Omni/SCASH のループ再分割を用いた動的負荷分散拡張の実装と評価

栄　　純 明[†1]　松 岡　　聡[†2]
佐 藤　三 久[†3]　原 田　　浩 [†4]

ハイパフォーマンスコンピューティング（HPC）の分野で，PC/WS をノードとする大規模なクラスタがますます重要なプラットホームとなりつつある．このようなコモディティクラスタ環境では，プロセッサテクノロジーの急速な進歩，ユーザーの要求など様々な理由から段階的なアップグレードが行われ，ノード間に性能のヘテロ性が生じるケースが増えている．これはロードインバランスの原因の一つであり，なんらかの動的負荷分散機能が必要である．本論文では Software Distributed Shared Memory (SDSM), SCASH 上の OpenMP の実装である Omni/SCASH に対して我々が現在行っている動的負荷分散機能拡張に関して報告する．動的負荷分散機能を用いることによって，アプリケーションプログラマが明示的にデータやタスクの配置を指定することなく，ランタイムシステムによって性能ヘテロなクラスタにおいてロードバランシングが行える．

Implementation and Evaluation of Dynamic Load Balancing Using Loop Re-partitioning on Omni/SCASH

Yoshiaki Sakae,[†1] Satoshi Matsuoka,[†2] Mitsuhisa Sato[†3]
and Hiroshi Harada [†4]

Increasingly large-scale clusters of PC/WS continue to become majority platform in HPC field. Such a commodity cluster environment, there may be incremental upgrade due to several reasons, such as rapid progress in processor technologies, or user needs and it may cause the performance heterogeneity between nodes from which the application programmer will suffer as load imbalances. To overcome these problems, some dynamic load balancing mechanisms are needed. In this paper, we report our ongoing work on dynamic load balancing extension to Omni/SCASH which is an implementation of OpenMP on Software Distributed Shared Memory, SCASH. Using our dynamic load balancing mechanisms, we expect that programmers can have load imbalances adjusted automatically by the runtime system without explicit definition of data and task placements in a commodity cluster environment with possibly heterogeneous performance nodes.

1. は じ め に

近年，HPC 向けの計算機の多くは SMP を多数結合したクラスタの形態を取る物が増加している[10)]．とりわけ汎用品を構成要素とするコモディティクラスタはコストパフォーマンスや管理の容易性などの面から，多くの研究機関や大学で使用されている．しかしながら，コモディティクラスタではプロセッサやネットワークテクノロジーの急速な進歩や費用面の問題からノードの段階的な追加や，プロセッサやメモリの段階的な増設が比較的頻繁に起こりノード間性能のヘテロ性が生じ，ロードインバランスの原因の一つとなっている．またノード間性能にヘテロ性が無いケースでもマルチユーザー環境がノード間の負荷バランスを崩し，結果的にロードインバランスにつながるケースもある．

環境およびアプリケーションごとにプログラマが明示的にロードバランシングを行うのは困難であり，下位のランタイムシステムによる自動負荷調整が必要とされている．この問題に対処するために，我々はコモ

†1 東京工業大学大学院情報理工学研究科 数理・計算科学専攻
Tokyo Institute of Technology
†2 東京工業大学大学院情報理工学研究科 数理・計算科学専攻 / JST
Tokyo Institute of Technology / JST
†3 筑波大学 電子・情報工学系 計算物理学計算センター
Tsukuba University
†4 ヒューレットパッカードジャパン
Hewlett-Packard Japan,Ltd.

ディティクラスタ向けに OpenMP をプログラミングインターフェースとし，データの再配置および動的負荷分散を自動的に行うシステムを開発している．具体的には，性能的にヘテロな環境においての自動負荷分散機能として，実行時性能モニタリングに基づくループ再分割機能を SDSM SCASH 上の OpenMP の実装である Omni/SCASH[8] に実装した．本論文では Omni/SCASH のループ再分割機能拡張の実装と評価および，現在行っている SCASH レベルの page fault counting に基づく動的ページマイグレーション機能の概要に関して報告する．これらの機能を用いることによって，プログラマが明示的にデータやタスクの配置を指定すること無く，ランタイムシステムによって自動的にロードバランシングが実現できると期待している．

