Vol.2011-HPC-129 No.26
2011/3/16

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

山崎 健生[†1]　中山 雅哉[†1]

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．ベクトル型プロセッサや，GPGPU などのヘテロジニーアスな環境や階層化非対称構造というような複雑な構造でのプログラミングが課題となっており，さらに今後は大型計算機と端末・センサ等が連携したユビキタス・コンピューティングの時代が到来すると考えられ，より一層の複雑化が予見される．このように環境の複雑化が予測される中，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．今回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，新たに C++用ライブラリ tpdplib として設計・一部実装した．本稿ではその基本部分の実装と T2K オープンスーパーコンピュータ上で性能の評価をおこない，基本部分の性能が既存の言語と遜色がないことを確認した．

An Implementation and NPB Evaluation of C++ Task Allocation Library (tpdplib) on T2K Open Supercomputer

Takeo Yamasaki[†1] and Nakayama Masaya[†1]

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 (vector processor and GPGPU (general purpose computing on graphics processing units)) and asymmetric architectures (NUMA (Non-Uniform Memory Access) architectures). Furthermore ubiquitous computing system that consists of large scale computer, terminal-PC, and sensor-PC is expected to become popular. Therefore, more productive paradigm that assists development of parallel distributed processing applications is required and have been considered. In this paper we pay attention to task mapping paradigm, and design C++ parallel distributed programing library, tpdplib, and develop a part of them. Finally we report some performance evaluation on T2K open supercomputer and we confirm that tpdplib bears comparison with existing language.

1. はじめに

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多く検討されている．古くから共有メモリ空間を用いて計算機構造を隠蔽し生産性向上を実現していたが，環境の複雑化に対応するため配列の分散割り当てや Partitioned Global Address Space (PGAS) といったパラダイムが注目されている．またデータの局所性だけでなくタスクを明示的に計算機に割当てるモデルが注目されている．例えば X10[1)] では以下のようにタスクを特定の場所で実行する．

src 1: X10 task creation

```
1 async at (here.next()) {
2     // job;
3 }
```

X10 では現在の処理が行われている場所を here キーワードにて取得できる．この例では here の次の場所を at 節で指示し，非同期的に処理を実行している．タスクの明示的な割り当てによって記述コストは増加してしまうが，自動的な割り当てによる負荷分散では実現できないチューニングが可能となり，特に異種混合環境等や通信の複雑な環境での利用が期待される．

我々はこの明示的なタスクの割り当てに注目し，C++用ライブラリとして設計してきた[2)]．ライブラリの設計では C++ Standard Template Library (STL) が持つ構造を模倣し，プログラミングインターフェースは次期 C++標準である C++0x にて標準ライブラリに導入されるスレッドライブラリを参考にしている．これにより C++プログラマが容易に並列分散処理プログラミングへと移行できるよう配慮している．また実装は現在普及している C++98 の文法の範囲内でなされたユーザレベルライブラリでありコンパイラ拡張もなく，既存の C++コードや IDE/デバッガ等の開発環境が流用可能である．

以下 2 章にてライブラリの設計について説明し，3 章にてライブラリの詳細について，4 章にて今回実装した部分の特性評価を行い，最後に 5,6 章にてまとめと課題を述べる．

†1 東京大学工学系研究科
Graduate School of Engineering, The University of Tokyo

2. ライブラリの設計

今回設計したライブラリはタスクを計算機資源へと割り当てるものである．これは資源割り当て問題の一種であるが，C++にはデータのメモリ資源に対する割り当て問題が STL の中にまとまっている．今回設計した計算機資源の割り当てライブラリはこれを参考に設計されているため，以下順に STL におけるデータ割り当て構造，tpdplib におけるタスク割り当て構造と説明する．

2.1 データ割り当て (C++ STL)

