不均質なクラスタ環境を対象とする データ再配置による動的負荷分散機構の設計と実装

野口繁一† 吉瀬謙二† 片桐孝洋† 弓場敏嗣†

性能が不均質な計算機によって構成されるクラスタや複数のユーザによって共有されるクラスタ（これらを不均質なクラスタ環境と呼ぶ）において並列アプリケーションプログラムを実行する場合、実行中に計算機の間で負荷を動的に調整することで、そのクラスタの性能を最大限に引き出すことができる。本論文では、不均質なクラスタ環境を対象とするデータ再配置による動的負荷分散機構の設計と実装について述べる。動的負荷分散では、負荷の調整に伴う通信の時間がボトルネックになり、性能を低下させてしまう場合がある。その通信にかかる時間を予測して、性能が低下するようであれば、その通信を行わないという機構を導入することでこの問題を解決する。

Design and Implementation of the Dynamic Load Balancing Mechanism Based on Data Replacement for Heterogeneous Cluster Environment

Shigekazu Noguchi,† Kenji Kise,† Takahiro Katagiri† and Toshitsugu Yuba†

There are clusters which are composed of various kind of computer and which are shared by many users. When a parallel application program is executed on a cluster, a dynamic load balancing among computers on executing causes the cluster's maximum performance. In this paper, we designed and implemented the dynamic load balancing mechanism based on data replacement for heterogeneous cluster environment. There is the problem that communication time of dynamic load balancing might decrease the performance of a cluster. We solved this problem by adding a mechanism that predicts the communication time. This mechanism avoids the communication that decreases performance.

1 はじめに

近年高性能コンピューティング環境として、計算機（ノード）を高速ネットワークで接続した PC クラスタが普及してきた。クラスタの構成としては、各ノードの性能が同一である均質クラスタ(Homogeneous Cluster)が一般的である。しかし、計算機資源の再有効利用の観点から見ると、均質クラスタに性能の異なる計算機を追加した不均質クラスタ(Heterogeneous Cluster)が構築されることもある。

クラスタを用いて並列処理を行う場合、計算負荷は各ノードに均等に割り付けるのが最も単純で一般的である。この方法は均質クラスタにおいては最適な割付けである。しかし、不均質クラスタでは、各ノードの性能に適した負荷を割り付けなければ、クラスタ性能にみあった処理速度を得ることができない。また、クラスタは一般的に複数のユーザが同時に利用するものである。他のユーザに利用されると、均質クラスタの場合でも、ノード間に性能の不均質が生じる。このときも、均等な負荷割付では性能に見合った処理速度を得ることはできない。

そこで、並列アプリケーションの実行中に各ノードの負荷を検出し、それに応じてノード間で負荷の割合を動的に変動させていく、効率的な動的負荷分散の実現が課題とされている。この課題に関しては、現在までに多くの研究が行われている [2〜6]。

動的負荷分散にも問題においては、負荷分散に要する時間の問題がある。負荷分散の時間が大きくなると、たとえアプリケーションの処理が速くなったとしても、負荷分散の時間が原因で、全体の実行時間が大きくなってしまうことがある。

本研究では、不均質なクラスタ環境を対象とするデータ再配置による動的負荷分散機構を提案・実装する。実装した動的負荷分散は、負荷分散の時間と分散後の残りのアプリケーションの処理時間を予測する機能を持つ。この予測機能によって、先の負荷分散の時間に関する問題点を解決する。

2 動的負荷分散機構の提案と設計

2.1 提案

動的負荷分散の実現については、さまざまなアプローチがある。その 1 つとしてアプリケーションに動的負荷分散機構を直接記述するものがある。これは、ユーザが動的負荷分散に精通していないと容易に記述することはできず、負担は大きい。他の方法として、ライブラリレベルでの実装がある[6]。これは動的負荷分散機構のライ

† 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro-Communications

表 1：API 一覧

機能	関数名と引数
1.動的負荷分散機構の初期化	LoRP_Init(int argc, char **argv, int per, int inter);
2.負荷分散のため分散するデータの設定	LoRP_Dist(void *var, char *tag, MPI_Datatype dtype int dim, int *ary, int dir, int blk, int *sleeves);
3.同期変数の設定	LoRP_Sync(int *var, char *tag, int type);
4.負荷分散の実行ポイント	LoRP_Sched(int loop, int total);

