C++用タスクマッピングライブラリの実装と異種混合環境での評価

山崎健生[†1]　宮本大輔[†2]　中山雅哉[†2]

近年の計算機環境はマルチコア・クラスタ・グリッド・クラウドと並列分散化が進んでいる．これらの環境では，マルチコア・マルチ CPU といった階層化非対称構造など複雑な構造でのプログラミングが課題となっており，さらに今後は S.C. (スーパーコンピュータ) や PC を組み合わせて利用するなどの複雑な環境でのプログラミングも必要となってきている．このような複雑な環境の中，並列分散処理アプリケーション開発の効率化が必要とされ，多くの言語やパラダイムが検討されている．我々はその中から明示的にタスクを資源に割当てるパラダイムに着目し，C++用ライブラリ TPDPL(Template Parallel Distributed Processing Library) として設計・実装している．本稿ではライブラリ中の PE(Processing Element) コンテナとタスクマッピングアルゴリズムの実装をおこない，それらを S.C.(T2K 東大版) とプライベートクラスタとクラウドの異種混合環境で評価し，負荷分散効果を確認した．

An Implementation of C++ Task Mapping Library and Evaluation on Heterogeneous Environments

Takeo YAMASAKI,[†1] Daisuke MIYAMOTO[†2]
and Masaya NAKAYAMA [†2]

Modern computing architectures are increasingly parallel distributed. This trend is driven by multi-core processors, grid, cluster and cloud-computing. These systems are complicated because of their scale, heterogeneous structures, and asymmetric architectures. Therefore, more productive paradigm that assists development of parallel distributed processing applications is required and has been considered. In this paper we pay attention to task mapping paradigm, and design C++ parallel distributed programing library, TPDPL (Template Parallel Distributed Processing Library), and develop PE (Processing Element) Containers and task mapping algorithms. Finally we report the performance evaluation of them on T2K open supercomputer and private cluster computer and cloud computer and we confirm the performance of TPDPL task mapping system.

1. 背　　景

複雑化・大規模化する計算機環境に対して生産性の高い開発手法が必要とされ，これまで多くの検討がされてきた．古くは共有メモリ空間を用いて計算機構造を抽象化することによってハードウェアの差異を隠蔽し生産性向上を実現してきた．しかし近年では完全な抽象化や自動並列化手法ではなく，ある程度の資源分散性を意識したコーディングをプログラマに意識させるプログラミングスタイルが注目され，様々な検討がなされている．

また近年では FPGA や GPU といった専用ハードウェアを一般的な計算に用いる試みがなされている．そのような新しいハードウェアでの開発環境は C 言語をベースしたものが多い．さらに GPU では C++用の環境が整備され，さらなる生産性向上が期待されている．

C++における並列分散処理環境としては，OpenMP や MPI，CORBA，スレッディングライブラリ，ソケットライブラリ等を組み合わせて用いたものが主流であり，生産性は高いとは言えなかった．しかしその中で，既存のライブラリを拡張してグリッドに対応したものや，部分的な文法拡張によるソフトウェア共有メモリ，テンプレートを用いた生産性向上等の検討がなされてきた．さらに C++次期標準である C++11 からスレッディングが標準ライブラリに入ることが決定されており，C++の更なる並列分散処理利用が期待される．

†1 東京大学工学系研究科
Graduate School of Engineering, The University of Tokyo
†2 東京大学情報基盤センター
Information Technology Center, The University of Tokyo

このような背景の元，我々は C++用並列分散処理ライブラリ，TPDPL(Template Parallel Distributed Processing Library) を設計・実装してきた．このライブラリでは，データやスレッドの局所性を意識したプログラミングが可能であると同時に，資源を C++独特の階層構造にて抽象化・隠蔽することにより生産性の向上を目指している．これまで我々は，資源の抽象化と非同期呼び出しの実装と評価をおこなってきた．本稿では，階層構造による資源の隠蔽による異種混合環境での自動割り当てに関する簡単な実装と評価をおこなう．

以降 2 章にて既存手法について，3 章にて設計したライブラリについて，4 章にて異種混合環境でのタスクマッピングの評価をおこない，最後に 5 章にてまとめと課題を述べる．

