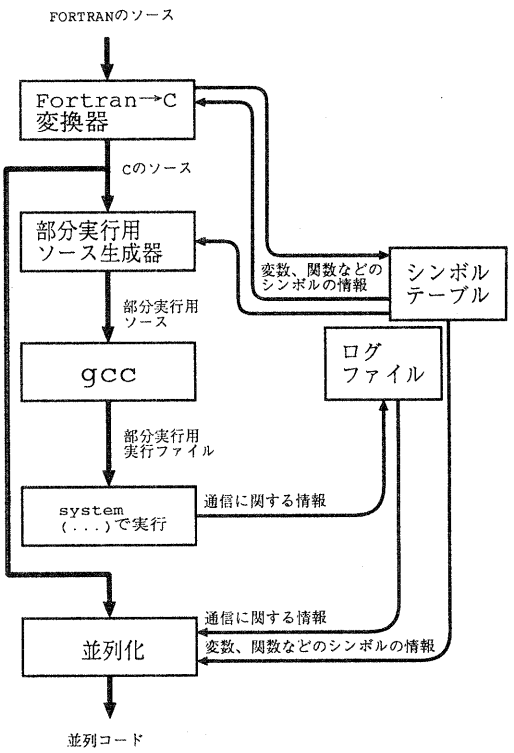


図3　部分実行を用いた並列化の流れ

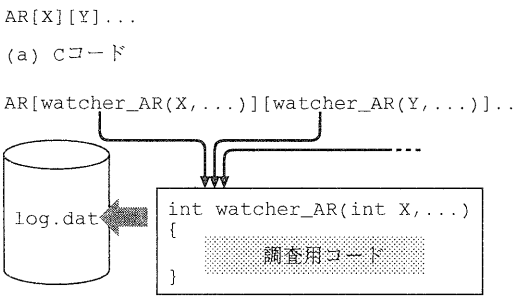


図4　配列変数へのアクセスの変更

最初に、コンパイラは、通信の情報を取るために、ソースを部分実行する。変換されたCのソースは部分実行用のコードに変換されて、gccに渡され、実行される。このコードは、ログを取るためのコードが埋め込まれている。

次に、コンパイラは、ログを用いてCのソースを並列化する。このフェーズはシンボルテーブル及びログファイルの情報を用いて並列化を行なう。コンパイラは、配列変数がどのように分散処理されるか(ブロック数など)情報を、シンボルテーブルから読みだし、その変数がどのように通信を要求するか(どの反復で通信を必要としたか、どのプロセッサと通信をひつようとしたか、配列変数のどの要素を必要としたか等)情報を、ログから読み出す。この二種類の情報を用いる事によって、本論文のコンパイラは、並列化のための静的な解析を省く。

4.2 部分実行

部分実行は、ソースコードの一部分を実行する。本論文では、部分実行する範囲は、並列化を指定されているループの1プロセッサ分である。このため、部分実行用のコードは、ソースコードのループを中心に加工したものである。

4.2.1 部分実行コード

部分実行用のコードを出力するには、Cに変換後、コードの以下の点を変更する必要がある。

- ループの境界、配列の範囲
- 分散処理される配列変数へのアクセス
- 現在の反復の位置の書き出し用コードの追加

まず、ループの境界を、$for(I=1;I<N;I++)$ から $for(I=1+N*(PID-1)/PNUM;I<1+N*PID/PNUM;I++)$ に変換する。ただし、反復数$=N$、プロセッサ数$=PNUM$、プロセッサID$=PID$、制御変数$=I$とする。なお、ループはFortranからCに変換する時点で、正規化されている。

次に、分散処理される配列変数へのアクセスを記録するために、これを調査するコードを埋め込むようにした。これは、分散処理される配列変数をアクセスする箇所を図4で示されるように変換する事で行なった。なお、(a)が変換前で(b)が変換後である。

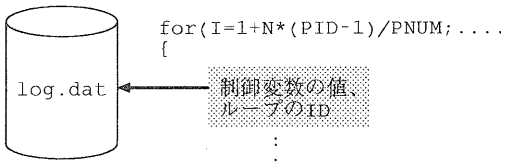


図5　ログへの反復の記録

アクセスを記録する関数は、配列変数へのアクセス毎でなく、次元毎に挿入するようにした。この位置に挿入することによって、調査する配列変数が、代入、参照に関わらず、同様に監視できるので、容易に実装できる。アクセスを記録する関数に渡されたデータは、その関数の中で、変数へのアクセス毎に変換され、二次記憶装置に記録するようにした。なお、関数に渡す情報は、制御変数の値、プロセッサローカルな制御変数の値、変数名、ソース上の変数に一意に付けられたID等である。

最後に、現在の反復の位置の書き出し用コードの追加するようにした。ブロックで分散する指定があるループのボディの先頭に、ソースコード中のループを特定できるIDと制御変数の現在の値を出力するコードを埋め込むようにした(図5)。並列化する際、この記録をもとに、通信のあった反復を特定する。