2. Background

2.1 Omni OpenMP Compiler

Omni OpenMP コンパイラは OpenMP プログラムを入力とし，ランタイムライブラリ呼び出しを含むマルチスレッド C プログラムを生成する translator である．フロントエンドとして C-front および F-front がありそれぞれ C および Fortran コードを入力とし，中間コードである Xobject を出力する．Xobject の表現形式はデータタイプ情報を持った AST (Abstract Syntax Tree) で，その各ノードはソースコードの各要素を表現する Java のオブジェクトであり，Exc Java tools により容易に変換可能である．Exc Java tools はプログラムの Xobject の表現レベルでの解析や変換を実現するクラスやメソッドを提供する Java のクラスライブラリ群である．OpenMP プログラムのマルチスレッド C プログラムへの変換はこの Exc Java tools を用いて実現されている．生成されたマルチスレッド C プログラムはネイティブバックエンドコンパイラによってコンパイルされ，ランタイムライブラリとリンクされる．

2.2 SCASH

SCASH[4] は低レイテンシかつ広バンド幅な通信ライブラリである PM[9] と，OS の提供するメモリプロテクションなどのメモリ管理機能を用いて実現されているページベースの Software Distributed Shared Memory System である．SCASH のメモリモデルはマルチプルライタプロトコルを用いた Eager Release Consistency (ERC) であり，ユーザーレベルのランタイムライブラリとして実装されている．

ページ単位の一貫性維持プロトコルとしては invalidate および update の両方が実装されており，実行時に選択可能となっている．

SCASH では *home* ノードでページの最新データと，ページを共有しているノードの集合を保持するページディレクトリを管理し，*base* ノードが最新の home ノードを常に記録している．すべてのノードは，全ページの base ノードを記録しており，最新の home が不明になったノードは base に home を問い合わせることで最新の home を得ることができる．

2.3 OpenMP プログラムの SCASH への変換

OpenMP のプログラミングモデルでは，特に指定しない場合グローバル変数は共有される．その一方で SCASH ではグローバルスコープで宣言された変数はプロセッサプライベートに割り当てられ，共有領域は実行時にメモリアロケーションプリミティブを用いて明示的に割り当てる必要がある．OpenMP プログラムを SCASH の “shemem メモリモデル” に変換するために，コンパイラはグローバル変数を実行時に共有アドレスに確保するように変換する必要がある．

具体的には Omni コンパイラは OpenMP プログラムを以下のように変換する．

(1) グローバル変数の宣言はすべて，共有領域に割り当てられたデータへのポインタに変換される．
(2) すべてのグローバル変数の参照を，対応するポインターを介しての間接参照に書き換える．
(3) コンパイル単位ごとにグローバルデータの初期化を行う関数を生成する．この関数が共有領域へのオブジェクトの割り付けを行い，そのアドレスを対応する間接ポインタに格納する．

OpenMP のディレクティブは， SCASH のプリミティブを用いて同期やプロセッサ間通信を行うランタイムファンクションに変換される．

OpenMP の parallel ディレクティブでアノテートされた逐次プログラムを fork-join モデルの並列プログラムに変換する際，コンパイラは各 parallel リージョンを独立したサブファンクションとしてくくり出す．マスターノードはランタイムファンクションを用いてスレーブスレッドを呼び出し，このサブファンクションを並列実行させる．各ノードのすべてのスレッドはプログラムの起動時に作成され，スレーブノードで fork が行われるまで待機している．また，Omni/SCASH では nested parallelism はサポートしていない．

SCASH ではすべての共有メモリ領域の一貫性はバリア同期の際に取られる．これは OpenMP のメモリモデルと対応している．OpenMP のロックと同期操作は特定のオブジェクト（e.g. ロック変数）に対する

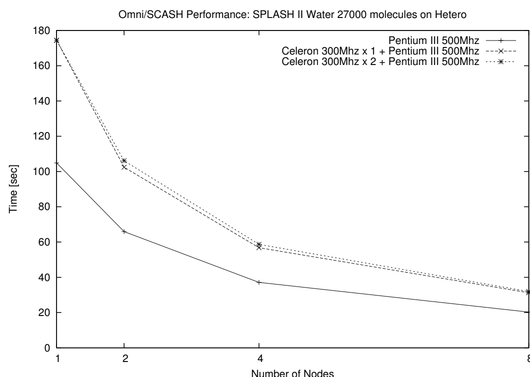


図 1 性能ヘテロなクラスタにおける SPLASH II Water の実行時間

SCASH の一貫性保持プリミティブを明示的に用いて実現されている．

2.4 性能ヘテロな環境での性能低下

実行時負荷分散拡張に関して述べる前に，ここで性能ヘテロな環境における典型的な性能低下の例を示しておく．

図 1 に性能ヘテロな環境で SPLASH II の Water を動作させたときの性能を示す．このクラスタは CPU 性能のみヘテロなクラスタであり Pentium III 500MHz のノードが 8 台に，Celeron 300MHz のノードが 2 台で構成されている．Water のコンパイルに使用したのは動的負荷分散機能を備えていないバージョンの Omni/SCASH である．グラフからわかるように性能の低いノードが全体性能を支配しており，性能ヘテロな環境では何らかの方法で負荷分散を行う必要があることがわかる．