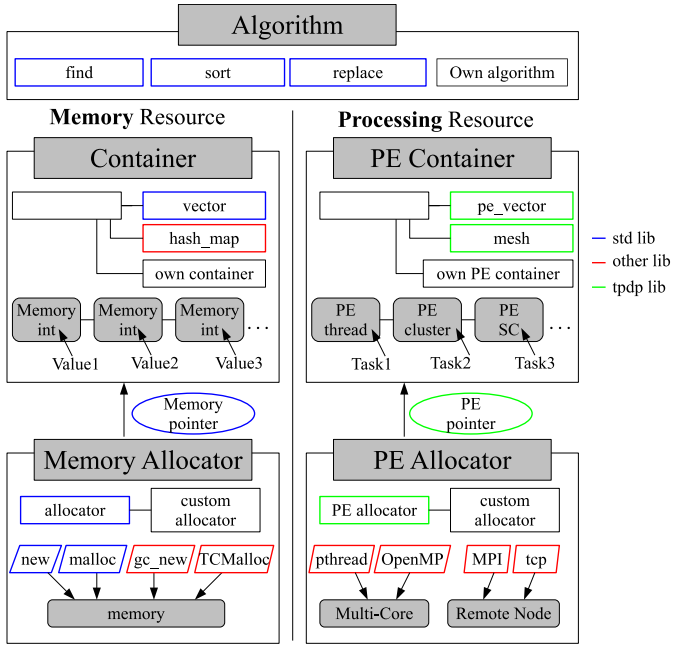

図 1 Memory and PE Allocation Library

STL ではメモリ確保・データ割り当て・アルゴリズム処理，と一連の流れがライブラリ化されている．図 1 における左側がその構造を示している．メモリを allocator によって確保し，コンテナによってデータ構造を構築する．その際コンテナの種類によってスタックやキューなど様々なデータ構造が提供される．そのようなコンテナに対してソートや探索といったアルゴリズムが実装されており，データに対する操作がおこなわれる．データ操作では，メモリ確保が大きなボトルネックになることが多いが，標準で用意されている allocator に不満があった場合，独自に Custom allocator を実装することができる．例えばメモリプールを用いて高速化したものや，ガベージコレクションのような高機能化を図った allocator を実装することができる．この Custom allocator は上位レイヤにある Container やアルゴリズムを変更せずに利用することが可能である．また，新しいデータ構造を記述した Container を自分で設計することもでき，既存のソートアルゴリズム等をそのまま利用可能である．このようにメモリ構造を隠蔽してデータ構造を構築し，またデータ構造を隠蔽してアルゴリズムを記述できるようになっている．これによって各階層の開発が分離され再利用性が高まっている．しかし実際に構築されるデータ構造と，それに対する操作であるアルゴリズムの間に不一致があった場合実行速度が遅くなってしまう．このためデータ構造に関する知識は体系化され，プログラマが適切に選択できるように教育されている．この点がプログラマに対して要求する負荷の妥協点として残っている．

2.2 タスク割り当て (tpdplib)

今回設計したライブラリでは，タスクの割り当てをデータの割り当て同様に階層化し，計算素子 (Processing Element (PE)) の確保・タスクの割り当て・アルゴリズム処理，と分離して設計出来るようにした．これにより計算機構造を意識しない自動的なタスク割り当てと，意識した明示的な割り当てが共に可能になっている．図 1 における右側が今回設計した PE 割り当てのライブラリ構造の模式図である．計算素子は PE Allocator によって確保され，PE Container によって運用され，タスクが割り当てられる．またデータ構造の時と同様に，custom PE allocator や自作 Task Container を設計することが可能である．例えば，スレッド管理を TBB[3] に一任した PE Allocator や，新規に開発した通信ライブラリを用いて遠隔ノードに PE を確保する PE Allocator を実装することが可能である．PE Container についても，通信トポロジにマッチした PE 構造を構築したり，実行するタスクの構造にマッチした構造を設計することができる．データ構造の時と同様に，実際に構築される計算機構造と，タスク構造，そしてアルゴリズムの間に不一致があった場合実行速度が遅くなってしまう．このためタスク構造に関する知識を体系化し，プログラマが適切に選択できるように教育する必要がある．

3. tpdplib の詳細とプログラミングインターフェース

以下の節にて，PE Allocator, PE Container と順に説明していく．STL および tpdplib ではユーザが Custom Allocator や自作 Container を実装することで機能を拡張することが出来る．以下の節では tpdplib が持つ機能の拡張点を中心に順に解説する．3.1〜3.4 節にて，PE Allocator と PE について，3.5〜3.6 節にて PE Container について記載する．

