計算機アーキテクチャ　128－20
ハイパフォーマンス
コンピューティング　70－20
（1998．3．6）

非均質環境向け並列化コンパイラ *hetero-TINPAR* — 動的負荷分散方式の改良 —

田中慎司[†]　後藤慎也[††]　窪田昌史[†††]
五島正裕[†††]　森眞一郎[†††]　富田眞治[†††]

本稿では、処理性能の異なる計算機が接続された非均質環境向けの並列化コンパイラ *hetero-TINPAR* が生成するコードに組み込まれた、実行時の動的負荷分散方式について述べる。本動的負荷分散方式では、ループのイタレーションを実行時に各計算機の処理能力に応じて割り当てる方式をとっているが、1) 連続したイタレーションしか割り当てることができない、2) 性能の低い計算機にも最低限のイタレーションを割り付ける必要があり、その計算機ために全体の実行性能が低下する、3) 処理の割り当ての変更の有無を判断するために全プロセッサが同期をとる必要があるという問題点があった。これら3つの問題点を改良した結果、それぞれ約10%、15%、10%の性能向上が得られた。

hetero-TINPAR: A Parallelizing Compiler for Heterogeneous Computing Environment — Improvement of the Dynamic Load Balancing Scheme —

Shinji Tanaka,[†] Shin-ya Goto,[††] Atsushi Kubota,[†††]
Masahiro Goshima,[†††] Shin-ichiro Mori[†††] and Shinji Tomita[†††]

In this report, we present the dynamic load balancing scheme at run-time embedded in the code generated by parallelizing Compiler *hetero-TINPAR*, which is oriented to heterogeneous computing environment where computers with different available computing power are connected. In this scheme, iterations of loops in the program are assigned to the processors at run-time. In the conventional scheme, 1)iterations must be assigned to the processors contiguously, 2)at least the minimum amount of tasks must be assigned to each processor even if the processor has the extremely small computing power, which degrades the total performance and 3)all processors must join the barrier synchronization to check the necessity of the load balancing among the processors. By the improvement of these drawbacks, we can achieve the improvement of performance by 10%, 15% and 10% respectively.

1. はじめに

近年、ワークステーションやパーソナルコンピュータは、安価で高性能になってきており、これらの計算機同士は、イーサネットなどによって相互に接続されている。多くのユーザーは、各計算機の上で、単独にジョブを実行させる程度にとどまっていて、計算機資源を有効に利用しているとは言いがたい。

様々な種類の計算機によって構成される非均質な計算機環境では、計算機のアーキテクチャや、各プロセッサの性能が異なる。また、それらの計算機を接続するネットワークも、様々な速度のものが混在し、そのトポロジーも複雑である。さらに、マルチユーザー環境下では他のユーザープロセスの影響によって利用可能な計算機能力が動的に変化する。

そのため、動的に変化する各計算機の能力に応じて、配分するタスク量を調整しなければ、各計算機の計算能力を最大限に引き出すことはできない。我々はこのような環境に対応するため、各計算機の計算能力に応じてタスクを配分する非均質環境向け並列化コンパイラ *hetero-TINPAR* を開発している。

hetero-TINPAR では、実行時の計算能力の変化に対応するため、動的に負荷を分散する機能を提供している。しかしながら、従来の *hetero-TINPAR* では、負荷分散機能は比較的単純なものであり、各計算機へのタスク割り当てが柔軟ではない、動的負荷分散のオーバーヘッドが大きいなどの問題があった。そのため、現実の様々な環境やアプリケーションに対して、

† 京都大学工学部
Faculty of Engineering, Kyoto University
†† 住友電気工業（株）
Sumitomo Electric Industries, Ltd
††† 京都大学大学院工学研究科
Graduate School of Engineering, Kyoto University

常に最適であるとは言いがたかった。

本研究では、柔軟なタスク割り当てを可能にし、かつ、動的負荷分散のオーバーヘッドの低減する手法を提案する。これにより、より多くの環境とアプリケーションに対応することが可能となる。

以下、第2章では、従来の動的負荷分散方式を含めた *hetero-TINPAR* の概要について述べる。第3章では本研究で提案する動的負荷分散方式について述べる。第4章では提案した手法の評価を行い、第5章でまとめる。