加えて，性能ヘテロなクラスタにおけるノード間性能の差は環境ごとに異なるため，この負荷分散はランタイムシステムにより実性能に基づき，たとえば動的に実現されることが非常に重要である．

3. 動的負荷分散

1 節で述べたようにロードインバランスの原因は様々である：（1）対象アプリケーションが元々ロードインバランスを持つ場合,（2）マルチユーザー環境の影響でノード間の負荷に差がある場合,（3）アプリケーションが性能ヘテロなクラスタで実行される場合，などが考えられる．(2)（3）のケースではロードインバランスを事前に決定できないため，静的な負荷分散技法は不十分であり，実行時の性能情報に基づく動的負荷分散機能が必須である．

また，NUMA や SDSM 環境ではデータのローカリティーが性能に大きな影響を与えることは広く知られており，様々なデータ配置手法が提案されている:（1）システムのサポートするファーストタッチメモリアロケーション機能を利用するために，メインループの前に初期化ループを挿入する方法,（2）データとタスクの affinity 情報をソースコード上で付加する手法[5],（3）ディレクティブなどを用いてアプリケーションプログラマが明示的にデータの配置を指定する手法，などである．しかしこれらはいずれも静的な手法であり，実行時のデータアクセスパターンの変化や動的な負荷変化などと言ったものに対処できるものではない．

本研究では，実行時性能のセルフプロファイリングに基づく動的ループ再分割機能と SCASH レベルでの page fault counting に基づく動的ページマイグレーション機能を組み合わせる手法を提案する．不用意なループ再分割は不必要なページマイグレーションを生じるため，ループ再分割機能とページマイグレーション機能を適切に協調動作させることが重要となる．

3.1 ループ再分割のためのディレクティブ拡張

データ並列アプリケーションの実行時負荷分散機能としてループ再分割の手法を用いる．OpenMP の *dynamic* や *guided* などのスケジューリングオプションはある程度の負荷分散を達成できるが，双方ともスケジューリングのたびに集中管理されている chunk queue へアクセスしアップデートする必要があるため同期操作を伴い，特にクラスタのような分散メモリ環境では大きなオーバーヘッドとなる．このオーバーヘッドを避けるため，我々の提案する **profiled** スケジューリングではターゲットループの初期イテレーションの実行時性能を計測し，各プロセッサが計測した性能差に基づきプロセッサローカルに chunk サイズを調整することで負荷分散を達成する．

profiled スケジューリングは OpenMP の *schedule* 節で以下のように指定する．

```
schedule(profiled[, chunk_size[,
                eval_size[, eval_skip]]])
```

profiled スケジューリングが指定されると，まず eval_skip で指定した回数のイテレーションを各スレッドで通常実行し，続く eval_size で指定した回数のイテレーションの実行時性能計測を行う．その後，各スレッドは実行時性能計測の結果に基づき，残りのイテレーション空間を chunk_size を基底値として性能比にしたがってサイクリック分割する．

ただし，eval_skip, eval_size, chunk_size はこの順に省略可能であり，eval_skip を省略もしくは 0 とした場合には先頭のイテレーションから実行時性能計測

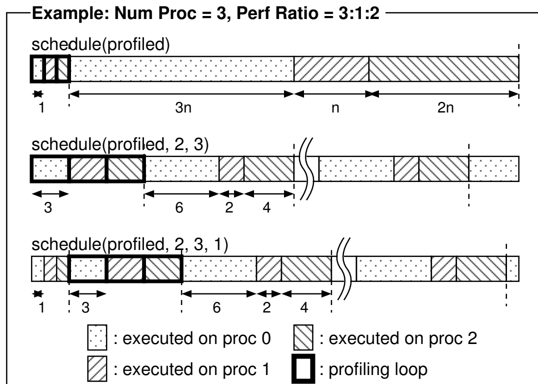
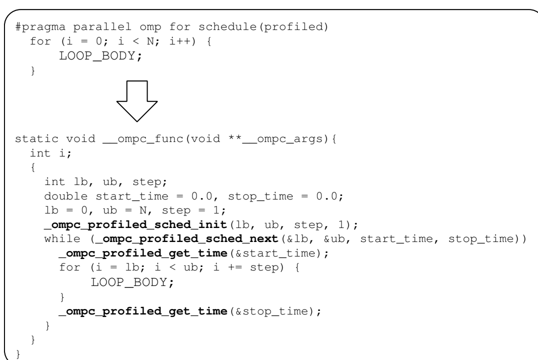


