

NAS Parallel Benchmarks の粗粒度並列性解析

石井 亮儀 合田 憲人

東京工業大学

本稿では、マルチグレイン自動並列化コンパイラの開発に関する研究の一環として、プログラムの粗粒度並列性解析を、並列計算機向けベンチマークプログラムである NAS Parallel Benchmarks に対して行った結果を報告する。本解析結果により、マルチグレイン自動並列化コンパイラにおいて有力なデータローカライゼーション手法であるループ整合分割がアプリケーションの 1 つである SP の中の主要なサブルーチンの、実行時のキャッシュミスヒット率を軽減することが発見され、実機上での予備評価により商用自動並列化コンパイラである KAP に対して約 13% の実行時間の短縮が得られることが確かめられた。

Analysis for Coarse Grain Parallelism of NAS Parallel Benchmarks

Akiyoshi ISHII and Kento AIDA

Tokyo Institute of Technology

This paper presents results of analysis for coarse grain parallelism of NAS Parallel Benchmarks. The results showed that the data localization scheme for the multigrain parallelization could reduce cache miss rate for one of application programs, SP. The preliminary evaluation on Sun Ultra 80 showed that the data localization scheme reduced the execution time of a subroutine in SP by 13% compared with commercial parallelizing compiler, KAP.

1 はじめに

逐次的なプログラムから自動的に並列化プログラムを生成する自動並列化コンパイラ分野において、様々な並列化手法が提案されているが、それらの手法は並列化を行う粒度により細粒度、中粒度、粗粒度の 3 つに分類することができる。細粒度並列化とはステートメントまたは演算レベルでの並列化を行うものであり、比較的小さな命令を複数のプロセッサで分担して処理する。中粒度並列化とはループ並列化とも呼ばれ、繰り返し処理を複数のプロセッサで分担して処理する。多くの自動並列化コンパイラが並列化の対象とするのがこのループ並列化であり、様々な形状のループに対応できるよういくつかの並列化技術が開発・実装されている。粗粒度並列化とはループや複数のステートメントを含んだマクロタスクを定義し、そのマクロタスクをプロセッサに割り当てることにより並列処理を行う。一般的な並列化コンパイラでは、細粒度または中粒度の 1 レベルの粒度における並列化を試みるものが多いが、上に述べた粗粒度並列性から細粒度並列性までを階層的に利用することにより、ループ並列化だけでは得られないような性能向上を目指すマルチグレイン自動並列化コンパイラに関する研究も進められている [1]。

並列化コンパイラに関する研究では、性能評価により新しく実装された最適化手法などの効果を正しく測定する必要があるが、ベンチマークプログラム全体の実行時間による従来からの性能評価では特定の最適化手法による効果を直接示すデータは得られない。個別の最適化手法の効果を知るためには、元のプログラムに対して個々の最適化手法によってどの程度性能向上が得られるのかを詳細に解析する必要がある。このような詳細な解析をルー

プ並列化に関して行った研究として Eigenmann らによるもの [2] があるが、同じような観点から、マルチグレイン並列化を視野に入れて粗粒度並列性まで考慮して調査されたものはほとんどなく、マルチグレイン並列化の性能評価において重要な指針となると思われる性質はまだ明らかになっていない。本稿では、既存のベンチマークプログラムの粗粒度並列性を解析し、マルチグレイン自動並列化コンパイラにより期待される性能向上を分析した結果を報告する。

2 マルチグレイン並列化コンパイラ

マルチグレイン並列化コンパイラでは、逐次プログラムであるソースプログラムを解析し、細粒度、中粒度、粗粒度レベルでの並列性を階層的に抽出したコードを出力するコンパイラである。マルチグレイン並列化コンパイラの 1 つである OSCAR コンパイラ [1] では、ソースプログラムとして FORTRAN プログラムを入力し、OpenMP により並列化されたコードを出力する。OSCAR コンパイラにおける粗粒度レベルの並列化では、コンパイラがソースプログラムを疑似代入文ブロック (BPA)、繰り返しブロック (RB)、サブルーチンブロック (SB) の 3 種類のマクロタスク (MT) に分割し、各マクロタスク間の並列性を最早実行可能条件の形式で抽出する。次に、コンパイラが生成するスケジューリングコードがマクロタスクを複数のプロセッサエレメント (PE) から構成されるプロセッサクラスタ (PC) に割り当てることにより、マクロタスク間の並列処理を実現している。