4.2.2 部分実行の出力

並列化に必要な情報には以下のものがある。通信があった反復の制御変数の値、その通信の通信先、通信によって参照(または代入)される要素の位置等の、通信に関わるもの(これらの情報を通信情報と呼ぶ)と、ソースコード中のどの部分で通信を必要としたか、ソースコード中の位置を特定するための情報(これらの情報を位置情報と呼ぶ)である。

位置情報は、通信情報に付随した形で存在する。本論文では、以下の形式で、部分実行を記録する事にした。

#_通信情報の種類:位置情報:通信情報の内容

部分実行の出力先には大きく二種類が考えられる。一つ目はメモリであり、二つ目は二次記憶装置などのファイルである。部分実行の結果をメモリ上に置く事は、柔軟で高速な操作を可能にする。しかし、部分実行の結果のサイズは、要求のあった通信量に比例するので、現実的でないメモリ量を必要とする可能性がある。部分実行の結果をファイル上に置く事は、柔軟で高速な操作には向かないが、より多くのデータを記録する事ができる。

本実装では、部分実行の結果をファイル上に置く事にした。現実的なシステムではどちらが妥当であるか、慎重な吟味が必要であると考えられる。本論文では、実装の上で制限は厳しいが、現実的な環境で確実に実行できる理由から、後者を利用した。

4.3 並列化

本実装はプロトタイプである。部分実行による並列化を用いたコンパイル時間が妥当である事を示すために、

コンパイラを実装している。並列化のフェーズは極めて単純な手法を採用した。本論文の手法は、部分実行と並列化は独立したフェーズであり、コンパイルタイムが問題になるのは部分実行のフェーズであるため、このプロトタイプを用い、現実的な利用が可能かどうか裏付ける事は妥当である。

並列化は、以下の仮定に基づいて行なわれる。通信がループの境界付近に発生する。ループで分散処理される各々の配列変数の依存距離は、全ての反復において等距離で定数である。

並列化はログの一部を走査する事で、通信の必要な反復を特定する (図 6)。まず、N 次元のループの、中央を通るように、各次元の軸方向にログを走査して、その次元の通信の状況が変わる境界を求める。次に、ループ中、その配列にアクセスする部分を、2^N に分割する。そして、それぞれの領域に、適した通信命令ないしは配列変数へのアクセスを記述する。

図 6は二次元配列の例である。分散処理される配列変数はループに対して、(-2,-1) の距離で参照される (図 6(a))。ループのどの領域で、配列へのアクセスが通信を必要とするか、部分実行によって生成されたログファイルを走査して調べる (図 6(b))。この例では、二次元配列を二重ループでアクセスする例なので、各方向について、計二回の走査が必要になる。反復を通信の種類で分類し (図 6(c))、それに合わせてソースコードにソケットによる通信を入れる。

5. 関連研究

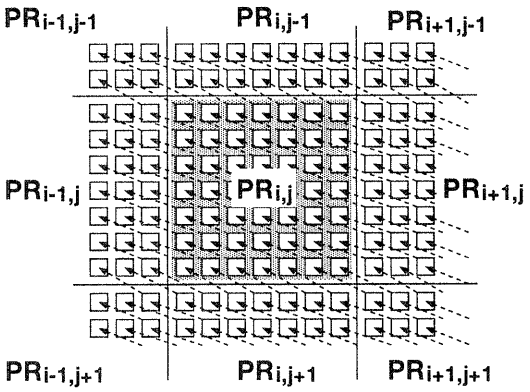
並列化の際、動的に通信とローカルなアクセスを判定する手法に、Inspector と Executor[2] の組み合わせがある。これは、配列の添字が他の配列の内容を用いるようなケース、仲立ちをする配列を介した、配列の要素にアクセスするループに用いられる。次はその例である。

```
DO[BLOCK=4] I=1,12
        Y[I]=Y[I]+AP[IP[I]]*X[I]
END DO
```

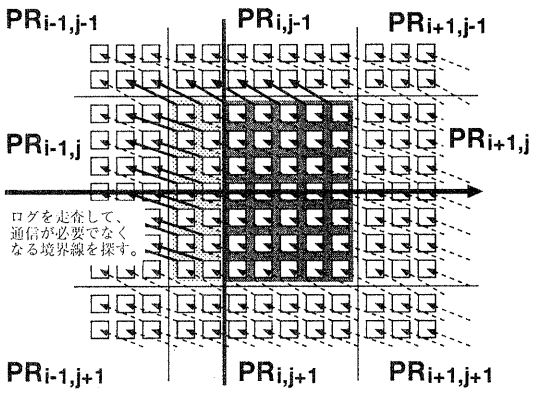
Inspector と Executor 本体は、コンパイラによってコードに組み込まれる。Inspector は、配列へのアクセスが、ローカルなデータか、通信が必要か、それがどのプロセッサが持つものか調べる。Executor は、Inspector で得た情報を元に実際の配列の要素のデータを送受信し、ループのボディの処理を行う。