ブラリを作成し、それをユーザが並列アプリケーションに記述して利用する方法である。これによって、ユーザがアプリケーションごとに動的負荷分散機構を記述する必要がなくなり、ユーザの負担を軽くすることができる。本研究では実用性などを考慮して、このライブラリレベルでの実装を選択する。

本研究では、クラスタの各ノードに割り付けるデータをノード間で再配置させることで、動的負荷分散を実現する。具体的には、計算対象のデータ配列をブロック分割し、各プロセスに割り付け、その分割したデータをプロセス間で再配置させることで負荷分散を行う。従って、本機構が適用できるアプリケーションは、計算対象のデータをブロック分割することで並列処理を行うアプリケーションに限定される。

負荷分散の時間の予測には以下の方法を用いる。アプリケーションのメインルーチンを実行する前に、2 つのノード間で異なるサイズのデータを 3 回送受信する。それぞれの送受信の所要時間から、データサイズと送受信の時間の 1 次関数を作成する。この 2 つの値を利用して、再配置させるデータサイズに応じた負荷分散の時間を予測する。この関数は 1 回の負荷分散に対して、1 回の全対全通信を行う動的負荷分散の関数を作成するのが理想的である。しかし今回は実現の容易さを考慮し、1 回の負荷分散に対して、1 回の 1 対 1 通信を行う動的負荷分散の関数を作成する。本機構は動的負荷分散 1 回につき、1 回の 1 対 1 通信を行う方式をとる。分散後の残りのアプリケーションの処理時間の予測は以下の方法を用いる。負荷分散前の単位データサイズあたりの処理時間から、分散後のデータサイズの処理時間を求める。それに残りの処理回数をかけたものを、残りの処理時間として扱う。これらの機能を利用すると、負荷分散を行った場合とそうでない場合の残りの実行時間を予測できる。この 2 つの時間を比較して負荷分散を行うか否かを決定することにより、性能低下につながる無駄な負荷分散を避けることができる。

本研究では、上記の手法を用いて、動的負荷分散機構を実現する API (Application Programming Interface)ライブラリである LoRP(Load Balancer based on Data Re-Placement)を提案する。

2.2 対象アプリケーション

計算対象となるデータをブロック分割し、その分割データを各プロセスが処理するアプリケーションに、動的負荷分散機構 LoRP を適用できる。

さらに、LoRP は重複領域を必要とするアプリケーションにも対応している。例えばアプリケーションのコア計算部分が

```
a[i] = a[i-1]*a[i]* a[i+1];
```

の場合、単純に配列 a をブロック分割するだけでは、a[i-1]や a[i+1]といった値を参照できない。この場合、隣接するプロセスとの間でいくつかのデータを重複してもつ必要がある。このような領域を重複領域という。LoRP はこのような重複領域を必要とするアプリケーションにも対応する。

2.3 設計

表 1 に LoRP の API の一覧を示す。本 API は、文献[4]で作成された API を参考に作成した。この API には通信ライブラリ MPI(Message Passing Interface)の定数を使用している。従って、この API を使用する前に、必ず MPI の初期化を行わなければならない。

2.3.1 LoRP_Init

LoRP_Init は動的負荷分散機構 LoRP の初期化を行う。この関数を設定する前には、本機構の他の関数は設定できない。

LoRP_Init は 4 つの引数を設定する必要がある。1 つ目と 2 つ目の引数である argc, argv は、本機構内で MPI ライブラリを使用するために必要となる。3 つ目の引数は、負荷分散を行う条件となる閾値を指定する。4 つ目の引数は、負荷分散の実行ポイント LoRP_Sched を実行する間隔を示す。

2.3.2 LoRP_Dist

LoRP_Dist は、負荷分散を行うデータを設定するために必要な関数である。この関数は負荷分散を行いたいデータの数に応じて、複数設定できる。これを用いて設定したデータは、負荷分散の実行ポイント LoRP_Sched において、プロセス間でその計算サイズが調整される。