図 2　profiled スケジューリングの例



```
#pragma parallel omp for schedule(profiled)
  for (i = 0; i < N; i++) {
      LOOP_BODY;
  }

static void __ompc_func(void **__ompc_args){
  int i;
  {
    int lb, ub, step;
    double start_time = 0.0, stop_time = 0.0;
    lb = 0, ub = N, step = 1;
    _ompc_profiled_sched_init(lb, ub, step, 1);
    while (_ompc_profiled_sched_next(&lb, &ub, start_time, stop_time)) {
      _ompc_profiled_get_time(&start_time);
      for (i = lb; i < ub; i += step) {
          LOOP_BODY;
      }
      _ompc_profiled_get_time(&stop_time);
    }
  }
}
```

図 3　profiled スケジューリング時のコード変換

を行う．eval_size を省略した場合には eval_size = 1 として実行時性能計測を行う．chunk_size を省略もしくは 0 とした場合には残りのイテレーション空間を実行時性能比に基づいてブロック分割する．

図 2 に *profiled* スケジューリングの例を示す．ここではプロセッサ数を 3 としその性能比を 3 : 1 : 2 とする．

アプリケーションプログラマが並列ループに対して *profiled* スケジューリングを指定すると，Omni はそのパラレルリージョンをサブファンクションとしてくくりだし，パラレルリージョンの実行に参加しているスレッドによって並列実行される．それに加えて，Omni によりイテレーションの実行に要する時間を正確に計測するためループの前後に計測コードが挿入され，スレッドごとのターゲットループの実行性能の差に基づきループ再分割を行う（図 3）．実装では正確な性能計測のために，計測コードでは PAPI[2) を用いて，CPU がサポートするハードウェアリアルタイムカウンターを利用している．

各スレッドに割り当てられるサブループのインデックス，上限下限などの情報は各スレッドごとに管理され，`_ompc_profiled_sched_init()` で初期化される．各スレッドに割り当てられるサブループのイテレーション空間の計算は `_ompc_profiled_sched_next()` によって計算される．`_ompc_profiled_sched_next()` では `_ompc_profiled_get_time()` によって計測される実行時性能を元に，イテレーション空間をスレッド間で調整することでループ再分割を実現している．

以下に `_ompc_profiled_sched_next()` によるループ再分割アルゴリズムの概要を示す（図 4 参照）．

（1） profiled スケジューリング機能が有効か否か．
  （a） 前回のイテレーションにおける各スレッドの実行スピードを計算．そのデータを他のスレッドと交換．
  （b） 残っているイテレーションを，先に計測した実効性能を元に再分割した場合の予想実行時間を計算．
  （c） 現状の分割の割合で実行した場合の予想実行時間を計算．
  （d） ループ再分割により一定以上の性能向上が得られるか否か．
    （i） 各スレッドが担当するイテレーション数を計算．このデータは chunk_vector として各スレッドで保持．
    （ii） 各スレッドごとに自スレッドの担当するループインデックス，上限，下限などを調整し，終了．
  （e） 一定以上の性能向上が得られない場合，profiled スケジューリング機能を無効にし，計算済の chunk_vector を元に自スレッドの次の担当ループを計算し，終了．
（2） profiled スケジューリングが無効である場合，すでに計算済みの chunk_vector を元に自スレッドの次の担当ループを計算し，終了．

これらのうち他のスレッドとの通信をともなうのは (a) の**速度データ交換**の操作だけであり，スレッド間の性能差が，アプリケーション自体のロードインバランスによらず，ノード性能差や定常的な負荷によるものであった場合には，初回のみ通信をともない，それ以降は計算済みの割り当て比率によって，イテレーション割り当てが行われる．そのため，スケジューリングごとに中央管理された chunk queue にアクセスするために通信の発生する dynamic, guided などのスケジューリングに比べて，速度の向上が期待できる．