2. 既存の言語やライブラリ

2.1 タスクマッピング

スレッドやデータの局所性を指示する言語が注目を集めており，そのような言語ではスレッドや配列を明示的に資源に割当てることが可能である．例えば X10 では[1)]，Place と呼ばれる資源に対して Activity と呼ばれる軽量スレッドを明示的に割り当てることができる．これによりチューニング性が向上して高速化が見込める．X10 での資源の管理方法は，Place をリスト構造で管理しているが，ツリー構造で資源を管理する手法[2)] などの資源管理についての研究もなされており，一つの課題となっている．

2.2 C++による並列分散処理

C/C++での並列分散処理は，OpenMP や MPI を併用したり，C/C++標準ではないスレッディングライブラリやソケットライブラリを環境ごとに使い分けることでおこなわれてきた．そこでは複数のライブラリを併用することの生産性の低さが問題となっていたため，生産性を向上させる検討が行われてきた．例えば OpenMP を拡張して複数ノードに渡る分散メモリ環境に対応したもの (Omni OpenMP/SCASH[3)],XcalableMP[4)])，MPI を拡張してグリッドに対応したもの (Grid MPI[5)])，といったライブラリの適応範囲を広げる試みや，テンプレートを利用して統一的なインターフェースでの開発を可能とし，生産性を向上させるようなライブラリ (MPC++[6)]) が検討されてきた．資源運用に関しては，TBB[7)] 等でスレッドをプールする物などノード内のものの検討や，グリッド等の広域な環境での資源予約[8)] やプロセスレベルでの資源管理[9)] は多く検討されているが，ジョブ起動後の複数ノードでの計算資源運用についての検討は少ない．

2.3 C++11

C++は次期標準からスレッディングをサポートする[10)]．thread クラスを中心に，future，promiss といったパラダイムや，関数オブジェクトを非同期実行する async，mutex や condition_variabl といった同期機構，メモリバリアを用いた atomic operation 等が組み込まれる．これらは単一ノード内での並列処理を対象としたものでノード間連携をするようなものではなく，またスレッドプールのような資源運用に関しても組み込まれておらずその次の標準への課題となっている．

3. ライブラリの設計

我々のライブラリでは，X10 における Place と Activity のように，計算機資源である PE (processing_element) クラスに対してタスクを割当てる，といった抽象的なモデルを用意している．タスクとは関数ポインタとその引数のセットであり，非同期に実行可能な処理である．PE クラスはひとつのタスクを実行する単純な物となっている[12)]．これは MPC++にておこなわれた，スレッドや遠隔ノードに対するタスクの割り当て操作の統一に，資源クラスの概念を追加し，インターフェースを次期標準のスレッドクラスに合わせたものになる．

抽象的なタスク割り当てモデルだけでなく，PE から様々な PE を派生させることにより具体的な PE を直接操作してチューニングすることも可能となっている．例えば, スレッドを立ててそれに対してタスクを割り当てる thread_pe や，TCP や MPI による通信を用いて遠隔ノードに PE を確保し，具体的なノードを指定したタスク割り当てが可能である tcp_pe や mpi_pe などがある．また今後の実装で，GPU や FPGA などの専用ハードウェアに対応した PE を設計しそれに対する明示的な割り当てによって，より直接的なチューニングが可能となる．

抽象的なモデルではチューニング性が低く，具体的 PE の操作では生産性が低下してしまう問題がある．我々は，特殊な PE を階層構造によって順次隠蔽することで生産性の向上を狙っている．資源を確保するアロケータ，構造のテンプレートを記述したコンテナとイテレータ，イテレータを通しコンテナの各要素を操作するアルゴリズム，と順に機能を階層的に分離し，他の層の実装に依存しないプログラミングを可能とする．この構造は標準にあるメモリ資源を管理する

STL(Standard Template Library) を参考にしており既存の C++プログラマに理解しやすい構造となっている (図 1)．プログラマはチューニング性と生産性のトレードオフのもと，アルゴリズムからコンテナ，アロケータと深い層に降りていくことによってチューニングしていく．