LoRP_Dist は 8 つの引数を設定する必要がある。1 つ目の引数は、負荷分散を行うデータに関連付けるポインタ変数を指定する。2 つ目の引数は、負荷分散を行うデータを識別するためのタグを指定する。3 つ目の引数は、負荷分散を行うデータの型を指定する。4 つ目の引数は、負荷分散を行うデータの次元数を指定する。これは 4 次元まで対応している。5 つ目の引数は、負荷分散を行うデータの各次元のサイズを指定する。6 つ目の引数は、負荷分散を行うデータの分割する次元を指定する。7 つ目の引数は、負荷分散を行うデータの分割する次元の初期サイズを指定する。8 つ目の引数は、負荷分散を行うデータに必要な重複領域のサイズを指定する。

2.3.3 LoRP_Sync

LoRP_Sync は、その負荷分散に合わせて変更すべき変数（同期変数）を設定する関数である。ここで設定した変数の値は、計算サイズの調整に合わせて自然に変更される。この関数は同期変数に設定したい数に応じて、複数設定できる。

LoRP_Sync は 3 つの引数を設定する必要がある。1 つ目の引数は、同期変数に設定したい変数を指定する。2 つ目の引数は、LoRP_Dist で指定した、負荷分散を行うデータの識別タグを指定する。そのタグを持つデータの負荷分散に同期して、同期変数の値が変更される。3 つ目の引数は同期変数が担っている役割を指定する。本機構では BLKSIZE(データのサイズ), START(データのスタート値), END(データのエンド値)の 3 種類を指定できる。

2.3.4 LoRP_Sched

本機構は、負荷分散を行うポイントを、ユーザがアプリケーションに明示的に記述する必要がある。そのポイントを指定する関数が、LoRP_Sched である。LoRP_Sched は負荷情報から、負荷の不均衡が確認されたときのみ実行される。負荷情報とは LoRP_Sched を呼び出す間の時間である。この関数を実行すると、まず動的負荷分散の可否を決定する。そこで可と出た場合のみ、負荷分散の対象のデータが負荷分散され、同期変数の値も変更される。この関数は複数設定できる。複数設定した場合、設定した区間の実行時間が負荷情報となる。

LoRP_Sched は定期的に呼び出す必要があるため、メインループに組み込むのが一般的である。その場合、分散対象のデータが安定している箇所に組み込むことをユーザが保証しなければならない。LoRP_Sched をコア計算の途中などに組み込んでしまうと、一部の負荷分散を行うデータが更新されないままデータを移動してしまい、データの整合性がとれなくなってしまうからである。

LoRP_Sched は 2 つの引数を設定する必要がある。1 つ目の引数はコア計算部分の現在の計算回数を指定する。2 つ目の引数はコア計算部分の総実行回数を指定する。

3 動的負荷分散の処理フローと組み込み例

3.1 処理フロー

LoRP を組み込んだアプリケーションの処理フローを図 1 に示す。色のついている処理が、LoRP によって行われる処理である。

アプリケーションを実行すると、まず動的負荷分散の初期化、分散するデータなどの各種設定、負荷分散の時間の予測データの取得を行う。その後に分散されたデータの初期化を行う。メインルーチン内にはスケジューリングポイントを設ける。そのポイントを通過したときに負荷の不均衡が検出されたら、まず動的負荷分散を行うか否かを決定する。まず負荷分散の時間の予測データを使って負荷分散に要する時間を予測する。次に分散前のデータサイズでの 1 ループの処理時間から、分散後のデータサイズでの 1 ループの処理時間を予測する。そして、分散しない場合の残りのループ回数の処理時間と、予測