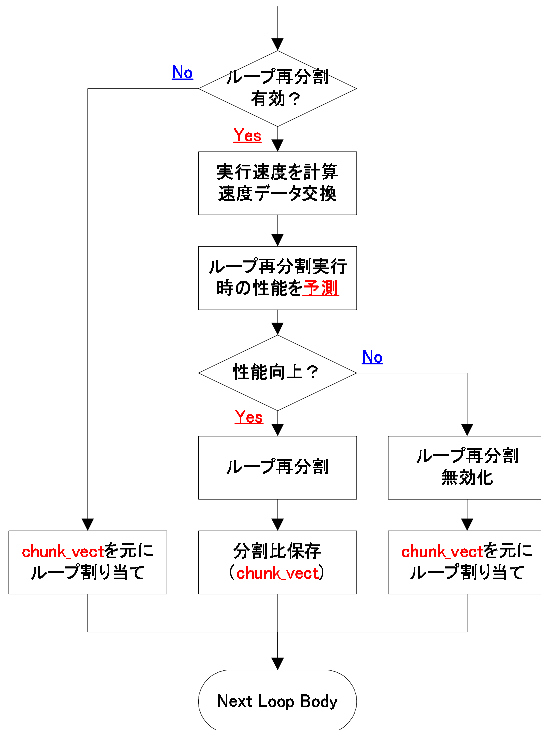


図 4 ループ再分割アルゴリズムの概要

4. 評　　価

profiled スケジューリングの評価に NAS Parallel Benchmarks (NPB-2.3) の EP, CG の OpenMP バージョンを利用した．オリジナルの Fortran バージョンから OpenMP への移植は RWCP によって行われたものである．

EP EP はモンテカルロ法などで使用される乱数の生成を行うカーネルベンチマークプログラムである．種となる乱数列に対して隣り合った要素のみに依存する計算が行われ，$2n$ 個の浮動小数点の疑似乱数が生成される．データ依存はごく少なくこの問題は Embarassingly Parallel と呼ばれている．こういった特徴から，profiled スケジューリングのデータのローカリティに左右されない純粋な性能を評価するため EP を用いた．

CG CG は大規模で正値対称な粗行列の最小固有値の近似解を Conjugate Gradient（CG）法を用いて求める問題である．行列積と不規則な通信を含む unstructured grid computation の典型でり，CG のカーネルループでは共有配列に対する頻繁なアクセスが生じる．そのためデータのローカリティが性能に大きな影響をおよぼす．その結果，profiled スケジューリングで動的負荷分散を行ったことによる性能向上が相殺される可能性がある．

表 1 評価環境: Performance Heterogeneous Cluster

	Fast nodes	Slow node
CPU	Pentium III 500MHz	Celeron 300MHz
Cache	512KB	128KB
Chipset	Intel 440BX	Same as “Fast”
Memory	SDRAM 512MB	Same as “Fast”
NIC	Myrinet M2M-PCI32C	Same as “Fast”

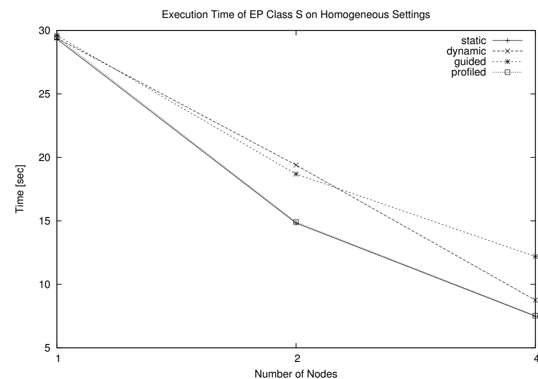


図 5 Execution Time of EP Class S on Homogeneous Settings (Pentium III nodes only)

4.1 評 価 環 境

評価環境を表 1 に示す．評価に使用した性能ヘテロなクラスタは CPU 性能のみヘテロな設定となっており Pentium III 500MHz のノードと Celeron 300MHz のノードから構成される．

OS には RedHat 7.2 を，クラスタシステムソフトウェアには SCore をコンパイラには gcc-2.96 -O をそれぞれ用いている．

評価結果では profiled スケジューリングに関して，もっとも良い性能の得られた chunk_size, eval_size, eval_skip いずれも指定しなかった場合の結果のみを示す．

4.2 性能ホモな設定における結果

まずはじめに，profiled スケジューリングそれ自身のオーバーヘッドを確認するために，データ配置が性能に影響を与えない EP を用いて性能的にホモな設定でベンチマークを行った．