本稿では、OSCAR コンパイラにより抽出可能と予測される粗粒度並列性を解析の対象とする。

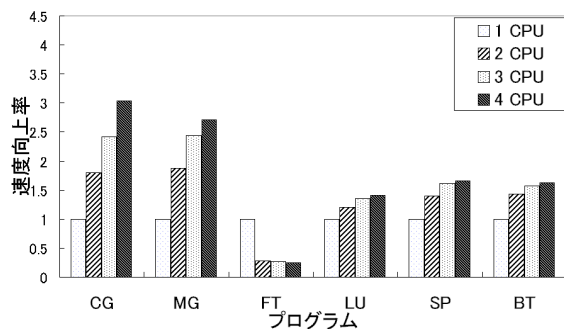


図 1: NPB-KAP による性能向上

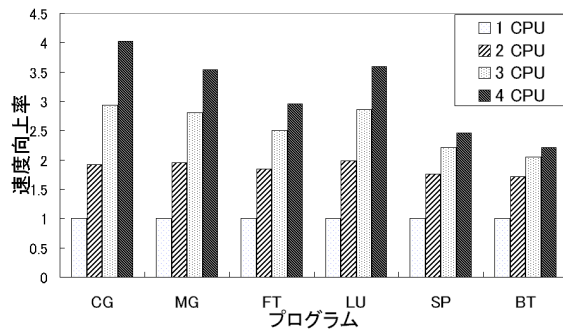


図 2: NPB-OmpC 版による性能向上

3 ベンチマークプログラム

本稿が解析するベンチマークプログラムは、並列計算機のベンチマークとして広く知られている点から、NASA Ames Research Center が公開している NAS Parallel Benchmarks(NPB)[4] の逐次版 (NPB2.3-Serial) とする。NPB については RWCP により、OpenMP を用いて並列化および最適化が施されたコードが公開されており [5], 本コードによりプログラムのループ並列性は明らかになっているが、粗粒度レベルの並列性についてはまだ明らかになっていない。次節では NPB の粗粒度並列性解析の予備評価として行った、NPB のループ並列化に関する性能評価について述べる。

3.1 NPB のループ並列化の性能評価

NPB の逐次版プログラム (以後 NPB-Serial と表記) を Visual KAP for OpenMP Version3.9[6] (KAP と表記) により並列化したプログラム (NPB-KAP と表記) と RWCP により C 言語版 OpenMP であらかじめ並列化された NPB (NPB-OmpC と表記) を SUN Ultra80 上でコンパイルし、その結果を比較した。なお、NPB には 8 個のベンチマークプログラムが含まれるが、そのうち IS については FORTRAN ではなく C 言語で記述されていること、EP についてはメインループが最外周ループで並列化でき、高い並列化性能が出るのが明らかであるため、今回の粗粒度並列性解析の対象からは外してある。また、評価に利用した NPB のクラスは A である。図 1、図 2 にそれぞれ NPB-KAP と NPB-OmpC を用いた場合の並列処理による速度向上率を示す。なお、実験に使用したマシンの構成は表 1 の通りであり、OpenMP プログラムのコンパイラには SUN FORTE for HPC 6 を利用した。

プロセッサ台数に対するスケーラビリティに注目してみると、NPB-Kap の速度向上率は全てのプログラムにおいて NPB-OmpC による速度向上率を下回っていることがわかる。特に FT については複数 CPU を利用することにより逆に実行時間が増大してしまっている。NPB-KAP の結果のうち比較的 NPB-OmpC に近い結果を示しているのが、Kernel ベンチマークの CG と MG であり、Application ベンチマークである LU, SP, BT の 3 つにつ