3.1 PE Allocator と PE

STL ではメモリ確保が Allocator によっておこなわれ，確保結果がポインタで返される．そして Container がそのポインタを運用する．これに対して今回実装したライブラリでは，PE Allocator が PE(processing_element クラス) へのポインタを返し，PE Container がその PE を運用する．今回は PE の実装をメインにおこなっており，インターフェースである processing_element クラスから派生した，thread_pe, mpi_pe クラス等を実装している．これらのクラスは直接インスタンスを生成可能であり，標準にある std::allocator を用いて操作可能である．今回は実装していないが，計算機上の論理 CPU 数分だけ thread_pe をプールするような custom PE Allocator を設計することで資源管理をおこなうことが可能である．

3.2 Task Mapping

PE に対するタスク割り当ては talloc メンバ関数を通しておこなわれる．例えば src.2 では talloc に関数ポインタと引数を渡すことで処理が非同期的に実行される．このインターフェースは C++0x から導入される thread クラスのインターフェースを参考にしている．(src.3).

src 2: task mapping interface

```
1 void func1(int a1, int a2){ printf("%d,%d", a1, a2); }
2 void test(){
3     thread_pe tpe;  // スレッドプールに処理を割当てる PE
4     int arg1=1, arg2=2;
5     int id1 = tpe.talloc(&func1, arg1, arg2); // 関数ポインタと引数を指示し,非同期実行
6     tpe.join(id1);  // タスクの終了待ち
7     int id2 = tpe.talloc(&func1, arg1, arg2); // thread と違いもう一度割り当て可能
8     tpe.join(id2);
9 }
```

src 3: C++0x std::thread interface

```
1 void func(int a1, int a2){ printf("%d,%d", a1, a2); }
2 void test(){
3     int arg1=1, arg2=2;
4     std::thread th(&func, arg1, arg2); // 関数ポインタと引数を指示し,スレッド生成
5     th.join();  // スレッドの終了待ち
6 }
```

talloc 関数は可変長引数メンバ関数になっており，可変長引数テンプレート (Variadic Templates) を用いて実装されている．この機能は C++0x からのサポートになるが，C++98 の範囲内で可変長引数テンプレートを実現する手法が boost ライブラリなどで用いられている．talloc 関数内では，渡された関数ポインタと引数をメンバに持ったファンクタ (task クラス) を生成する．生成されたファンクタの処理方法は PE の種類によって異なる．例えば thread_pe のような共有メモリ空間内で引数を共有している PE の場合は，そのままファンクタをスレッドプールに渡してタスクを実行する．一方 mpi_pe のように遠隔地にあるノードにタスクを割り当てる場合，一度バイナリデータにシリアライズした後に送信され，遠隔ノードにある待ち受けスレッドによって受信・デシリアライズした後に実行される．

PE を拡張する場合 processing_element クラスを継承し，ファンクタ (task) を処理する仮想関数をオーバーライドしたクラスを実装する．また，PE Allocator を自作することで，後述する PE Container での PE の確保・開放・運用方法をユーザが設計可能となる．現在は高度な PE 運用はしておらず，標準ライブラリの allocator にて割り当てをおこなっている．

現在実装されている遠隔呼び出し系 PE を利用する際の制約として，引数がシリアライズ可能であることと，遠隔呼び出し可能な関数は事前に通知する必要がある．シリアライズには src.4 のような記述が必要で，遠隔呼び出しの通知は src.5 のように記述する．ノード間での呼び出しではこの 2 項目による記述量の増加が問題となっている．対策としては，文法を拡張してプリプロセッサを一段増やすことで自動化が可能であるが今回実装はしていない．

src 4: Serialization

```
1 class user_class_t{ int a, b; };
2 namespace tpdp{
3     template<class ARC>
4     struct srlz<ARC, user_structure_t>{
5         void operator()(srlz_stream<ARC>& strm, user_class_t& data){
6             strm & a & b;
7         }
8     };
9 }
```

src 5: Preparing for Remote Procedure Call

```
1 void hoge_func(){ /* task */ }
2 int main(){
3     tpdp::rt_func_reflection::register_funciton( &hoge_func );
4     return 0;
5 }
```