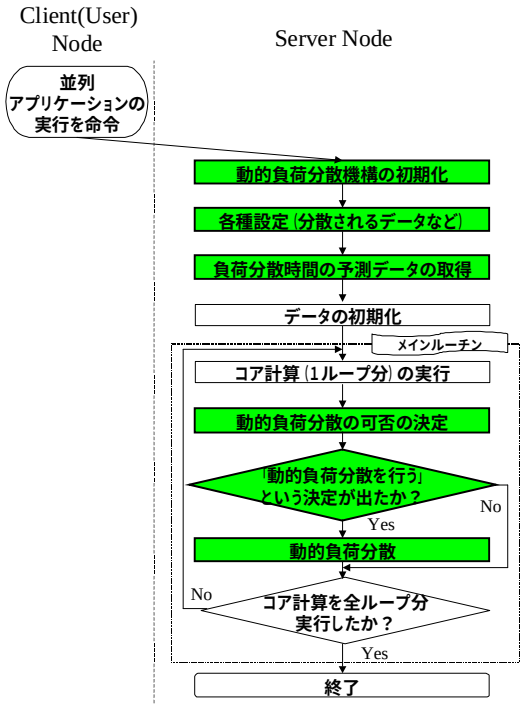

図 1：処理のフローチャート

した負荷分散の時間と分散した場合の残りのループ回数の処理時間を足し合わせた時間を比較する。ここで、後者の予測した時間のほうが短い場合、「動的負荷分散を行う」という決定を出す。「負荷分散を行う」という決定が出たら、データを再配置し、負荷分散を行う。これをメインルーチンが終了するまで繰り返すことで、そのときの負荷状況に応じた適切な負荷分散が可能となる。

3.2 組み込み例

具体的な例として、並列アプリケーションである姫野ベンチマーク[1]に LoRP を組み込んだものを、オリジナルのプログラムと共に図 2 に示す。姫野ベンチマークとは、非圧縮流体解析コードの性能評価に使われるベンチマークである。コア計算部分では、ポアッソン方程式を、ヤコビ反復法を用いて解いている。

図 2(b)の点下線部分が LoRP を組み込んだ部分である。4 行目で、LoRP ライブラリのあるヘッダファイル LoRP.h をインクルードしている。9 行目で、負荷分散を行うデータを設定するために必要なポインタ変数****pp を宣言している。18～20 行目で、LoRP_Dist で用いる変数を設定している。23 行目で本機構の初期化関数 LoRP_Init を実行している。24 行目で LoRP_Dist を使い、int 型 3 次元配列*pp[MX0][MY0][MZ0]を確保している。30 行目で、確保した配列*pp に、負荷分散を行いたいデータの p アドレスを代入している。この代入によって、配列 p[MX0][MY0][MZ0]は、負荷分散の行うデータとして設定される。33 行目で、LoRP_Sync を使い、変数 imax を同期変数に設定している。imax は、識別タグ”p”のデータと同期して値が変更される、計算サイズ型の同期変数となる。メインルーチン内の 46, 54 行目で、負荷分散の実行関数 LoRP_Sched を組み込んでいる。図 2 のように、

```
01  #include <stdio.h>
02  #include "mpi.h"
03  #include "param.h"
04
05  float jacobi(int);
06  ...
07  static float  p[MIMAX][MJMAX][MKMAX], ...;
08  static int npe,id;
09  static int imax,jmax,kmax;
10  ...
11
12  int
13  main(int argc,char *argv[])
14  {
15    int    i,j,k,nn;
16    float  gosa;
17
18    ...  /*Initializing (p,…,imax,…,nn)*/
19    MPI_Barrier(MPI_COMM_WORLD);
20    gosa = jacobi(nn);      /* main loop */
21    ...
22  }
23
24
25  float jacobi(int nn) /*core calculation*/
26  {
27    int i,j,k,n;
28    float s0, ...;
29
30    for(n=0 ; n<nn ; ++n){
31      ...
32      for(i=1 ; i<imax-1 ; ++i)
33        for(j=1 ; j<jmax-1 ; ++j)
34          for(k=1 ; k<kmax-1 ; ++k){
35            s0 = p[i+1][j  ][k  ] * ...
36          }
37      ...
38    }
39    return(...);
40  }
41
```