またこの構造では計算機資源をコンテナを用いて運用するが，STL で用いられるコンテナのように様々な構造を組み合わせて使うことが可能で，環境に合わせた PE コンテナを記述したり，タスクに合わせてツリー構造やリスト構造で資源を管理するなどユーザがプログラムや環境に合わせて記述する．STL にあるコンテナはそれぞれ独自のポリシーを持って設計されており，それぞれに得意なアプリケーションがある．例えば vector はランダムアクセス性能が重要なアプリケーションに有効で，list は要素の挿入・削除が頻繁に発生するアプリケーションに有効である．これと同様に PE コンテナにおいても，アプリケーションによって PE コンテナやマッピングアルゴリズムを使い分ける．現状ではまだ PE コンテナの種類は少ないが，これまで検討されてきた様々な並列処理パラダイムを実装したり新しい PE コンテナの設計することによって，様々な種類のアプリケーションに対応していくことが可能である．このように，このライブラリが想定する対象アプリケーションは複数のアプリケーションが混じり合ったプログラムや複数の環境が混じり合った異種混合環境を対象としている．

これまで我々は，各種 PE の実装やその性能評価をおこなってきた[11)12)]．本稿ではより上位の PE コンテナとタスクマッピングアルゴリズムの実装と評価をおこなう．資源を環境に合わせてプールする PE コンテナと，異種混合環境にて PE をリスト構造で管理する PE コンテナ，for ループのような単純な繰り返しタスクを自動的に負荷分散するタスクマッピングアルゴリズムについての記述をする．

3.1 PE コンテナ

PE コンテナの層では，PE アロケータによって確保した PE の運用や構造化をおこなう．本稿では実装した三種類の PE コンテナを用いて説明をおこなう．まず単純に PE へのポインタを管理するだけの `pe_vector`，次に実資源に合わせて PE をプールする `thread_pp`, `mpi_pp`, `tcp_pp`，そして複数の種類の PE コンテナを繋いで管理する `hetero` である．以下順に述べる．

3.1.1 単純な PE コンテナ

ソース.1 にある `pe_vector` は単純なコンテナであり，内部にて `std::vector` を用いて PE へのポインタを保持している．`talloc` 関数にてタスクの割り当てが成功した場合には `join_set` を返す．`join_set` は割り当て先の PE へのポインタとタスク id，戻り値を格納する変数へのポインタといった情報を保有しており，タスクの終了待ちや戻り値の管理をおこなう．

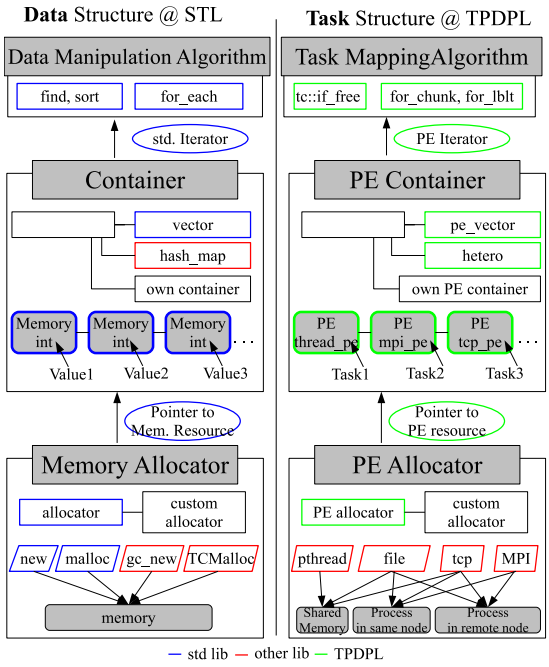


図 1 STL と設計した TPDPL の構成比較

ソースコード 1 ： `pe_vector` の使用例

```
1  int add(int a, int b){ return a+b; }
2  void test(){
3      // thread_pe 4個のコンテナ
4      pe_vector<thread_pe> pevec(4);
5      join_set js;
6      for(int i=0; i<4; i++){
7          js += pevec[i].talloc(add, 1, i);
8      }
9      js.join_all();
10 }
```

3.1.2 PE プールコンテナ

ソース.2 にある三種類の PE コンテナでは `pe_vector` と違い，有効な資源数を取得するメカニズムが追加されている．例えば `thread_pp` では CPUID 命令によって論理 CPU 数を取得し，その分だけ `full_assign` にて割り当てる．

`mpi_pp` では，MPI にて総ランク数を取得し自分以外のランクに対して論理 CPU 分の `thread_pe` を遠隔生成してプールしている．PE を用意する側のプロセ