表 1: 実験に使用したマシンの構成

SUN Ultra80	
CPU	UltraSPARC II 450MHz × 4
2nd cache	4MB
memory	1GB

いては、最大でも 4CPU で約 1.7 倍という低い速度向上率にとどまっている。

まず、KAP による並列化がどの程度適切に行われていたかについて、実際に出力されたコードを詳しく解析した。Kernel ベンチマークである CG, MG については NPB-OmpC のコードと比較して並列化可能であるのに KAP が並列化できなかったコードは存在しない。実際の性能を比較すると NPB-OmpC に比べて性能が劣るのは安全のために用意されたバリア同期等のために CPU が同期待ちなどを行うためと考えられる。FT については台数を増やすほど実行時間が長くなるという結果になってしまっているが、これはもっとも実行時間のかかるメインループにおいて手続き間解析が適切に行われず、実際には並列実行可能なループが逐次で実行されたことに加えて、メインループ内の小さなループが並列化されたことによりスレッドの fork/join によるオーバーヘッドが CPU 台数の増加に伴い増大したことが原因と考えられる。

Application ベンチマークの LU, SP, BT については、KAP はほぼ全てのループを並列化できていたが、ループ内部でスレッドの fork/join が発生するようなコードを出力しており、それによるオーバーヘッドによりスケーラビリティが悪化したと考えられる。

4 NPB の粗粒度並列性解析

3.1 節において、自動並列化コンパイラが出力したコードをあらかじめ並列化されたコードと比較して並列化性能の評価を行った結果を簡単に示したが、マルチグレイン自動並列化コンパイラに対しては同様の方法では性能評価を行うことができない。マルチグレイン並列化を考

慮したベンチマークプログラムの解析が行われてきていなかったため、既存のベンチマークプログラムの粗粒度並列性があまり明らかになっていないためである。

そこでベンチマークプログラムの粗粒度並列性に注目し、マルチグレイン自動並列化によりどの程度の性能向上が見込まれるのかを調査した。

4.1 粗粒度並列性の解析方法

粗粒度並列性解析の手順については、基本的に OSCAR コンパイラが行う手順に準じており、MT 間のデータ依存とコントロール依存を考慮して MT 間の並列性を表現するマクロタスクグラフ [1] を生成する。その際、マクロタスクの実行時間を考慮して複数の MT を 1 つに統合するなどの処理を行っている。NPB には結果表示などのための処理に IF 文を含むブロックが存在するが、これらをまとめたことにより粗粒度並列性解析を行ったマクロタスクグラフにコントロール依存がほとんど存在しない結果となった。また、サブルーチン呼び出しが行われる SB については、その内部についてさらに粗粒度並列性解析を行うことにより、階層的な粗粒度並列性を抽出することができるが、本稿では並列化による性能向上の効果を考慮し、実行時間が大きな SB に絞って階層的な粗粒度並列性解析を行っている。

4.2 Kernel

図 3 は CG について粗粒度並列性の解析を行って得られたマクロタスクグラフである。図中、ノードは MT、ノード間エッジはデータ依存を示している。特に実行時間が長い MT について階層的な解析を行ったが、各階層において粗粒度並列性はほとんど発見することができず、並列実行可能なタスクは、第 1 階層における MT3,4,5,10 など各階層でいくつか見られるが、特定のタスクに実行時間が集中しており、これらの MT 間で並列実行することによる実行時間の短縮はほとんど期待できないことがわかった。

図 4 は MG について粗粒度並列性の解析を行って得られたマクロタスクグラフである。メインルーチンは MT29 であるが、MT8 から MT15、および MT16 から MT24 までのグループにそれぞれ最大で 3 の粗粒度並列性が見られる。これらのグループの実行時間はそれぞれ全体の処理の約 3 割を占めており、粗粒度並列化によりある程度の性能向上が見込まれると考えられたが、実際にはグループ内の実行時間の 80% を占める MT11 と MT13 間、及びグループ内の実行時間の 65% を占める MT19 と MT20 間にデータ依存があり粗粒度並列化による性能向上はほとんど見込めないことがわかった。

図 5 は FT について粗粒度並列性の解析を行って得られたマクロタスクグラフである。最外側の階層も MT がほぼ一直線に繋がった形になっており、メインルーチンである MT17 の内側を階層的に解析した結果においても、その内部に並列実行可能な MT は存在しなかった。