2. *hetero-TINPAR* の概要

hetero-TINPAR では、非均質環境に対応するために、**抽象プロセッサ**という概念を導入し、この概念を応用して、動的負荷分散を実現している。本章では、抽象プロセッサと動的負荷分散機能の概要について述べる。

2.1 抽象プロセッサ

非均質環境では、処理能力の異なるプロセッサに、その処理能力に応じた量の負荷を分散させることによって高速実行が可能となる。このため、均質環境に対するようなブロック分割や、サイクリック分割などの単純な規則的な分割によるデータの割り付けはできない。ところがこれらの分割以外の任意の分割を行った場合は、データを所有するプロセッサの判定式が単純な式にはならず、実行時のオーバーヘッドが大きくなる。

そこで *hetero-TINPAR* では、均質環境の時の規則的なデータ分割によるデータの所有者判定の単純さを非均質環境においても利用するため、仮想的な抽象プロセッサ (Abstract Processors) という概念を導入する。また、コンパイラの最終的なコード生成の対象となる実プロセッサを以下のように定義する。

抽象プロセッサ　処理能力が等しい要素プロセッサで構成された仮想的並列計算機を想定し、その要素プロセッサを抽象プロセッサと呼ぶ。データは、各抽象プロセッサに均等に分割される。抽象プロセッサの数は、ユーザーがソースプログラムの中でディレクティブの形で与える。

実プロセッサ　コンパイラが最終的なコード生成の対象とする物理的なプロセッサを実プロセッサと呼ぶ。実行時には、実プロセッサの処理能力に応じた個数の抽象プロセッサが配分される。

hetero-TINPAR では、実行文のスケジューリング方式として、分割された配列データを所有するプロセッサがそのデータに対するアクセスを行う文を実行する Owner Computs Rule を採用している☆。

データは各抽象プロセッサに均等に分散されており、また、各データに対する処理は均質なものと仮定するため、各抽象プロセッサの演算処理量も均質となることが期待できる。

hetero-TINPAR は、抽象プロセッサの概念を応用し、以下のように非均質環境に対する最適な負荷分散をする。

(1) データを抽象プロセッサに均質に分散する。
(2) 実プロセッサの処理能力に比例した個数の抽象プロセッサをその実プロセッサに配分する。
(3) 具体的に、どの抽象プロセッサをどの実プロセッサに割り付けるか決定する。

(1) では、均質環境向けの並列化を行うだけであるので、従来の均質環境向けのコンパイル手法をそのまま利用することができる。

(2) と (3) で、本研究の対象である動的負荷分散機能を提供する。これらに関しては第2.2節で述べる。

各実プロセッサは、自分に配分された各抽象プロセッサに均質配分されたタスクを処理する。ただし、各実プロセッサではスレッドを1つだけ起動し、そのスレッド内で複数の抽象プロセッサに対する処理を行う。

各抽象プロセッサが実行すべきタスクは、各抽象プロセッサ毎に配列の参照範囲の異なるループとなる。1つの実プロセッサ内では、これらのループはその初期値、終了値を実行時に変更することで1つのループとすることで、タスクが融合されることになる。その詳しい処理方式は[1]を参照されたい。

2.2 動的負荷分散機能

非均質環境において、高速に並列実行するためには、各計算機に対して、その処理能力に応じた適切な量のタスクを割り振る必要がある。事前に最適な各計算機の処理能力比を与えられない場合、これを自動的に求め、それに従って負荷分散を行う必要がある。また、各計算機の処理能力が動的に変化する場合、これの変化に対応するために、動的負荷分散を行い、時間と共に変動する計算機の能力に対して最適な計算量の割り当てをする必要がある。

これらを実現するために、実行時に計算機の負荷を実測し、それをもとに適切なタスク分散比を求める。求められたタスク分散比に応じて、実行時に各実プロセッサへのタスクの割当てを変更する、動的負荷分散機能が必要となる。