スでは，12 行目の mpi_pe_slave() を呼び出しサーバ処理をおこなう．

tcp_pp の場合，各遠隔のクライアントプロセスにて tcp_pe を生成してサーバに接続し，set_pe 関数にて thread_pe をサーバに登録している (23 行目から)．tcp_pe の場合は，ハード的な限界値がなく full_assign が定義できないため，assign にてプールしたい数を指定している (10 行目)．

ソースコード 2 ： PE プールコンテナの例

```
1  void test(){
2      // thread_pe のプール
3      thread_pp tpp;
4      tpp.full_assign();
5      // mpi_pe のプール
6      mpi_pp mpp;
7      mpp.full_assign();
8      // tcp_pe のプール
9      tcp_pp spp;
10     spp.assign(64);
11 }
12 void mpi_pe_slave(){
13     // MPI のランク 0 以外で呼ぶ
14     mpi_pe_singleton::start_server();
15     while(mpi_pe_singleton::is_server_working()){
16         Sleep(10);
17     }
18 }
19 void tcp_pe_slave(){
20     // クライアントで呼ぶ
21     network_tools::init_sock();
22
23     int port = 50000;
24     tcp_pe mta("127.0.0.1", port);
25     thread_pp tpp;
26     tpp.full_assign();
27     for(uint32_t i=0; i<4; i++){
28         mta.talloc(&tcp_pe_singleton::set_pe,
29                 (void*)&tpp.at(i));
30     }
31     mta.talloc(set_pes, inst);
32     while(tcp_pe_singleton::is_server_working()){
33         Sleep(10);
34     }
35 }
```

3.1.3 異種混合コンテナ

ソースコード 3 ： 異種混合コンテナの例

```
1  void test(){
2      hetero<thread_pp, mpi_pp, tcp_pp> pec;
3      // thread_pe を物理 CPU 分確保
4      pec.get_pec0().full_assign();
5      // mpi_pe を全ノードの物理CPU 分確保
6      pec.get_pec1().full_assing();
7      // tcp_pe を 64PE 分確保
8      pec.get_pec2().assign(64);
9      pec.reflush();
10
11     hetero<thread_pp, mpi_pp, tcp_pp>::iterator
           it;
12     for(it=pec.begin(); it!=pec.end(); it++){
13         it.talloc(/* task */);
14     }
15     pec.join_all();
16 }
```

今回は異種混合環境に対応するため，種類の異なった PE コンテナをまとめてひとつのコンテナにする hetero コンテナを実装した．このコンテナによってこれより下層の PE 構造は隠蔽される．

ソース.3 にある異種混合コンテナでは，種類の異なる複数の PE コンテナを管理する．各資源へアクセスするには get_pecN 関数を用いる．この時 N はテンプレート引数で指定した順にインデックスが割り振られている．4 行目では get_pec0 にて thread_pp に，6 行目では get_pec1 にて mpi_pp に，8 行目では get_pec2 にて tcp_pp に PE を確保している．

12 行目から始まる for ループでは，イテレータにて begin() から end() まで順に PE にアクセスしているが，この時のアクセス順は PE が確保された順ではない．最初に thread_pp の各要素へ，次に mpi_pp，最後に tcp_pp と，テンプレートで指定した順にアクセスする．

3.2 タスクマッピングアルゴリズム

この節では，PE コンテナに対してどのようにタスクを振り分けるかといったタスクマッピングアルゴリズムについて述べる．基本的には PE イテレータによって PE に連続アクセスし，ポリシーに見合った PE にタスクをマッピングしていく．このマッピングアルゴリズムは PE や PE コンテナによらず設計することが出来る．今回は for 文を PE コンテナ内の PE に自動的にマッピングして負荷分散するアルゴリズムを 3 種類実装した．すべての PE で同じ回数分実行する even 方式，CPU のクロック比で実行回数を分ける clock 方式，for 文 1 回実行するのにかかる時間を測定し，その時間の逆比で分割する test 方式を用意した．呼び出し方法はソース.4 のようになっており for_xxx に割り当て対象となる PE 群を保持する PE コンテナを渡し，割り当てたいタスクを talloc にて指定するだけでマッピングアルゴリズムが適応される．