(a) オリジナル

```
01  #include <stdio.h>
02  #include "mpi.h"
03  #include "param.h"
04  #include "LoRP.h"
05
06  float jacobi(int);
07  ...
08  static float  p[MX0][MY0][MZ0], ...;
09  static float  ****pp, ...;
10  static int npe,id;
11  static int imax,jmax,kmax;
12  ...
13
14  int main(int argc,char *argv[])
15  {
16    int    i,j,k,nn;
17    float  gosa;
18    int    ary_p[3]={MX0, MY0, MZ0};
19    ...
20    int    blksize, sleeves[2] = {1, 1};
21
22    ...
23    LoRP_Init(argc, argv, 10, 10);
24    LoRP_Dist(&pp,"p",MPI_FLOAT,3,ary_p,0,
25                                imax,sleeves);
26    ...
27    for(i=0; i<MX0; i++)
28      for(j=0; j<MY0; j++)
29        for(k=0; k<MZ0; k++){
30          pp  [i][j][k] = &p  [i][j][k];
31          ...
32        }
33    LoRP_Sync(&imax, "p", BLKSIZE);
34
35    ...  /*Initializing (p,…,imax,…,nn)*/
36    gosa = jacobi(nn);      /* main loop */
37    ...
38  }
39
40  float jacobi(int nn) /*core calculation*/
41  {
42    int i,j,k,n;
43    float s0, ...;
44
45    for(n=0 ; n<nn ; ++n){
46      LoRP_Sched(n, nn);
47      ...
48      for(i=1 ; i<imax-1 ; ++i)
49        for(j=1 ; j<jmax-1 ; ++j)
50          for(k=1 ; k<kmax-1 ; ++k){
51            s0 = p[i+1][j  ][k  ] * ...
52          }
53      ...
54      LoRP_Sched(n, nn);
55      ...
56    }
57    return(...);
58  }
```

(b) LoRP 組み込み後

図 2：サンプルプログラム（姫野ベンチマーク）

この関数を 2 つ使用すると、2 つの関数の区間（47～53 行目）の実行時間が、負荷情報として取得される。このポイントで、負荷の不均衡が確認され、負荷分散をした方がアプリケーションが早く終了すると予測したときのみ、プロセス間で負荷の調整、すなわち、データの再配置を行う。

4　評価

4.1　評価環境

動的負荷分散機構 LoRP の評価を行う。評価用のアプリケーションとして、姫野ベンチマークを用いる。計算サイズは M(128×128×256)を用いる。

評価に用いる計算機には、性能が不均質なクラスタを用いる。ノード数は 3 つである。各ノードの性能を表 2 に示す。ノード間は Gigabit Ethernet で接続されている。

評価方法を説明する。本評価では負荷分散時間の予測機構の有効性を検証する。各クラスタで、負荷分散時間の予測機構がある LoRP を組み込んだ姫野ベンチマークと、予測機構のない LoRP を組み込んだ姫野ベンチマー

表 2：不均質クラスタのスペック

	node01, node02	opt01
CPU	Pentium4 2.0GHz	Opteron 1.8GHz*2
Memory	1GB	2GB

クを実行する。その実行途中、特定のノードに、レジスタ上で加算を繰り返すだけのプログラムを実行する。こうすることで、人為的な負荷を想定した不均質クラスタ環境をつくる。加算プログラムの加算回数は、実行している姫野ベンチマークが終了する直前付近で加算が終了するよう設定した。これは、予測機構が正しく動くかを確認するためである。上記のように加算回数を設定し、加算プログラムが終了すると、プログラムが終了したノードは負荷が減り、ノード間に負荷の不均衡が生じる。その場合、予測機構のない LoRP を組み込んだ姫野ベンチマークは、無条件で負荷を調整する。しかし、予測機構のある LoRP を組み込んだ姫野ベンチマークは、負荷分散の前に、負荷分散を行った場合と行わなかった場合の残りの実行時間を比較し、より速く姫野ベンチマークが終了できる方を選択する。姫野ベンチマークの残りの処理時間が短いのなら、負荷分散をしない方が、速く終了すると考えられる。よって、予測機構のある LoRP は、負荷分散を行わない。

姫野ベンチマークのコア計算のループ回数を 200 回とした。実行開始の 30 秒後（30 ループ付近）に node02 で加算プログラムを実行し、その 10 秒後（40 ループ付近）にさらに node02 で加算プログラムを実行した。