この方法の長所は、仲立ちをする配列を持つ配列へのアクセスを、明示的な送信および受信で行える点である。したがって、自動送信の機構 (データを要求されたプロセッサが、自動的にローカルなデータを返信する機構) を準備する必要がない。自動送信の機構は、オーバーヘッドになる。

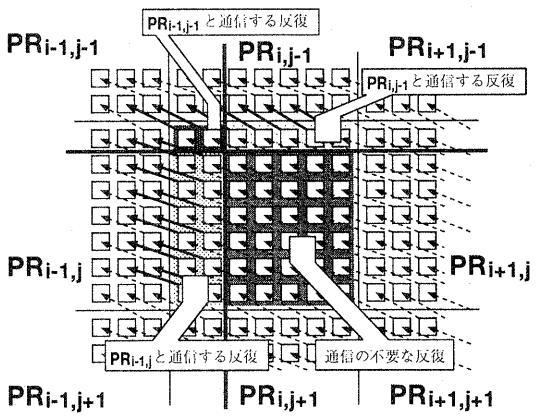
この方法の短所は、通信のための下準備として Inspector がオーバーヘッドになる点である。



(a) 分散処理される配列と依存



(b) 通信を必要とする領域の決定



(c) 配列の領域と通信先

図 6 並列化

Inspector と Executor を用いた方法は、通信が必要な個所の特定と通信先の特定を動的に行う。この点が本論文との基本的な立場の違いである。

6. まとめ

部分実行による並列化は妥当な時間内に処理可能である。部分実行による並列化では、一般的に膨大なコンパイル時間が必要であることが予想されたが、ソースコードの妥当な一部分を選択して実行することで、有効な時間内に並列化できる。

また、部分実行による並列化は静的な解析と動的な解析の両方の利点を持っている。静的な解析を用いた並列化に比べ、並列化が容易であり、動的な解析等のように、通信が必要な個所を特定するコードと通信先を特定するコードを、生成されるコードに残さない。

部分実行によって得られた情報は、通信の実態であり、極めて直感的で分かりやすいものである。したがって、並列化のアルゴリズムも立て易く、より高度にハードウェアを利用した並列コードを生成できるような、並列化コンパイラを実装することが容易に可能である。

7. 今後の課題

部分実行を用いた並列化において、今後の課題となる点は大きく二つある。一つ目は、静的な解析では困難な、より高度な並列化を可能にする点と、二つ目は、プログラマがソースコードに対して持つ責任を減らす点である。

まず、一つ目に関してであるが、今回の実装はプロトタイプ版である。したがって、並列化に関しては、単純なレベルのものしか実装しておらず、出力されるコードは貧弱である。しかし、実際に実行されるであろう通信のパターンを、コンパイル時に知ることができ、その情報を利用した、より高度な並列化フェーズが実装可能である。

例えば、DMA 転送をサポートしている並列計算機上での利用を考えた場合、部分実行が生成したログから、通信が続けて行なわれる反復の集合を知ることができるので、DMA がまとまったデータを一度に送信するようなコードを出力する自動並列化コンパイラを実装することも可能である。

次に二つ目のプログラマの責任に関してである。部分実行を用いた並列化を行なう場合、プログラマが持たなければならない責任がいくつかあった。

プログラマが持たなければいけない責任とは、部分実行によるものと、並列化によるものがある。部分実行による並列化は、本質的に、この問題を抱える。現時点の実装においては、この責任をプログラマに任せた。しかし、一般的な並列化コンパイラでは、コンパイラが責任を持つ。この点が、一般的な並列化に対して本論文に基づくコンパイラが持つ弱点である。

この、プログラマがコンパイラに対して持つ責任を、自動化することが、今後の課題である。

参考文献

1) David F. Bacon, Susan L. Graham, Oliver J. Sharp: "Compiler Transformations for High-Performance Computing" *ACM '94 Computing Surveys pp.345 – 419*
2) Michael Wolfe: "HIGH PERFORMANCE COMPILERS FOR PARALLEL COMPUTING" The Addison-Wesley Publishing Company
3) B. K. Rosen, M. N. Wegman, F. K. Zadeck: "Global Value Numbers and Redundant Computations" *ACM SIGPLAN '88 POPL pp.12 – 27*
4) R. sytron, J. Ferrante, B. K. Rosen, M. N. Wegman: "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph" *ACM SIGPLAN '88 PLDI pp.451 – 490*
5) Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren: "The program dependence graph and its use in optimization" *ACM '87 Transactions on Programming Languages and Systems pp.319 – 349*
6) Allan L. Fisher, Anwar M. Ghuloum: "Parallelizing Complex Scans and Reductions" *ACM '94 PLDI pp.135 – 146*
7) Anwar M. Ghuloum, Allan L. Fisher: "Flattening and Parallelizing Irregular, Recurrent Loop Nests" *ACM '95 PPOPP pp.58 – 67*
8) Steven Brawer, 大森 健児 訳: "並列プログラミングの基礎" 丸善株式社