図 5 から分かるように，dynamic および guided スケジューリングでは chunk 割り当てごとに中央管理されたループインデックスにアクセスするためのリモートアクセスがオーバーヘッドとして現れており，性能低下が見られるのに対し，profiled スケジューリングではオーバーヘッドがほとんどなく static スケジューリングと同等の性能が得られている．

なお，guided スケジューリングで 4 ノードを用いて実行した際に他に比べて性能が大きく低下しているのは，guided スケジューリングでは各ノードに異なるサイズの chunk を割り当てていくことになるため，結果としてロードバランスが大きく崩れたためだと考えている．

4.3 性能ヘテロな設定における結果

図 6 に static, dynamic, guided, profiled の各スケジューリングポリシーで chunk size を指定せずに EP class S を実行したときの性能を示す．

点線に黒四角でプロットされているグラフはノードとしてすべて “Fast” を利用する設定で static スケジューリングを用いた場合の EP の性能で，期待される性能の上限値を示している．実線にプラス記号でプロットされているグラフはヘテロな設定で static スケジューリングを用いた場合の EP の性能で，性能改善の基準値と見なすことができる．

EP のターゲットループのイテレーション空間は比較的小さく，その結果 chunk queue へのアクセス回数が少なくすむため，dynamic スケジューリングのオーバーヘッドは小さく static スケジューリングよりも高い性能を示している．

guided スケジューリングは 2 ノード実行時には性能ヘテロな環境での static スケジューリングよりも性能が向上しているが，4 ノード実行時には chunk の割り当てがノード間性能差にマッチせず，期待するロードバランシングは行えていない．

profiled スケジューリングでは，EP のターゲットループのループボディサイズが比較的大きく正確に実行時性能の計測が行えているため，ノード間性能比に沿って chunk_vector が計算されていることを確認した．その結果，性能ヘテロな環境での EP は profilied スケジューリングでもっとも良い性能が得られている．

次に，図 7 に CG class A を static, guided および profiled スケジューリングを用いて chunk size を指定しなかった場合の性能を示す☆．

CG kernel ベンチマークは共有配列へのアクセス数が多いため，リモートアクセスのコストが高いクラスタのような分散メモリ環境では，特にタスクとデータを同じノードに割り当てるようにすることが性能の観点から重要である．さらに，たとえばデータ初期化フェーズと計算フェーズのように，“parallel for” リージョンごとにループ再分割が行われるため，タスクとデータの affinity が崩れる可能性がある．その結果，CG の場合には static スケジューリングに対して profiled スケジューリングの性能が低くなっている．この profiled スケジューリングがデータのローカリティに対しておよぼす影響を次の節で調査する．

☆ dynamic スケジューリングに関してはオーバーヘッドのため実行時間が非常に大きくなってしまうため割愛

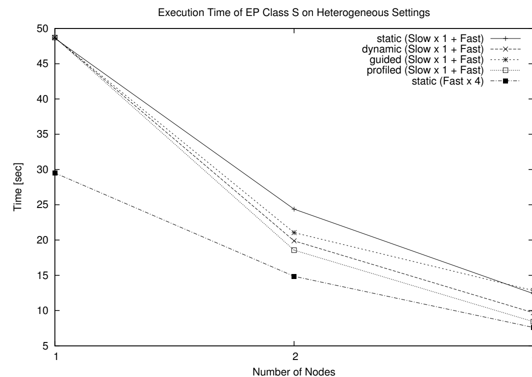


図 6 Execution Time of EP Class S on Heterogeneous Settings (one Celeron node + Pentium III nodes)

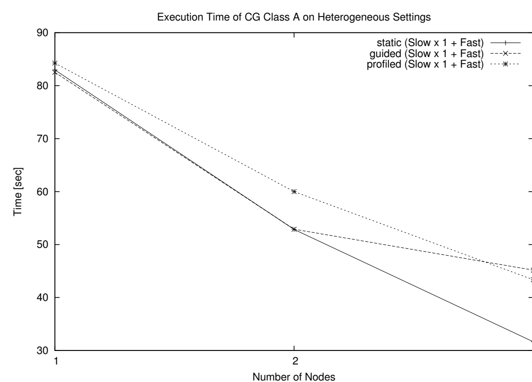


図 7 Execution Time of CG Class A on Heterogeneous Settings (one Celeron node + Pentium III nodes)