ソースコード 4 ： 負荷分散タスクマッピングコード

```
1  void test(){
2      thread_pp pec;
3      {
4          reducer<int> ret;
5          ret += for_even(pec).talloc(load, 1,
                   10000);
6          ret.jreduce(); // join & reduce
7      }
8      {
9          reducer<int> ret;
10         ret += for_clock(pec).talloc(load, 1,
                   10000);
```

```
11          ret.jreduce();
12      }
13      {
14          reducer<int> ret;
15          ret += for_test(pec).talloc(load, 1, 10000);
16          ret.jreduce();
17      }
18  }
```

4. Task Mapping の異種混合環境での評価

この章では手動でタスクを明示的に割り当てることが難しい環境で，自動的にタスクを PE に割り当てる負荷分散タスクマッピングアルゴリズムの評価をおこなう．このような環境でのプログラミングはこれまで難しいものであったが，今回設計したライブラリのコンテナやアルゴリズムによって簡略化されている．今回の実験はこの機構の動作確認と問題の発見を目的とする．

4.1 評 価 環 境

実験環境は，以下に示す S.C. と研究室にあるクラスタ，HaaS としてレンタルした計算機群 (StarBED[13]) の三種の異種混合環境でおこなった．それぞれの構成はまず S.C. は，T2K Open Supercomputer(東大版) HA8000．CPU は AMD Opteron 8356 2.3GHz 4 コアで 1 ノードに 4CPU 搭載され，4 ノードある．OS は RedHat Enterprise Linux 5, コンパイラは gcc version 4.1.2, MPI は ver.1.2 MPICH-MX である．次に研究室にあるクラスタは，CPU は Intel Xeon W3530 2.8GHz で 4 コアで 2 ノード．OS は ubuntu10.04LTS, コンパイラは gcc version 4.4.3, MPI は MPICH2 である．最後に StarBED 上で借りたノードは，Intel Xeon X5670 2.93GHz 6 コア 2CPU で 5 ノードある．OS は Debian 6.0.2，コンパイラは gcc4.4.5 である．

図 2 に実験での PE 構成を示す．この環境では総計 128 個の PE が生成できる．研究室にあるノードをマスタとしたマスターワーカ型となっており，マスタとなる node0 には thread_pe が 4 個，同一クラスタ上にある node1 は mpi_pe として 4 個．S.C. 上には tcp_pe として 60 個，StarBED 上にも tcp_pe として 60 個の PE が確保される．(本来 S.C. では 64 コアあるため PE を 64 本作れる．しかし，tcp_pe や mpi_pe の現在の実装では，処理の待ち受けをおこなう制御用スレッドが 1 ノードに 1 スレッド存在するため，今回は 4 スレッド分引いた 60 個の PE を用意した．StarBED やクラスタでも同様な制御スレッドは存在する．しかし物理 CPU 数以上のスレッドを立てても大きな性能劣化が確認されなかったため，物理 CPU 数分の PE を用意した．制御スレッド分を減じるかどうかはハードウェアの構成やスレッディングライブラリ，OS 等の実装などによって変わると考えられるが，現状では事前のテストすることで判断している.)

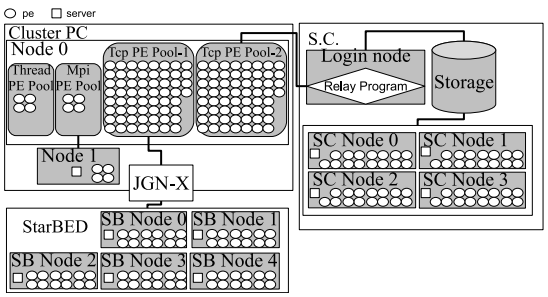


図 2 実験環境上での PE の配置図

マスタノードで用いた PE コンテナは，ソース.6 に示すとおり `hetero` コンテナであり，`thread_pp` と `mpi_pp` と `tcp_pp2` 個 の異種混合環境となる．

今回使用した S.C. の計算ノードはネットワーク的に隔離されており，外部と TCP セッションを張ることができない．このため遠隔呼び出し時のシリアライズデータをストレージにダンプする `file_pe` を用いて出力し，ログインノードがそのダンプファイルを NFS 経由で取得し，研究室の node0 に存在する `tcp_pe` に転送している．ログインノード及び計算ノード，ストレージシステム間は 1.25GB/s から 7.5GB/s のネットワークで接続されている．また研究室のマスタノードと StarBED のノードは tcp_pe にて接続されている．物理的には離れているものの JGN-X[14] によって接続されており，高速に通信可能である．StarBED 側は各ノード 1Gb/s のインターフェースを持ち研究室のマスターノードは 1Gb/s のインターフェースが 2 本，それぞれ S.C. と StarBED につながっている．研究室と S.C. は同一の建物にあるため高速な通信が期待できるものの，WAN および NFS を経由しているため外乱が発生する．