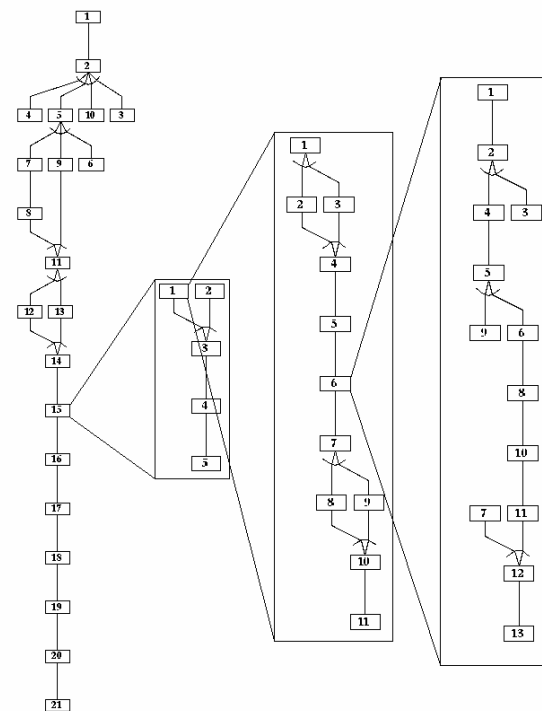


図 3: CG のマクロタスクグラフ

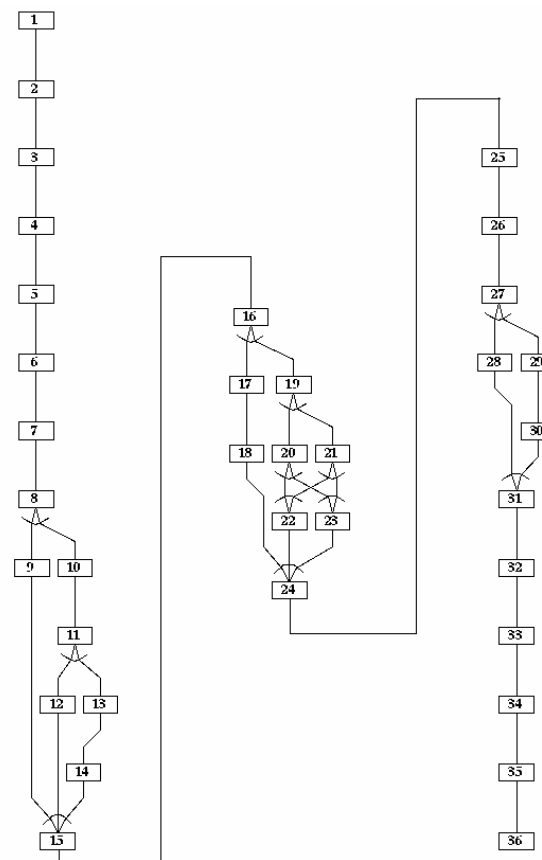


図 4: MG のマクロタスクグラフ

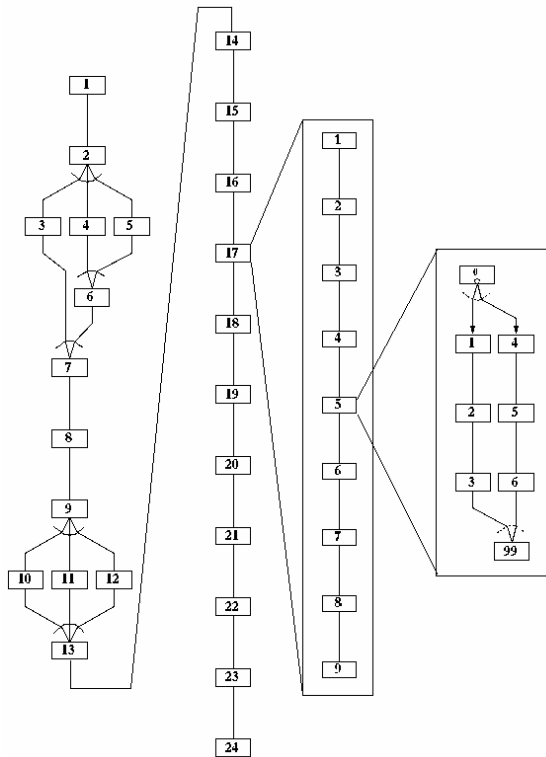


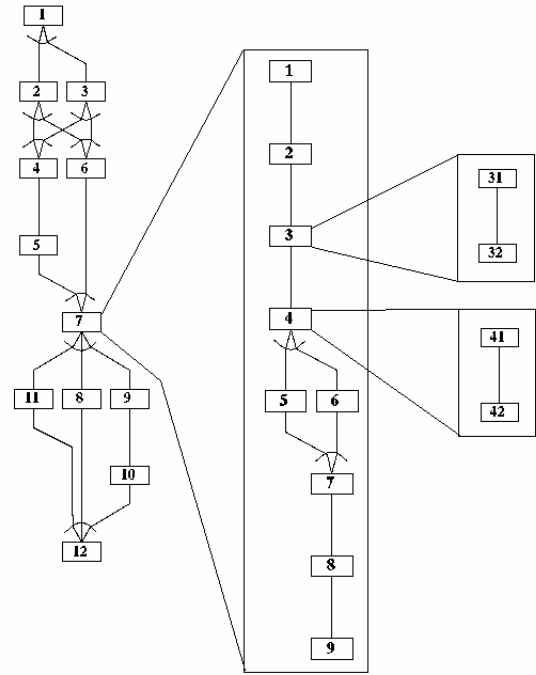
図 5: FT のマクロタスクグラフ

上記 3 つは Kernel ベンチマークを粗粒度並列性解析したものであるが、いずれも性能向上につながる大きな粗粒度並列性を発見することができなかった。Kernel ベンチマークはいずれも比較的単純な処理を抜き出してベンチマークとしたものであり、特定のループあるいはサブルーチンが実行時間の大部分を占めているためループ並列化で大きな並列化効果を得られる可能性があるが、粗粒度並列化による効果は期待できないものと考えられる。

4.3 Application

図 6 は LU について粗粒度並列性の解析を行って得られたマクロタスクグラフである。MT7 が全体の処理の 99%以上を占める非常に大きな MT となっており、MT7 内の MT3,4 がその大部分を占める。MT7 内部の MT5 と MT6 において粗粒度並列性が見られるが、実際にはこの部分の処理は実行時間が短いため、粗粒度並列化による性能向上は小さい。

図 7 は SP について粗粒度並列性の解析を行って得られたマクロタスクグラフである。SP は今までのマクロタスクグラフに比べ複雑な構成となっている。第 3 階層において MT1,3,4,5 と実行時間の長いサブルーチンが次々に呼び出されているのが特徴である。第 4 階層以下の主要なループはほとんどが DOALL ループであるが、第 3 階層以上には DOALL ループは存在しない。そのため深い階層下で並列化を行うことになり、NPB-OmpC でもあまり並列化性能が得られなかった原因となっている。こ