また、*hetero-TINPAR* は、Owner Computes Rule を採用しているため、抽象プロセッサの再配分は実際には、そのプロセッサに割り当てられているデータの転送によって行なわれる。以下、単に**抽象プロセッサの移動**とは、抽象プロセッサに割り当てられているタスクとデータを移動させることを表わすものとする。

hetero-TINPAR では、計算に参加している全プロセッサが、ある定められた時点で、協調して負荷分散を行う調歩協調型の負荷分散戦略を採用している。

再分散は、抽象プロセッサに割り当てられたタスク

☆ 代入文は原則として左辺のデータを所有するプロセッサが実行する。

の最外ループの1イタレーションに対応する実プロセッサの処理が終わった時点で行なうものとする。これをチェックポイントと呼ぶ。チェックポイントでは、全実プロセッサが並列処理を一時停止して同期をとることが可能である。

チェックポイントでは、一つ前のチェックポイントで配分された抽象プロセッサ数と、そのチェックポイントからの計算時間を負荷情報として利用し、再分散が必要と判定すれば再分散を行ない、そうでなければ再分散を行なわずにチェックポイント以降の実行をそのまま継続する。

2.3 従来の動的負荷分散機能の問題点

従来の *hetero-TINPAR* では、動的負荷分散機能が、比較的単純な機構によって実現されていたが、それでは、様々な計算機環境、様々なアプリケーションに対して、常に最適な負荷分散が実現されているとは言いがたい。

問題点としては、以下のようなものがあげられる。

(1) 抽象プロセッサの割当方式に、単純な連続割当のみが可能である。このため、計算機の配置による動的負荷分散に要する時間への影響が大きい。

(2) 全ての実プロセッサに抽象プロセッサを少なくとも1つは配分しなければならない。このため、非常に処理能力の低い実プロセッサが存在した場合、その実プロセッサの抽象プロセッサ1つ分の処理のために、全体の実行時間が増大する。

(3) 再分散のための負荷情報収集のために、実際には再分散を行わなくても、大きなオーバーヘッドを伴う全実プロセッサの同期が必要となる。

そこで、第3.1節で、1番の問題点に対して、新たに最小移動コスト方式による対処を提案する。第3.2節で2番の問題点に対して、他に高負荷のユーザープロセスが存在するなどの理由によって、処理の力が低くなっている実プロセッサに抽象プロセッサを配分しないことによる対処を提案する。第3.3節で3番の問題点に対して、自律負荷判断手法による対処を提案する。

3. 動的負荷分散機能の改良

本章では、従来の *hetero-TINPAR* の動的負荷分散機能に対する改良について述べる。

3.1 適応型抽象プロセッサの割当方式

3.1.1 連続割当方式と最小移動コスト方式

hetero-TINPAR の動的負荷分散機能では、各実プロセッサに対して配分する抽象プロセッサの個数を決定したのち、具体的に、どの抽象プロセッサのどの実プロセッサに割り当てるかを決定する。(以下、配分すると割り当てるは、上記の意味で使用する。)

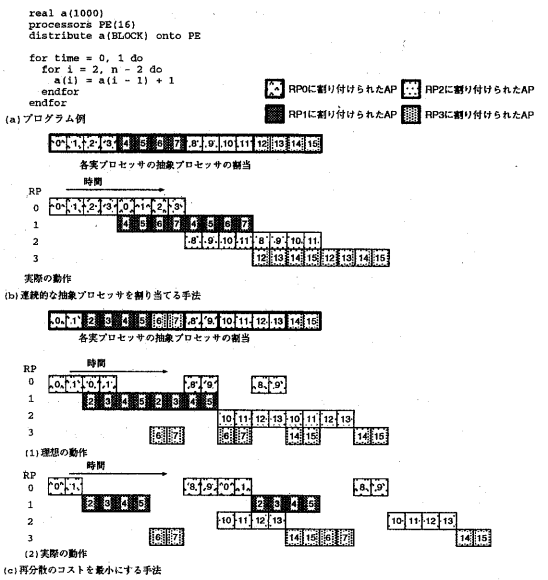

図1 プログラムにループ運搬依存がある場合

抽象プロセッサを実プロセッサへ割り当てる方法には、実プロセッサの番号順に抽象プロセッサの連続した領域を割り当てていく連続した抽象プロセッサを割り当てる手法 (**連続割当方式**) と動的再分散に伴って必要となる実プロセッサ間の抽象プロセッサの移動をできるだけ削減して再分散の時の通信コストを最小に抑える手法 (**最小移動コスト方式**) が考えられる。