4.2 評価結果

予測機構のない LoRP を用いた場合の実行結果を図 3、予測機構のある LoRP を用いた場合の実行結果を図 4 に載せる。

2 つの図とも 10, 20 ループ目でマシン性能の不均質を動的負荷分散によって修正しているのが分かる。次に 30 ループ目で node02 に加算プログラムによる負荷が加わる。分かりにくいが、40 ループ付近にも node02 にもう一つの加算プログラムが実行されている。この負荷の不均質も 40, 45, 55 ループ目の負荷分散によって修正されている。ここまでの処理では 2 つの図の間に変化はないが、170 ループ以降では動きが異なる。図 3 では加算プログラムの 1 つが 170 ループ付近で終了し、node02 の処理速度が上がっている。このノード間の不均衡によって、180, 190 ループ目で負荷分散を行っているのがわかる。185 ループ付近で node02 の処理速度が上がっているのは、もう一つの加算プログラムが終了したためである。図 4 では 170 ループ付近で 1 つの加算プログラムが終了しているが、それ以降で負荷分散をせずにアプリケーションを終了している。これは、予測機構によるものである。予測機構ありの LoRP は負荷分散の時間と調整後の残り 20 ループの処理時間を予測する。そして、その予測から負荷分散しない場合とした場合の残りの実行時間を予測し、比較する。その結果、負荷分散をすると処理速度が低下すると判断し、それ以降では負荷分散を行わなかった。

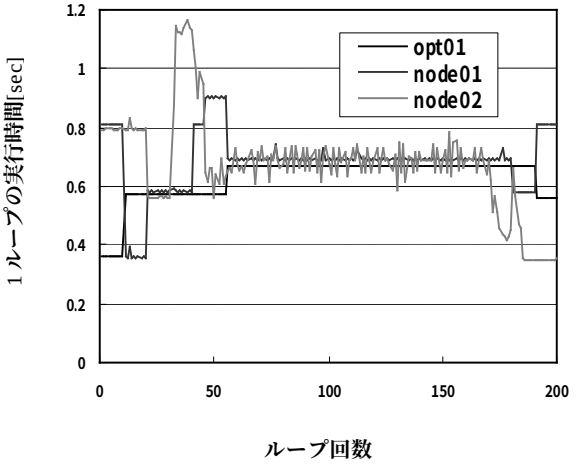

図 3：LoRP あり・予測なしの姫野ベンチマーク

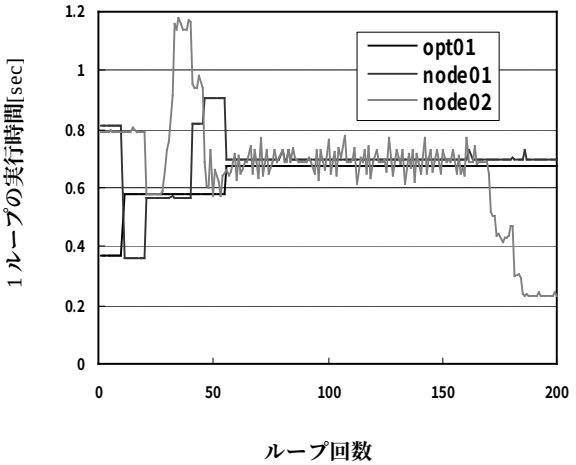

図 4：LoRP あり・予測ありの姫野ベンチマーク

表 3：不均質クラスタでの評価結果

	LoRP なし	LoRP あり	
		予測なし	予測あり
負荷分散時間[sec]		10.0	6.9
実行時間[sec]	247.6	170.4	165.6

た。

全体の負荷分散時間と実行時間は表 3 のようになった。まず、予測機構のない LoRP を組み込んだベンチマークの負荷調整にかかった時間が 10 秒であるのに対し、予測機構のある LoRP を組み込んだベンチマークでは約 7 秒となり、予測機構のないものに比べて約 3 秒短くなっていた。この結果から、予測機構のある LoRP は必要のない通信を行っていないことが分かった。次に、予測機構のない LoRP を組み込んだベンチマークの結果が約 170 秒であるのに対し、予測機構がある LoRP を組み込んだベンチマークでは約 165 秒となり、予測機構のないものに比べて約 5 秒速くなっていた。この結果から、負荷分散の時間の予測機構によって、約 3%速度が向上したこと