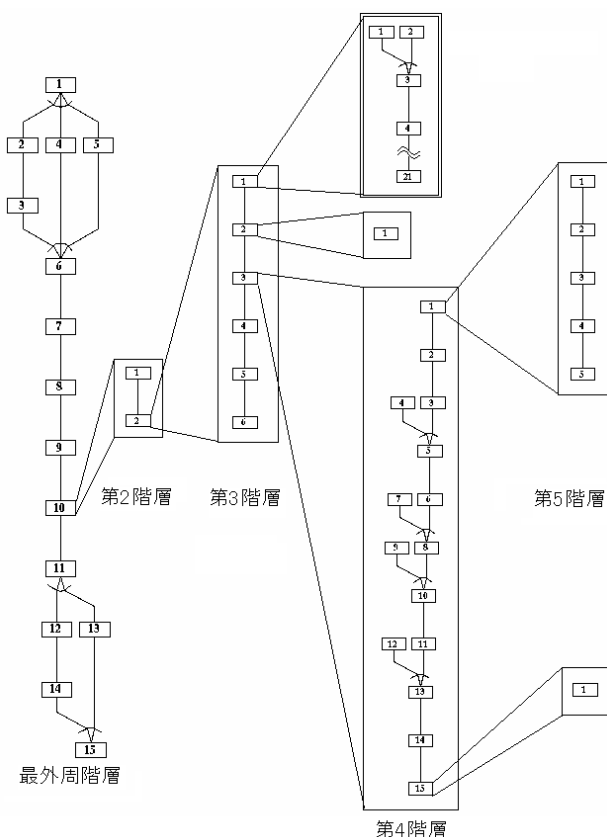


図 7: SP のマクロタスクグラフ

なすことで、この手法の適用が可能である。

5.2 ループ整合分割の適用

ループ整合分割の適用による性能向上が見込まれる対象として、Application ベンチマークの SP における `compute_rhs` というサブルーチンに注目した。図 7 中で 2 重枠で示したタスクグループが `compute_rhs` である。このサブルーチンは `rhs` という 4 次元配列に対して参照 / 代入を行う DOALL ループが 20 個繋がったもので、それぞれのループにはデータ依存があるため粗粒度並列化はできない。 `compute_rhs` というサブルーチンはメインループのイタレーション毎に呼び出され、全体の実行時間に占める割合は約 3 割と大きい。

このサブルーチン内部の処理の特徴として、計算対象である 3 次元のグリッドにおける端部分と内部で処理の異なる計算を必要とする点が挙げられる。非常に簡略化して記述すると以下のようなコードになる。

```
! ループ A
i = 1
do j = 1, n
  R(i,j) = R(i,j) + A
enddo

! ループ B
do i = 2, n-1
  do j = 1, n
```

```
    R(i,j) = R(i,j) + B
  enddo
enddo
```

```
! ループ C
i = n
do j = 1, n
  R(i,j) = R(i,j) + C
enddo

! ループ D
do i = 1, n
  do j = 1, n
    R(i,j) = R(i,j) + D
  enddo
enddo
```

全てのループは DOALL ループであるが、このコードを単純に並列化してしまうとループ A と C ではインデックス変数 j により分割された変数 R の要素がキャッシュに載り、ループ B ではインデックス変数 i により分割された変数 R の要素がキャッシュに載るといった形になる。従って各ループの実行時にキャッシュミスが発生し、性能低下を招く原因となる。

このようなコードに対しループ整合分割を用いることにより、配列の参照範囲を限定し、キャッシュ効率の向上を図ることができる。具体的にはループ A,B,C を実行する際の各 CPU のデータ参照範囲がループ D を並列化した際の各 CPU のデータ参照範囲と同一になるようにループ A,B,C をグループ化してから並列化を行う。

例えば、2CPU 用にループ整合分割を行う場合は以下のようなコードになる。

```
! ----- CPU1 により実行されるセクション
! ループ A
i = 1
do j = 1, n
  R(i,j) = R(i,j) + A
enddo

! ループ B-1
do i = 2, n/2
  do j = 1, n
    R(i,j) = R(i,j) + B
  enddo
enddo

! ループ D-1
do i = 1, n/2
  do j = 1, n
    R(i,j) = R(i,j) + D
  enddo
enddo

! ----- CPU2 により実行されるセクション
! ループ B-2
do i = n/2+1, n
  do j = 1, n
    R(i,j) = R(i,j) + B
  enddo
enddo
```

```

! ループ C
i = n
do j = 1, n
  R(i,j) = R(i,j) + C
enddo

! ループ D-1
do i = n/2 + 1, n
  do j = 1, n
    R(i,j) = R(i,j) + D
  enddo
enddo

```

上記のようにループ A と C については並列化を行わず、ループ B については変数 R のデータ参照がループ D の並列化と一致するように分割する。

上記のような変換を手作業で適用し、実際に実行した結果を図 8 に示す。なお、ループ整合分割では各 CPU 毎に異なるコードを実行する必要があるが、これについては OpenMP の SECTION 文により実現した。ループの分割数は 4 分割とした。図 8 中の実行時間は compute_rhs の開始から終了までを計測した結果である。