ここで、SOR などのプログラムでは、抽象プロセッサ数の割り当てが連続していなければパイプラインが乱れ、性能低下が起こる。そこで、このような場合には連続割当方式を採用し、そのほかの場合は、再分散時のオーバーヘッドが小さくなるように、最小移動コスト方式を採用する。現在のところ、パイプライン実行が行なわれるかどうかは、ループ運搬依存が存在し、かつ、その外側にもう1段ループが存在する場合としている。

最小移動コスト方式をとったために、パイプラインが乱れる問題ついては次節で詳しく述べる。

3.1.2 最小移動コスト方式の問題点

SOR のように、パイプライン実行により並列性が抽出されている場合、最小移動コスト方式では、そのプログラムの並列性が発揮されない。図1は、図1(a)のプログラムの実行例を示している。図1(b), (c) それぞれ連続割当方式、最小移動コスト方式を採用した場合の実行例である。

図1(b) の連続割当方式では、抽象プロセッサはそれぞれ4つずつ順番に実プロセッサに割り当てられていっている。実際の実行では、まず、実プロセッサ0が抽象プロセッサ0から3の1回目の反復を処理した後、データを実プロセッサ1に転送する。その後、実

プロセッサ0は、2回目の反復の実行を開始し、実プロセッサ1は、抽象プロセッサ4から7の1回目の反復の実行を開始する。以下順次、パイプライン方式で実行が進む。

図1(c)の最小移動コスト方式の例では、抽象プロセッサ0,1,8,9は実プロセッサ0に、抽象プロセッサ2から5は実プロセッサ1に、抽象プロセッサ10から13は実プロセッサ2に、抽象プロセッサ6,7,14,15は実プロセッサ3に割り当てられている。図1(c)(1)は、その理想的な実行の様子を図示したものである。実プロセッサ0は、まず抽象プロセッサ0,1の1回目の反復の処理をした後、実プロセッサ1にデータを転送する。その後、実プロセッサ0が抽象プロセッサ8,9の1回目の反復の処理をするには、実プロセッサ3からデータを受信する必要がある。この受信の待ち時間に、抽象プロセッサ0,1の2回目の反復の処理を先に実行することにより高速実行している。

しかし、第2.2節で述べたように、*hetero-TINPAR*では、1つの実プロセッサに割り当てられている全抽象プロセッサをひとつのスレッドで処理しているため、同じ実プロセッサに割り当てられている全ての抽象プロセッサが、並列化するために分割されているループを処理し終わるまでは、次の処理に進むことはできない。したがって、この制約のために、実際の動作は、図1(c)(2)のようにつまり、抽象プロセッサ0,1,8,9が同じ実プロセッサに割り当てられている場合、抽象プロセッサ0,1の2回目の反復の処理と抽象プロセッサ8,9の1回目の反復の処理は、本来はどちらからでも実行できるにも関わらず、この制約によって後者を先に計算しなければならない。そのため、並列性が発揮されず、実行時間は大幅に増加する。

3.2 抽象プロセッサを配分されない実プロセッサ

従来の*hetero-TINPAR*では、各実プロセッサに対し少なくとも一つは抽象プロセッサを配分する必要があったが、抽象プロセッサが配分される数がユーザーが指定した閾値より少ない場合は、その実プロセッサに抽象プロセッサを配分しないようにした。

処理能力の低い計算機や、他のユーザープロセスによって高負荷になっている計算機が含まれている場合、その計算機に抽象プロセッサを配分するよりも、配分せずに通信のオーバーヘッドを減らした方が、実際の計算時間は短くなることがある。また、実プロセッサの処理能力比が極めて大きくなると、抽象プロセッサの粒度は固定されているため、その粒度が十分細かくない場合、抽象プロセッサの配分比が実プロセッサの処理能力比に対応できない場合が起こり得る。このような場合、処理能力の低い実プロセッサに割り当てられた抽象プロセッサの処理に時間を要するため、全体の処理が遅くなることがある。このような場合は、提供可能な処理能力の低い実プロセッサには、抽象プロセッサを配分しない方が良い。