ソースコード 5 ： 実験で使用する PE コンテナ

```
1  hetero<thread pp, mpi pp, tcp pp, tcp pp> pec;
2  // thread_pe pool を物理CPU 分 (4PE)確保
3  pec.pec0.full_assign();
4  // mpi_pe で他ノードの物理CPU 分(4PE)確保
5  pec.pec1.full_assing();
6  // S.C. 60PE 確保
7  pec.pec2.assign(60);
8  // StarBED 60PE 確保
9  for(int i=0; i<60; i++){
10     pec.pec2.assign(ip[i], port[i]);
11 }
```

4.2 テストプログラム

以上で述べた環境に対して，割り当てるテストタスクは２種類用意した (ソース.6)．計算が int 型中心になる load_int 関数と，double 型が中心となる load_double 関数となっている．これは浮動小数演算が得意な環境，不得意な環境とでタスクマッピングの差を観測するために用意している．計算内容自体は両者共に簡単な計算をループするだけのもので，ライブラリ内部の挙動を観測するため，ベンチマークを用いずに単純なものを選択した．実験では load 関数を 1 から 10000000 まで実行するのにかかる時間を計測する．今回は割当てに掛かる通信遅延を考慮しない単純なタスクマッピングアルゴリズムであるので，実行時間が割り当て時間に比べて十分大きくなるよう for 文のループ回数を設定した．

ソースコード 6 ：
タスクマッピングとその対象タスク

```
1  uint64_t load_int(int64_t start, int64_t end){
2      uint64_t a=0;
3      for(uint32_t i=start; i<=end; i++)
4          for(uint32_t j=0; j<=1000; j++)
5              a += (uint64_t)(i+j)*(i-j)*(i*j)*(i/j);
6      return a;
7  }
8  uint64_t load_double(int64_t start, int64_t end){
9      uint64_t a=0;
10     for(uint32_t i=start; i<=end; i++)
11         for(uint32_t j=0; j<=1000; j++){
12             double ii=(double)i,jj=(double)j;
13             a += ((jj/ii)*(ii-jj)/(ii*jj)*(ii/jj);
14         }
15     return a;
16 }
```

タスクマッピングアルゴリズムについては，前章にて紹介した even,clock,test の三方式を用いる．even 方式では，各々10000000/68 回繰り返し，clock 方式では，各々の PE には 10000000*クロック/(クロックの総和) 分だけ，test 方式では，各々10000000/時間/(時間の逆数の総和) 分のタスクが割り当てられる．

even 方式は資源が統一された環境でもっとも高速に動作することが期待でき，今までも用いられてきた単純な割り当て方法である．clock 方式では，異種混合環境へ対応するために CPU の性能によってタスクをマッピングする．これは実行前に各 PE に性能を遠隔呼び出しで取得するため，その分のオーバーヘッドが発生する．また CPU アーキテクチャが違った場合には正確な負荷分散は期待できない．最後に test 方式では，タスクを動的に 1 回試すため呼び出し一回分のオーバーヘッドがかかるが，for 文の回数が多い場合に高い負荷分散効果が見込める．

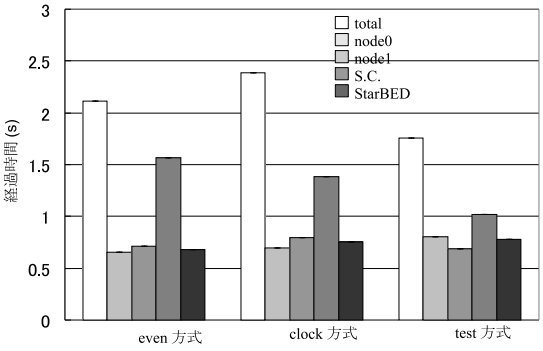


図 3 even,clock,test 三方式による load_int マッピングの実行結果

4.3 評価結果

図 3 にて load_int タスクの負荷分散の結果を示す．縦軸が経過時間になっており各マッピングアルゴリズムの結果ごとに５本の値がある．それぞれ左から順に，全体の時間，node0 で実行した時間の平均，node1 で実行した時間平均，S.C. で実行した時間，StarBED で実行した時間の平均になっている．

even 方式では 1 コアの性能で劣る S.C. での実行時間が長く，他は空き時間が長く効率が悪い．clock 方式にするとその差は改善されるが，いまだ大きな開きがある．これは，CPU アーキテクチャの違いやメモリアクセス速度の違いなどが原因であると考えられる．test 方式ではその差は改善され，負荷の分散が確認できる．