グラフは左より、オリジナルプログラムを逐次実行した結果、ループ整合分割を適用したプログラムを逐次実行した結果、KAP により並列化した場合の結果、NPB-OmpC の結果、ループ整合分割により並列化した結果となっている。最初の 2 つについては並列実行可能なコードではないのでプロセッサ台数 1 台の場合の結果のみである。

ループ整合分割を適用したプログラムを逐次実行した場合の結果に注目すると、1CPU で実行したにも関わらずオリジナルプログラムに比べて実行時間が短縮されていることがわかる。その理由として、ループ整合分割を行うことにより生成された小ループ内でデータ参照範囲が CPU の 2 次キャッシュに収まるようになり、その小ループを順次実行したためキャッシュ効率が向上したためと考えられる。

ループ整合分割を行い、並列実行した結果を見ると 1CPU では OpenMP のオーバーヘッドがあるにも関わらずわずかながら性能が向上している。また 2CPU, 4CPU

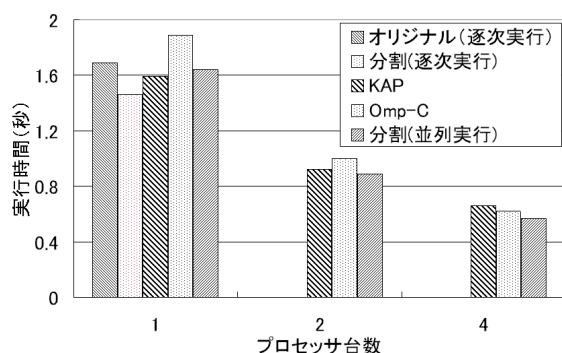


図 8: compute_rhs の実行時間

と CPU 台数を増やしていった際のスケーラビリティも非常に良いことがわかる。ループ整合分割のスケーラビリティがよい理由として、セクション内部での変数 rhs の受け渡しがすべてキャッシュを介して行われるため、低速な共有メモリにアクセスする時間を短縮することができたためと考えられる。

また、他の並列化手法と比較すると、ループ整合分割の適用により、4CPU の場合においては、NPB-OmpC に対して約 8%、NPB-KAP に対しては 13% の性能向上を達成することができた。

6 おわりに

本稿では、NAS Parallel Benchmarks において、マルチグレイン並列化の観点から今まで明らかにされていなかった粗粒度並列性の解析を行い、その特徴と粗粒度並列化の可能性について考察を行った。また、解析を行った結果から、アプリケーションプログラムの 1 つである SP 中にマルチグレイン並列化における有用なデータローカライゼーション手法である階層型ループ整合分割を適用できる可能性のあるサブルーチンを発見し、実際にループ整合分割を適用してその効果を調査した。その結果ループ整合分割が、局所的ではあるが従来型のループ並列化よりも 1 割以上の性能向上を示すことが確認された。

今後の課題としては、ループ整合分割により性能の向上が見込まれる部分をより大域的に調査し、マルチグレイン自動並列化コンパイラによる性能向上の可能性を検討すること、及び結果をコンパイラ開発へのフィードバックすることが挙げられる。

謝辞 本研究の一部は、NEDO アドバンスド並列化コンパイラプロジェクトにより行われた。

参考文献

- [1] 笠原 博徳, 小幡 元樹, 石坂 一久, 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理, 情報処理学会論文誌, Vol. 42, No. 4, pp 910-920, Apr., 2001
- [2] R.Eigenmann, J.Hoeflinger, D.Padua, On the Automatic Parallelization of the Perfect Benchmarks. *IEEE Trans. on parallel and distributed systems*, Vol. 9, No.1, pp 5-23, Jan., (1998)
- [3] OpenMP : <http://www.openmp.org/>
- [4] NAS Parallel Benchmarks : <http://www.nas.nasa.gov/Software/NPB/>
- [5] OpenMP C Version of NPB2.3 : <http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html>
- [6] Visual KAP for OpenMP: <http://www.kai.com/vkomp/>
- [7] 吉田 明正, 越塚 健一, 岡本 雅巳, 笠原 博徳, 階層型粗粒度並列処理における同一階層内ループ間データローカライゼーション手法, 情報処理学会論文誌, Vol. 40, No. 5, pp 2054-2063, May., 1999