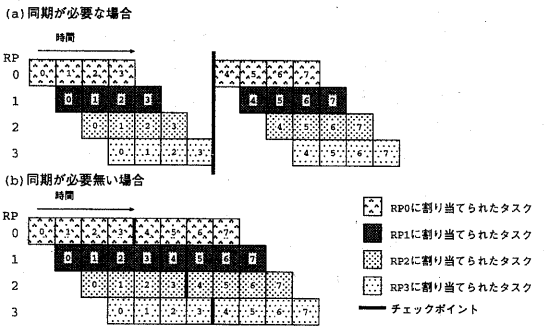

図2 同期が必要な場合と不要な場合

3.3 自律負荷判断方式

*hetero-TINPAR*では、再分散をするかどうかを判断するだけで、全実プロセッサの間で同期を取る必要があったが、現在の抽象プロセッサの配分比が最適であるかどうかの判断基準を、各チェックポイントにおいて前回のチェックポイントからの計算時間を計測し、その時間が変化しているかどうかとすることにより、同期を取らずに再分散するべきかどうかの判断を可能にした。これにより、再分散の必要性の判断のオーバーヘッドが大幅に減少した。

プログラムの実行の途中で全体で同期を取ると、パイプライン型の並列実行をするアルゴリズムを採用しているアプリケーションの場合には、オーバーヘッドが大きい。図2(a)に、このようなプログラムの実行中に同期を取った場合の実行の様子を示す。この図のプログラムは、ある実プロセッサがあるブロックを処理するには、1つ前の番号の実プロセッサが同じ番号のブロックの処理を終えている必要がある。パイプライン型の並列実行をするアルゴリズムを採用しているアルゴリズムでは、実行開始から実際に並列動作されるようになるまで、および同期点に全ての実プロセッサが到達するまでには、ある程度の時間がかかるので同期をとるたびにこの時間がオーバーヘッドとして積み重なっていくからである。

ここで、一旦、動的負荷分散が行われたあとでは、計算時間が変化していない間は、抽象プロセッサの配分比は最適であるが、計算時間の変化が検出されたときに、現在の抽象プロセッサの配分比が最適でなくなったと判定可能である。そこで、各チェックポイントにおいて、前回のチェックポイントからの計算時間を計測し、その時間が変化しているか否かで負荷の再分散が必要であるかどうかを判断するように改良を行なった。

ある実プロセッサが再分散が必要であると判断したときは、それを全体に通知する。この時に、自分が再分散可能な時点を指定する。この通知を受け取った他の実プロセッサは、同様に自分が再分散可能な時点を他の実プロセッサに通知する。各実プロセッサは、受

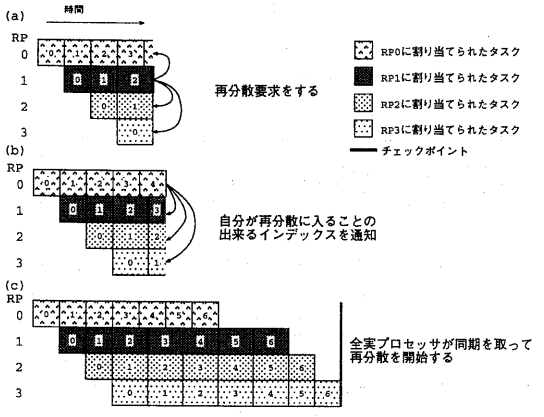

図3　再分散の必要性の非同期判断

け取った再分散可能な時点の中で、一番、処理の進んでいるところまで、処理を進め。そこで、全プロセッサで同期を取って再分散を開始する。

この手法により、再分散の必要性の判断のオーバーヘッドが極めて小さくなる。図2(b)に、パイプライン型の並列実行をするプログラムが再分散を行なう場合の実行の様子を示す。

また、従来は再分散の必要性の判断のためのオーバーヘッドを低減するために、チェックポイントを通過するたびに判断をするのではなく、ユーザーに指定された頻度で判断をすることによって判断の回数を減らしていたが、毎回判断をすることが可能になった。これにより、負荷の変化に敏感に反応して再分散をすることが可能になる。