次に図 4 にて load_double タスクの負荷分散の結果を示す．double 型の場合，even の段階で負荷の均整が取れた状態となっている．clock 方式では，クロックが相対的に遅い S.C. が若干少なく，それ以外は若干多くタスクが振り分けられており，それに従って各環境での平均実行時間が変動している．一方 test 方式では負荷の均整が崩れて大きくばらついてしまっている．test 方式では一回にかかる時間を計測するが，この時間があまりに小さかった場合には大きな誤差が発生してしまう．図 5 に，この test 方式にて計測した各 PE での for 文 1 ループにかかる時間を示す．縦軸は測定時間で，横軸が PE 番号になっている．左から 4 個が node0 での値，その右 4 個が node1，その右 60 個が S.C.，その右 60 個の値が StarBED での値になる．今回特に割り当て失敗した node0 と S.C. では高いピークが出ており，ピークが出た部分についてはタスクが極端に割り振られない状態になってしまう．もう一つ大きな問題として，S.C. での試行時間の平均が他のものに比べて長くなってしまっている．この

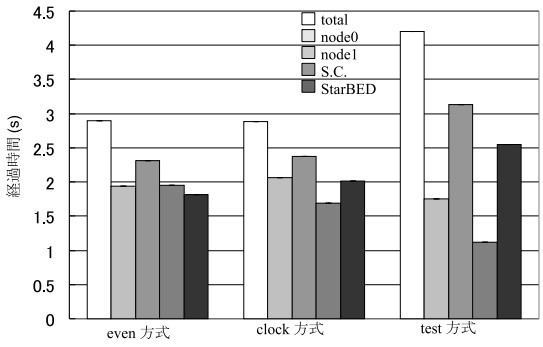


図 4 even,clock,test 三方式による load_double マッピングの実行結果

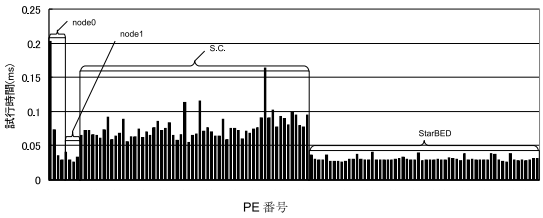


図 5 load_double での test 方式での試行時間

load_double タスクは even 方式の結果から分かる通り大きな差が出ないと考えていた．ここで計測している時間は，各 PE で呼び出された load 関数内で測定しているため，通信の遅延等のオーバーヘッドが原因ではないと考えられる．そのため，スレッドの切り替えや，コア間や CPU 間での変数の取り合い等がバックグラウンドで発生した時のコストが原因と考えられるが現在調査中である．

5. まとめ・課題

近年計算機環境は複雑化し生産性の高いプログラミング手法に関する研究が注目を集めている．我々は，その中からタスクを明示的に割り当てるパラダイムに注目し，C++用のライブラリ (TPDPL) として設計してきた．

本稿では設計したライブラリの内，コンテナによる資源管理方法，タスクマッピングアルゴリズムによる自動割り当てについて設計と実装をおこない，S.C.・クラスタ・クラウドの混合環境にて評価を行った．タスクの種類によっては大きな負荷分散効果が確認できたものの，アーキテクチャやタスクの性質によっては適切な負荷分散が出来ない事がわかった．しかしながら下層を隠蔽したタスクマッピングアルゴリズムの実装が可能であることが今回確認でき，今後，より高度なタスクマッピングアルゴリズムの実装によってさらなる生産性の向上が見込まれる．これにより複雑な異種混合環境やエクサスケールのような膨大な計算資源を用いた開発の効率化につながると考えられる．

今後の課題としては，test 方式のタイマー精度の向上や，様々な PE コンテナの実装，複雑なマッピングアルゴリズムの検討，また，PE 通信が発生するものなど具体的なアプリケーションでの定量的な評価がある．