3.3 Task Mapping の拡張

前節にてタスク割り当てに talloc を用いていた．これには引数に対し関数ポインタの他にファンクタをとることが可能で，標準の thread と同様の動作になっている．このインターフェースの他にメンバ関数を呼び出すためにインスタンスを指示したり，戻り値の格納先を指示可能なインターフェースとして rtalloc を用意している．

src 6: task mapping interface 2

```
1  struct obj{
2      int add(int a, int b){ return a+b; }
3  };
4  void test(){
5      thread_pe tpe;  // 直接 PE を生成
6      obj* inst = new obj;
7      int ret=0;
8  // ret = inst->add(1, 2);
9      int id1 = tpe.rtalloc(ret, tkolib::reducer_operator::eq, inst, &obj::add, 1, 2);
10     tpe.join(id1);
11 // ret += inst->add(1, 2);
12     int id2 = tpe.rtalloc(ret, tkolib::reducer_operator::add_eq, inst, &obj::add, 1, 2);
13     tpe.join(id2);
14     delete inst;
15 }
```

src.6 では obj 型のインスタンスから add 関数を呼び出し，結果を ret に格納している．引数の順は通常の呼び出しでの語順と同一で，=や+=等のオペレータはファンクタとしてユーザが自由に設計・指示可能である．非同期実行が終了した段階で ret に対して指示した処理が行われる．注意点として rtalloc ではファンクタは利用できない．ファンクタを利用したい場合は戻り値やインスタンスを適切に処理するファンクタを自分で作り，talloc を通じてタスクを割り当てる必要がある．

3.4 PE におけるタスク割り当ての特殊化とデータ転送

データの転送はボトルネックになることが多く，通信ライブラリを直接使うことが好ましい．これをサポートするため，タスク割り当ての特殊化が実装されている．例えば mpi_pe では，特定の関数での処理が特殊化されてお，具体的には remote_ memput, remote_memget 関数を特殊化し，遠隔ノードへのメモリアクセスでは通信ライブラリを直接利用している．

src 7: Remote Memory Access

```
1 char buf1[256]="test-test-test";
2 char buf2[256];
3 char buf3[256];
4 void test(){
5     mpi_smpe pe(1,1); // rank1 に 1 個 PE を生成
6     pe.talloc(&remote_memput, buf2, buf1, sizeof(buf1)); // buf1 の内容を遠隔の buf2 に転送
7     pe.talloc(&remote_memget, buf3, buf2, sizeof(buf2)); // 遠隔の buf2 を buf3 に転送
8     printf("%s\n", buf3);
9 }
```

src.7 では，remote_memput にて遠隔書き込みを，remote_memget にて遠隔読み込みをおこなっている．PE 内部では buf1,buf2 といったポインタがそのまま MPI 関数に渡されて転送終了までブロッキングする．このタスク割り当ての特殊化は PE ごとに設計でき，thread_pe の場合では memput,memget はそのまま memcpy を実行するように設計されている．またどの関数を特殊化するかはユーザが指示することが可能である．(src.7 は mpi 環境でグローバル変数のアドレスが同一であることを期待したサンプルコードである．)

3.5 PE Container

前節までは PE を直接生成していた．PE の確保や運用は PE Allocator を通して PE Container がおこなう予定である．PE Container の機能は，計算機が持つ階層・トポロジ情報を元に PE を自動的に用意したり，タスク構造に合わせて複数の PE を構造化することである．これにより X10 が用意している (src.1)here, prev, next といった要素を実装する予定である．この他の機能として，複数の PE に対してどのようにタスクを生成し，どのように割当てるかといった問題もここで記述する．例えば，遅延タスク生成法 (Lazy Task Creation) によって生成する

か，即時タスク生成法 (Eager Task Creation) によって生成するかといった問題や[4]，独自の評価関数によってタスクを評価し，軽量タスクはそのまま同期実行し，中量タスクはthread_peに，通信量が少ないタスクは遠隔ノードに割当てるといった動的な負荷分散機能もPE Containerのレイヤにて記述する予定である．以下単純な機能を実装したpe_vectorを元に解説する．

src 8: Example of PE Container

```
1  int add(int a, int b){ return a+b; }
2  void test(){
3      thread_pe  tpe[2];  // ローカルに直接 PE を 2 個生成
4      mpi_smpe  mpe(1,2); // rank1 に 2 個 PE を生成
5      pe_vector pevec;
6      for(int i=0; i<2; i++){
7          pevec.push_back(tpe[i]); // 管理したい PE を積む
8          pevec.push_back(mpe[i]);
9      }
10
11     join_set js; // 複数のタスクの終了待ちや強制終了をおこうクラス
12     for(int i=0; i<4; i++)
13         js += pevec.talloc(add, 1, i); // 空いている PE にタスクを割当てる
14     js.join_all();
15 }
```

pe_vectorは生成済みのprocessing_elementを複数保持可能なクラスで，PEの確保や開放，PE間の接続情報は管理しない単純なクラスである．タスクの割り当てでは空いているPEが無かった場合には何もせずに処理を返す．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強制終了をおこなう．現在実装済みのPE Containerはこれだけであるが，割り当て失敗時にそのまま同期処理するモデルや，成功するまでブロッキングするモデル，即時的にタスクを生成してキューに積むモデル，保有する全てのPEに同一の処理を割り当てるモデル等，様々なPE Containerを実装予定である．また自作Containerを設計することでユーザ自身がこの挙動を指示することができる．