4. 性能評価

本章では、動的負荷分散機能の改良によって得られた高速化の評価を行なう。

4.1 評価環境

評価には、SOR 法と Red-Black SOR(以下、RB-SOR) 法の 2 つのプログラムを用いた。使用した行列サイズは 1024×1024 である。特にただし書きのない場合は、抽象プロセッサの個数は 128 個とした。

プログラムの実行は、10Mbps のイーサネットスイッチにより接続された SPARCstation 20(以下 SS20)4 台、SPARC Ultra-1(以下 Ultra) 4 台と SPARC Classic1 台 (以下 Classic) により構成された環境において最大 4 台構成で行い、原則的に、他のユーザープロセスの無い無負荷の状態で測定を行った。また、通信ライブラリとして MPI[2] の実装処理系である MPICH[3] を使用した。

4.2 抽象プロセッサの配置の評価

抽象プロセッサの割り当ての時の、連続抽象割当方式と、最小移動コスト方式のそれぞれを採用したときの実行時間、計算時間、及び通信時間を比較し、両手法の評価を行った。

表1　抽象プロセッサの配置の評価

計算機の順番	使用手法	実行時間 [s]	計算時間 [s]	通信時間 [s]	再分散時間 [s]
SOR					
SUUS	最小	280.29	179.16	65.43	37.03
SUUS	連続	284.15	183.12	63.94	39.90
UUSS	最小	843.56	181.51	596.69	38.85
UUSS	連続	360.75	181.41	71.16	109.86
RB-SOR					
SUUS	最小	165.12	78.57	34.09	52.44
SUUS	連続	165.61	75.83	32.49	57.24
UUSS	最小	166.34	80.55	30.78	55.60
UUSS	連続	187.03	78.66	27.82	80.66

表2　抽象プロセッサを配分されない実プロセッサの評価

†	‡	最終分散比	実行時間 [s]	計算時間 [s]	通信時間 [s]	再分散時間 [s]
128	2	57:57:14:0	139.06	53.97	15.19	69.91
	0	57:57:12:2	137.96	42.30	26.82	68.90
64	2	29:28:7:0	137.73	54.06	13.86	69.81
	0	28:28:6:2	132.95	43.78	21.17	68.06
32	1	14:14:4:0	133.65	56.06	12.63	64.96
	0	14:14:3:1	144.05	54.47	25.43	64.14
16	1	7:7:2:0	133.89	58.03	11.77	62.59
	0	6:7:2:1	162.62	63.64	25.68	67.85

†:抽象プロセッサ数
‡:抽象プロセッサを配分しないようにする閾値

表1 に実行時間、及びその内訳を示す。実行時間、計算時間、通信時間、再分散時間はそれぞれ、プログラムの実行に要した時間、実行時間から通信時間を除いた時間、再分散以外の通信に要した時間、再分散時の通信に要した時間である。

SOR ではパイプライン実行を乱さない連続割当方式の方が再分散以外の通信時間が減少する。これは、通信時間の中には、パイプラインの乱れに伴って増大する受信待ち時間も含まれるためである。RB-SOR のようにループ運搬依存がなければ、最小移動コスト方式の方が、再分散時の通信量の減少による実行時間全体の減少が得られる。

RB-SOR を UUSS の配置で実行した場合に、最小移動コスト方式を採用した効果が最も大きく現れ、約 10%実行時間が減少している。

4.3 抽象プロセッサを配分されない実プロセッサの評価

Ultra 2 台、SS20 1 台と、Classic 1 台という構成で

表3　再分散の間隔による変化

再分散の間隔	実行時間 [s]	計算時間 [s]	通信時間 [s]	再分散時間 [s]
再分散のチェック時に同期をとる場合：				
1	140.81	45.97	30.70	81.79
5	109.52	46.33	18.59	48.21
10	103.07	45.88	18.57	40.37
50	101.91	47.48	18.97	35.74
再分散のチェック時に同期とらない場合：				
1	97.68	44.45	17.10	35.97

評価を行った。評価プログラムは、反復回数は500回のSORを利用し、抽象プロセッサ数を128、64、32、16と変えて評価を行った。全体の抽象プロセッサが128個か64個の場合は、抽象プロセッサが配分数が2つ以下の場合、全体の抽象プロセッサが32個か16個の場合は、抽象プロセッサが配分数が1つの場合は、全く配分しないとした。