また，guided スケジューリングに関しては後述する profiled スケジューリングで見られるスケジューリング回数の増加はなく，スケジューリングのオーバーヘッドは比較的小さく押さえられているため，2 ノード実行時には profiled スケジューリングより高い性能が得られているが，やはりデータのローカリティの問題から static スケジューリングと同程度の性能にとどまっている．4 ノード実行時には EP の時と同様に chunk の割り当てがノード間性能差にマッチせず，期待するロードバランシングは行えていない．guided スケジューリングは 1 台だけ性能が異なるような状況にはあまり適していないと予想される．

表 2 Memory Behavior of the CG Class A on Celeron×1 + Pentium III×3 settings for static/profiled scheduling

	static	profiled
L2 miss ratio	29.6%	31.1%
Page Fault at SCASH	16456	27201
Barrier	5088	8006

4.4 profiled スケジューリングがデータローカリティに与える影響

表 2 に CG Class A を “Slow” ノードを 1 ノードに “Fast” ノードを 3 ノードの計 4 ノードで static および profiled スケジューリングを用いて実行した場合の，L2 cache miss ratio，バリア同期，SCASH レベルの page fault の回数を示す．なお，L2 cache miss ratio は PAPI を用いて以下のように計算しいている:

$$\text{L2 cache miss ratio} = \frac{\text{PAPI_L2_TCM}}{\text{PAPI_L1_DCM}} \times 100$$

L2 cache miss ratio に関しては static, profiled ともに同程度の値であるが，SCASH レベルの page fault に関しては profiled スケジューリングの方が static スケジューリングより多数生じている．これは profiled スケジューリングによってループ再分割が行われるたびにタスクの割り当てが変更され，タスクとデータの affinity が崩れるためである．その結果，より多くのリモートメモリアクセスを伴い，性能低下につながっている．

profiled スケジューリングにおけるバリア同期回数の増加は以下のような理由による: まず，profiled スケジューリングではループ再分割を決定するために実行時性能のデータをすべてのノードで交換している．また，現状ではリモートメモリ参照の際に page migration が行われていないため，ループ再分割によってデータのローカリティが変化した場合，性能の予測値がループ再分割後の実際の性能と一致しない可能性が高い．この不一致がループ再分割の不必要な繰り返しにつながり，結果として余計なバリア同期増加により性能の低下の原因となっている．

この結果，ループ再分割後にデータのローカリティを回復するために page migration を組み合わせることが必須であると言え，現在その作業を行っている．また，動的ループ再分割はデータのローカリティに，page migration は実行時性能にそれぞれ影響を与えるため，両手法を協調動作させる際に，現在行っているように実行時性能比に沿ったループ再分割を即座に行うのではなく，漸次的に適用することを考えている．これによって不必要なループ再分割の繰り返しを避け，すばやく stable なタスクおよびデータ配置に安定することが期待できる．

5. Ongoing Work: Page Fault Conting に基づく Page Migration

3 節で述べたように，SDSM のような分散メモリ環境で高性能を達成するにはデータのローカリティを考慮することが特に重要である．しかしながら，本来共有メモリ環境を想定している OpenMP にはデータの配置をプログラマーが指定する手段はない．NUMA や SDSM 環境などを対象としてデータの配置をプログラマが明示するためのディレクティブ[1),6)]，ディレクティブによって affinity を指定することでスレッドとそれがアクセスするデータをそろえる手法[5)]，ハードウェアが提供する page reference counter を元に頻繁に参照するノードにデータをページ単位でマイグレーションする手法[7)] などが提案されているが，本研究では動的ループ再分割も同時に行うため，静的なデータ配置手法は行えない．

また，各ページに対するノードごとの参照回数をすべてカウントすることはハードウェアサポートのない SDSM 環境ではオーバーヘッドが非常に大きくなってしまう．そこで，本研究では SDSM レベルの page fault つまり，メモリ参照がローカルで解決できなかった回数を数え，あるページに対するリモート参照のもっとも多いノードにそのページをマイグレーションする．直接的にすべてのページ参照をカウントする手法に対し，この方法ではローカル参照がカウントに入らないため，ページマイグレーションによってかえってリモート参照が増大してしまうという可能性を避けられない．しかし，我々は SPMD スタイルのアプリケーションをターゲットとしており，並列ループにおけるメモリアクセスパターンがイテレーション間で変化しないという前提においては，数回のページマイグレーションで最適なページ配置に収束することが期待できる．

6. 関 連 研 究

Nikolopoulos らは SGI Origin 2000 上で Origin の提供する hardware page reference counter を用いて OpenMP プログラムの parallel loop における完全な page reference count に基づくユーザレベルの dynamic page migration による data placement を実現している[7)]．これによって適切な区間における適切な page reference counter の値に基づき，適切なタイミングで page migration を行うことが可能となっており，OS の提供する dynamic page migration 機