3.6 Containerと Reduce処理

非同期処理を記述する上でReduce処理を記述可能であるとコード量を削減可能である．PE Containerはタスク割り当て後にjoin_setを返す．このjoin_setを利用して処理が終わるまで待ち，自動的にReduceする仕組みが実装されている．

src 9: Reducer

```
1  int add(int a, int b){return a+b;}
2  struct custom_dump_reduce_ope{
3      template<class ret_type>
4      void operator()(ret_type& out, const ret_type& in){
5          printf("reduce␣result:␣out=%d,␣in=%d\n", out, in);
6          out = in;
7      }
8  };
9  void test(){
10     tpdp::reducer<int> ret=0;
11     ret = pevec.rtalloc( &add, 0, 1 );
12     ret += pevec.rtalloc( &add, 1, 2 );
13     ret -= pevec.rtalloc( &add, 1, 3 );
14     ret(tpdp::reducer_operator::max) = pevec.rtalloc( &add, 3, 1 );
15     ret.jreduce();
16     ret(custom_dump_reduce_ope()) = pevec.rtalloc( &add, 3, 4 );
17     ret.jreduce();
18 }
```

src.9では，10行目にてint型変数へのReduce処理を実行するreducer⟨int⟩型の変数retが宣言されている．PE Containerであるpevecにtaskを割り当てると，retに対してjoin_setが返り，retがこれを保有していく．最終的に15行目のret.jreduce(); にて全ての処理を待った後Reduceする．この時Reduce処理は，11行目では単純な代入処理が，12,13行目では+=,-=に従い加減算が，14行目では現在のretの値と，実行結果とのmaxが取られる．14行目にてret(〜)に渡されているのはファンクタであり，ユーザが自由に定義したものが指定可能である．例えば16行目にて引数と戻り値をダンプするファンクタ (2行目にて定義) を指定している．

src.9の例ではPE Containerが同時に複数のPEにタスクを割り当てていないのでReduce処理ではなくfutureのような動作になっている．ただしReduceは処理が終了したものが優先的におこなうので，futureのように評価の順に制約はない．これは探索問題等で枝刈りをする場合に有用である．全体に割当てるPE Containerが実装されると，MPI_ReduceやMPI_Gather等が実装可能となる．このように割り当てとjoin_setを組み合わせた処理に関しても様々考えられ，今後実装予定であるとともにユーザが任意に設計可能である．

4. 性能評価

今回実メインに装した PE によるタスク割り当てや，遠隔メモリアクセス等，ライブラリの基本部分の性能についてベンチマークプログラム用い評価した．ベンチマークプログラムについては NPB(NASA Advanced Supercomputing Parallel Benchmark) Ver.3.3.1 を用い，IS と EP を tpdplib にて実装し UPC(OpenMP)，UPC(MPI) と比較した[5]．テスト環境は T2K Open Supercomputer(東大版) HA8000 を用いた．CPU は AMD Opteron 8356 2.3GHz 4 コアで 1 ノードに 4CPU 搭載された NUMA 環境である．主記憶は 32GB，OS は RedHat Enterprise Linux 5, コンパイラは gcc version 4.1.2, UPC は Berkeley UPC-ver.2.12.0 を用いた．MPI は ver.1.2 MPICH-MX である．以下の節にて順に評価結果を示す．

4.1 EP Benchmark

EP は擬似乱数の生成プログラムであり，乗算合同法により一様乱数と正規乱数を生成し統計値を計算する．今回用いたのは CLASS は A(M=28) である．このベンチマークは特徴として PE 間での通信が割り当て時だけでかつ非常に少ないので並列化しやすい．thread_pe または mpi_pe の talloc を用いて実装されている．