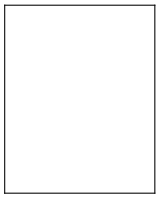
謝辞 本研究では StarBED での環境を構築するにあたり StarBED 運用チームの方々から有益な助言を頂いた．彼らからの助言と高質なテストベッドの提供に深謝の意を表する．

参 考 文 献

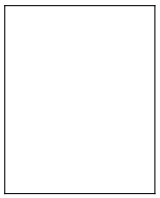
1) Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, David Grov: Report on the Programming Language X10 version 2.1, `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf` (2011)
2) Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar, Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement, Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC), October 2009.
3) 小島, 佐藤, 原田, 石川, 朴, 高橋:Ethernet によるクラスタ上での分散共有メモリ OpenMP Omni/SCASH の性能評価, 情報処理学会 HPC 研究会研究報告 2002-HPC-91-21、pp. 119-124, 2002.
4) 李珍泌、朴泰祐、佐藤三久：分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価, 情報処理学会論文誌コンピューティングシステム（ACS） Vol.3 No. 3,153-165 (2010-09-17), 1882-7829, 2010.
5) Y.Ishikawa, M.Matsuda, T.Kudoh, H.Tezuka, S.Sekiguchi:GridMPI - 通信遅延を考慮した MPI 通信ライブラリの設計, SWOPP03, 2003.
6) Yutaka Ishikawa , Atsushi Hori , Mitsuhisa Sato , Motohiko Matsuda , Jorg Nolte , Hiroshi Tezuka , Hiroki Konaka , Munenori Maeda , Kazuto Kubota :Design and Implementation of Metalevel Architecture in C++ - - MPC++ Approach - -, Reflection '96 Conference, April 20- -23, 1996.
7) Threading Building Blocks web site, `http://threadingbuildingblocks.org/` (2011)
8) 竹房あつ子, 中田 秀基, 工藤知宏, 田中良夫. :多種資源を対象とするオンラインコアロケーション手法の提案, 情報処理学会研究報告 2011-HPC-129 , 2011
9) 斉藤貴文, 千葉　立寛, 佐藤　仁, 松岡　聡:ワー

クフローアプリケーションに対する計算資源割り当ての最適化, 情報処理学会研究報告 2011-HPC-129 , 2011
10) The C++ Standards Committee http://www.open-std.org/jtc1/sc22/wg21/
11) 山崎 健生, 中山 雅哉: 並列分散処理環境におけるタスク割り当てライブラリの設計と C++での実装と評価, HPCS2011 シンポジウム論文集 IPSJ Symposium Series, Vol.2011, p.82 (2011)
12) 山崎健生、中山雅哉:C++用タスク割り当てライブラリ tpdplib の T2K オープンスーパーコンピュータ上での実装と NPB による評価, 情報処理学会，ハイパフォーマンスコンピューティング研究会，HPC-129, No.26, 2011 年 3 月
13) StarBED Project http://www.starbed.org/
14) JGN-X http://www.jgn.nict.go.jp/

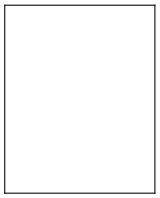
(平成 22 年 7 月 17 日受付)
(平成 22 年 9 月 17 日採録)



山崎健生（学生会員）
東京大学工学系研究科修士二年. 1986 年生まれ. 無線通信に関する研究の際，通信路シミュレータを作成これの並列化から並列分散処理に興味を持ち，修士課程より並列分散処理に関する研究に従事. 情報処理学会, 電子情報通信学会各学生員.



宮本大輔（正会員）
東京大学情報基盤センター助教（ネットワーク研究部門）。1977 年生まれ. 2009 年に奈良先端科学技術大学院大学情報科学研究科情報処理学専攻にて博士 (工学) を取得し、同年より独立行政法人情報通信研究機構セキュリティセンターのトレーサブルネットワークグループ専攻研究員として着任。2011 年から現職において，テストベッド，ネットワークセキュリティ研究に従事.



中山雅哉（正会員）
平元 東京大学大学院工学系研究科博士課程了 (工博). 現在、東京大学・情報基盤センター・准教授. 広域分散処理技術に関する研究に従事. IEEE, 情報処理学会, 電子情報通信学会各会員.