表2に実行結果を示す。Classicには、抽象プロセッサを配分しないと、抽象プロセッサ数が、32個か16個の場合は実行時間が短くなっている。最大で、抽象プロセッサ数が16個のときに約15%の性能向上が得られる。一方、抽象プロセッサが64個以上の時は、全体の実行時間が3%ほど増大している。

これは、抽象プロセッサ数が少ない方が負荷分散の粒度が大きくなり、計算能力の高い実プロセッサと低いプロセッサの性能比にあわせた負荷分散ができなくなっているためである。

4.4 自律負荷判断方式の評価

自律負荷判断方式による同期を取る回数の減少によるオーバーヘッドの低下を評価する。評価には、Ultraを2台とSS20を2台を、Ultra、SS20、SS20、Ultraという順番で配置した環境でSORを実行して行なった。

4.4.1 負荷が動的に変化しない場合

再分散の必要性の判断のために同期をとったときのオーバーヘッドを見るために、同期を必要とする場合のチェックの頻度を変化させた。

表3に評価の結果を示す。同期を必要とする従来の手法の場合、毎回チェックをすると、50回に1回行う場合に比べて、実行時間が40%も増加している。非同期による判断の場合は、毎回チェックを行っても50回に1回同期を取ってチェックを行う場合に比べて、実行時間はわずかであるが短縮されている。これにより、非同期判断の場合の判断のためのオーバーヘッドは十分に小さいことが示された。

4.4.2 負荷が動的に変化する場合

SORのプログラム(タスク1)の実行開始の52秒後に、2台のSS20のうちの1つに外的負荷として逐次環境用のSORのプログラム(タスク2)も実行した。

表4に実行結果を示す。従来の手法では、59.03秒の時点で再分散を開始している。それに対して、非同期判断の手法では、53.38秒の時点で再分散を開始している。タスク2の実行による負荷の変化は、52秒後に発生するので、改良後の手法では、負荷が変化するとすぐにそれを感知して動的再分散が実行されている。

表4　動的再分散による負荷分散の評価

最終分散比	タスク2の投入に反応して再分散を開始した時間 (s)	タスク1の実行時間 (s)	タスク2の実行時間 (s)
再分散のチェック時に同期をとる場合：			
48:18:15:47	59.03	141.54	377.36
再分散のチェック時に同期をとらない場合：			
49:19:12:48	53.33	130.82	342.78

そのため、タスク1、タスク2の実行時間は、従来の手法では、それぞれ141.54秒と377.36秒であったのが、自律負荷判断による手法では、それぞれ130.82秒と342.78秒となった。従来の手法に比べて、非同期判断による手法では、タスク1、タスク2共に8%ほど早くなっている。

5. おわりに

本稿では、非均質環境を対象とした並列コード生成を行う並列化コンパイラ *hetero-TINPAR* の動的負荷分散機能の改良について述べた。抽象プロセッサの割当方式を各アプリケーションと、各プロセッサの性能に応じて柔軟に変化できるようにした。また、再分散をするかどうかの判断を、同期を取らなくとも可能としたことによって、動的に変化する計算機環境に対して少ないオーバーヘッドで対応できるようになった。

性能評価により、多様なアプリケーションに対応できるようになったことで、改良前と比較して最大10%の実行時間の短縮を図ることができた。処理能力の低い実プロセッサに、抽象プロセッサを割り当てないことによって最大15%の性能向上が得られた。また、動的に変化する計算機環境に対して迅速に対応できるようになったことで、自らと、他のユーザープロセスの両方が10%程度の性能向上が得られた。

謝辞　本研究の一部は、文部省科学研究費補助金 特別研究員奨励費 課題番号00093007による。

参考文献

1) 後藤慎也 他：並列化コンパイラTINPARによる非均質計算機環境向けコード生成手法, JSPP'97, pp.205–212,1997.
2) Message Passing Interface Forum: MPI: Message-Passing Interface Standard,1995.
3) Bridges,P. et.al: Users' Guide to MPICH, a Portable Implementation of MPI, Argonne National Laboratory.