図 2 に UPC(OpenMP) と thread_pe による EP の実行結果をしめす．

図 2 は縦軸が経過時間，横軸はスレッド数になっており，同等の実行結果が得られていることが確認できる．EP では割り当て処理がスレッド数しか発生せず，割り当て時に発生するオーバーヘッドの差は評価できない．また SMP 環境なので通信もシリアライズも発生しないので差が出ないのは自明である．並列化しやすい問題においては正常に動作していることが確認された．

続いて表 1 に UPC(MPI) での EP の実行結果を，表 2 に mpi_pe を用いた EP の実行結果を示す．表中値の単位は秒である．MPI 環境でも差は見られず，並列度が高いタスクでは mpi_pe が予期通り動作していることが確認された．

表 1 EP performance implemented in UPC(MPI)

		ノード当たりのスレッド数				
		1	2	4	8	16
ノード数	1	64.39	32.44	16.18	8.14	4.08
	2	32.34	16.23	8.17	4.07	2.04
	4	16.26	8.12	4.09	2.05	1.04

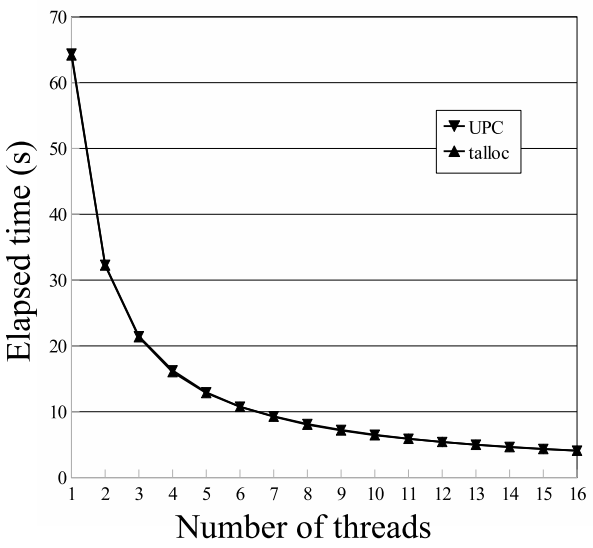

図 2 EP performance implemented in UPC(OpenMP) and tpdplib(thread_pe)

表 2 EP performance implemented in tpdplib(mpi_pe)

		ノード当たりのスレッド数				
		1	2	4	8	16
ノード数	1	64.25	32.32	16.10	8.09	4.06
	2	32.39	16.17	8.11	4.05	2.04
	4	16.17	8.09	4.06	2.03	1.03

4.2 IS Benchmark

IS は Int 型整数のソートプログラムである．ソートはバケツソートを用いており，PE 数は 2 の冪乗である必要がある．ベンチマークは CLASS A を用いており，要素数は 8,388,608 個で，繰り返し数は 10 回になっている．IS ベンチマークは特徴として，バケツのカウント後とバケツの全交換時に同期と通信が発生する．tpdplib における実装は遠隔メモリ書き込みが用いられている．

図 3 に UPC(OpenMP) と thread_pe による IS の実行結果を示す．縦軸が経過時間，横

軸がスレッド数である．

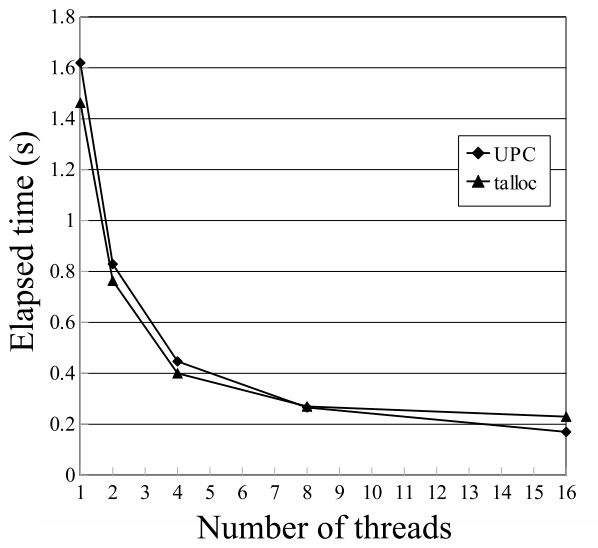

図 3 IS performance implemented in UPC(OpenMP) and tpdplib(thread_pe)