能よりも高い性能を NPB のいくつかのプログラムで得られている．我々の提案は彼らの手法を，ハードウェアサポートのないコモディティクラスタ環境に拡張するものである．

原田らは SCASH に，各プロセッサにおけるページ変更量に基づくホームマイグレーション機能を追加している[3]．バリア同期ポイントごとに各ページの変更量の多いノードを特定し，ページの home をそのノードに変更することによって，リモートアクセスによるオーバーヘッドを削減している．SPLASH2[11] の LU を用いた評価の結果，8 ノードまででは home を最適に割り当てたケースよりもこの方法によって高い性能を達成している．

7. Conclusion and Future Work

本論文では我々が Software Distributed Shared Memory, SCASH 上の OpenMP の実装である Omni/SCASH に対して現在行っている，動的負荷分散拡張に関して報告した．我々は目標は，ノード間性能のヘテロ性やマルチユーザ環境などに起因するアプリケーションのロードインバランスを半自動的に解決する手法の実現である．このような問題に対して静的なアプローチは不十分であり，我々はターゲットループの実行時性能セルフプロファイリングに基づくループ再分割と，ターゲットループにおける page fault conting に基づくページマイグレーションを用いた動的負荷分散技法を提案した．これらの手法を用いることにより，ユーザープログラマが明示的にデータとタスクの配置を記述することなく，ラインタイムシステムによって最適なロードバランシングが行える．

予備評価の結果，データローカリティが性能に影響を与えない EP ベンチマークでは，実行時性能に基づきループ再分割を行う profiled スケジューリングは，static スケジューリングと比較してもオーバーヘッドなく，性能ヘテロな環境で十分な負荷分散が行え，もっとも良い性能を示した．しかしながら，ループ再分割によってデータのローカリティが損なわれるため，データのローカリティが性能に強く影響する CG ベンチマークでは SDSM のリモートメモリ参照の増加，およびループ再分割を繰り返してしまうことによるバリア同期の増加により，性能低下が見られた．これは，データのローカリティ回復のため profiled スケジューリングを動的ページマイグレーション機能と協調動作させる必要があることを示しており，ハードウェアサポートなしに SDSM レベルの page fault counting を行うことで一定の精度で軽量なページマイグレーションを実現する方法を提案した．現在は，提案するページマイグレーション機能を Omni/SCASH に実装中であるが，早急に profiled スケジューリングとページマイグレーションの協調動作の効果を確認する評価を行う必要がある．

参 考 文 献

1) J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines: The Language. In *Proceedings of Workshop on OpenMP Applications and Tool (WOMPAT'2000)*, July 2000. San Diego, USA.
2) S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
3) H.Harada, Y.Ishikawa, A.Hori, H.Tezuka, S.Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory system. In *Proceedings of IEEE 4th HPC ASIA 2000*, pages 158–163, may 2000.
4) H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, and Y. Ishikawa. SCASH: Software DSM using High Performance Network on Commodity Hardware and Software. In *Proceedings of Eighth Workshop on Scalable Shared-memory Multiprocessors*, pages 26–27. ACM, May 1999.
5) A.Hasegawa, M.Sato, Y.Ishikawa, and H.Harada. Optimization and Performance Evaluation of NPB on Omni OpenMP Compiler for SCASH, Software Distributed Memory System (in Japanese). In *IPSJ SIG Notes*, 2001-ARC-142, 2001-HPC-85, pages 181–186, Mar. 2001.
6) J. Merlin. Distributed OpenMP: Extensions to OpenMP for SMP Clusters. In *Invited Talk. Second European Workshop on OpenMP (EWOMP'00)*, Oct. 2000. Edinburgh, Scotland.
7) D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP? In *Proc. of Supercomputing 2000*, Nov. 2000. Dallas, TX.
8) M. Sato, H. Harada, and Y. Ishikawa. OpenMP compiler for Software Distributed Shared Memory System SCASH. In *Proceedings of Workshop on OpenMP Applications and Tool (WOMPAT'2000)*, July 2000. San Diego, USA.
9) H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: An Operating System Coordinated High Performance Communication Library. In P. Sloot and B. Hertzberger, editors, *High-Performance Computing and Networking '97*, volume 1225, pages 708–717. Lecture Notes in Computer Science, Apr. 1997.
10) http://www.top500.org/.
11) S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.