が確認できた。

5 関連研究

負荷分散に関しては、静的・動的を含め、多くの研究がなされている。その中から、特に本研究と関連の深い 3 つの論文を紹介し、本研究との差異を議論する。

文献[4]では、自律コンピューティングをターゲットとしている。LAM/MPI に実装されている動的プロセス生成機能を利用して、アプリケーションの実行中にプロセスの生成・削除を行う。そのプロセスの生成・削除後の負荷割り当てとして動的負荷分散を利用している。この研究はあくまで自律コンピューティングの実現が主たる目的のため、処理時間の短縮を考慮していない。

文献[5]では、HPF(High Performance Fortran)を拡張し、コンパイラレベルでの動的負荷分散を実現している。この研究は本研究と同じく、ブロック分割したデータの割り当てサイズを変動させることで負荷分散を行っている。しかし、評価に用いたベンチマークが grid, tomcatv, shallow など単純なデータ並列を用いたアプリケーションであり、姫野ベンチマークのような分割データに重複領域が必要なアプリケーションは対応していない。

文献[6]では、アプリケーションの変更を最小限にとどめ、かつスケジューリングプログラムの再利用性を高めることを研究の軸としている。この研究では、関数にある変数を渡すことによって実行される計算の単位をタスクとし、そのタスクの数をプロセス間で調節することで負荷分散を実現している。これは、負荷の大きいプロセスのタスクを、負荷の小さいプロセスに転送するというアルゴリズムをとっている。しかし、この方式で並列処理を行うアプリケーションは、ブロック分割の並列処理のそれより一般性がなく、汎用性が低い。

6 おわりに

本研究では、負荷分散を行うか否かを決定する機能を追加した、データ再配置による動的負荷分散機構を提案し、これを実現した。本機構を不均質なクラスタ環境、他のユーザが一部のノードを使用しているクラスタ環境において実装・評価を行った結果、適切な負荷分散の実行の可否決定が行われていることが確認でき、その有効性を示すことができた。

また、本機構の実装によってさまざまな適用限界が明らかとなった。そのいくつかをあげる。

まず、メモリ利用による限界があげられる。本機構で負荷分散の対象となるデータを定義するには、その対象のデータの 3 倍のメモリサイズが必要となる。そのため、この機構をある計算機で使用すると、その中の最小メモリのノードに性能が縛られてしまう。

次にアプリケーションの限界があげられる。本機構は計算するデータをブロック分割し、それを各ノードに割り付けることで並列処理を行うアプリケーションにのみ対応している。つまり、それ以外の方法で並列処理を行っているアプリケーションには対応しておらず、適用できるアプリケーションが限られる。

次に、ハードウェア環境の限界があげられる。本論文で用いた不均質クラスタは不均質の程度がそれほど大きいものでなく、ノード数も 3 つと少なかったため、性能向上を見ることができた。しかし、不均質の程度が大きいクラスタや、ノード数の多いクラスタを用いた場合、必ずしも性能が向上するとはいえない。

これらの適用限界の改善することが、今後の課題としてあげられる。

参考文献

[1] Himeno Benchmark: http://accc.riken.jp/HPC/HimenoBMT/.

[2] Stephane Genaud, Arnaud Giersch, Frederic Vivien: Load-balancing Scatter Operations for Grid Computing, Parallel Computing 30, pp.923-946 (2004).

[3] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphael Couturier: Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms, IEEE Transactions on Parallel and Distributed Systems, VOL.16, NO.4, April 2005, pp.289-299 (2005).

[4] 松岡正純, 鈴木和宏, 勝野昭: 自律コンピューティングに向けた HPC 向け動的負荷分散機構, 情報処理学会論文誌:コンピューティングシステム Vol.44 No.SIG 11(ACS 3), pp.89-99 (2003).

[5] 荒木拓也, 村井均, 蒲池恒彦, 妹尾義樹: データ並列言語を対象とした動的負荷分散機構の実現と評価, 並列処理シンポジウム JSPP2002, pp.131-138 (2002).

[6] 潤田浩也, 弓場敏嗣, 佐藤直人: 種々の並列・分散アプリケーションに対して容易に統合可能な動的ロードバランサ pDLB の提案と実装, 情報処理学会研究報告 2000-DPS-102, Vol.2001, pp.151-156 (2001).