図よりいくつか特性の差が確認できる．4 スレッド以下では UPC が PGAS のためオーバーヘッドが大きくなっている．一方 tpdplib では 4thread 以上の環境では CPU 間通信が発生するため特性が劣化している．現在 pe_thread はデータのローカリティを考慮した割り当てをおこなっていない．この機能は PE Container に実装される予定で今後の課題となっている．

続いて表 3 に UPC(MPI) での IS の実行結果を，表 4 に mpi_pe を用いた IS の実行結果を示す．表中値の単位は秒である．

UPC の複数ノード時に特性の劣化が確認できる．これは PGAS の維持コストによるものと考えられる．またスレッド数が多い環境では，UPC(MPI) では PGAS のマネージャスレッドが内在するためか処理が終わらなかった．また mpi_pe においても本来 MPI 通信をおこなうマネージャスレッドが存在するが，fiber 機構による擬似並列処理よってスレッドが 1 本に統一されている．このため処理が終わる程度にコストは抑えられているものの特性の劣化が確認された．残るコストはメモリの遠隔書き込み系によるものと予想されるが，詳細は現在検証中である．また現状は mpi_pe と thread_pe を別々に評価したが，ハイブリッド化が可能であり，それによって mpi のマネージスレッドが統一されるのでコストを削減可能であると考えられる．PE 構造の階層化は PE Container の機能で今後の課題となっている．

表 3 IS performance implemented in UPC(MPI)

		ノード当たりのスレッド数				
		1	2	4	8	16
ノード数	1	1.63	0.86	0.49	0.29	0.17
	2	0.99	0.56	0.36	0.23	∞
	4	0.61	0.37	0.23	0.21	∞

表 4 IS performance implemented in tpdplib(mpi_pe)

		ノード当たりのスレッド数				
		1	2	4	8	16
ノード数	1	1.62	0.95	0.50	0.28	0.17
	2	0.85	0.48	0.26	0.20	0.32
	4	0.43	0.24	0.18	0.24	0.54

5. ま と め

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．今回我々は，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 tpdplib として設計した．ライブラリを階層的に設計することでプログラムの再利用性を向上し，抽象化したインターフェースにてプログラミングコストの低減を狙っている．この階層構造や表層のプログラミングインターフェースは標準ライブラリを参考にしており C++プログラマにとって直感的に理解しやすい．ライブラリは C++98 のユーザレベルライブラリとして提供されるので，既存の開発環境にそのまま使え，既存のプログラム資産にすぐに適用可能である．

今回は設計したライブラリの内，キーとなるタスクの割り当てや，並列分散処理において性能に直結する部分を中心に実装した．ライブラリの基本性能を評価するため，二つのベンチマークプログラムを用いて評価した．並列性が高い EP ベンチマークでは性能に差は見られず，タスクの割り当てに関する実装が予期したとおり動作していることが確認された．また並列性が低く，大きな通信が必要な IS ベンチマークでは，同等の特性がでたもの

Vol.2011-HPC-129 No.26
2011/3/16

の，thread_pe では NUMA 環境に対する割り当てができていないためスケーラビリティの低下が確認された．また分散メモリ環境では各ノードでスレッド飽和時に特性の劣化が確認され，それ以外の環境では同等の特性が確認された．これらの特性劣化は今回実装していない PE Container によって解決可能であると考えられる．今回測定したライブラリの基本部位である PE の特性としては既存の言語と遜色が無いことが確認できた．

6. 課　　題

ベンチマークにて明らかになった問題点の調査と解決が課題としてある．また基本特性を NPB によって評価したが，呼び出しコストや通信容量といった特性の定量的な評価が必要である．次に今回実装が及ばなかった PE Container の実装，機能的なタスク割り当ての実装が課題となっている．また異種混合環境や広域分散処理環境での機能的な割り当てのテストのため，GPU やクラウド等に対する割り当て機能を実装する必要がある．また，分散オブジェクトや配列の分散割り当て，共有メモリといったタスク割り当て以外のパラダイムについても実装，または他のライブラリと共存する仕組みを検討する必要がある．

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) 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)
3) Threading Building Blocks web site, http://threadingbuildingblocks.org/ (2011)
4) Eric Mohr, David A. Kranz, Robert H. Halstead, Jr: Lazy task creation: a technique for increasing the granularity of parallel programs, IEEE Transactions on Parallel and Distributed Systems, Vol.2, p.185-197 (1991)
5) The UPC Consortium, Jr: UPC Language Specification v1.2, http://upc.gwu.edu/documentation.html (2011)