

平成 24 年度修士論文

C++用並列分散処理テンプレートライブラリ
PDPTL の設計と実装・評価

Design, Implementation and Performance Analysis of C++ Parallel Distributed
Processing Template Library

指導教員： 中山雅哉 准教授

2012 年 2 月 8 日提出

東京大学院 工学系研究科
電気系工学専攻

37106506

山崎 健生

目次

第 1 章 序論	1
1.1 研究背景	2
1.2 本論文の構成	2
第 2 章 背景技術	3
2.1 背景とそこでの課題	4
2.2 近年用いられる手法	5
2.2.1 分散共有メモリへの対応	5
2.2.2 異種混合環境への対応	5
2.2.3 ラージスケール環境への対応	5
2.3 C++での並列分散処理の現状	6
2.3.1 C++用ライブラリ	6
2.3.2 C++での問題点	7
2.4 課題のまとめ	7
第 3 章 ライブラリの設計	8
3.1 デザインコンセプト	9
3.1.1 データ構造とデータ割り当て (C++ STL)	11
3.1.2 PE 構造と PE 割り当て (PDPTL)	12
3.2 異種混合環境への対応	13
3.2.1 具体的な計算資源への割り当て	13
3.2.2 タスクマッピングによる割り当ての自動化	15
3.3 ラージスケール環境への対応	16
3.4 分散共有メモリへの対応	16
3.5 特徴とまとめ	17
第 4 章 ライブラリの実装	19
4.1 PE クラス	20
4.1.1 <code>talloc</code> によるタスク割り当て	20
4.1.2 <code>talloc</code> の拡張	21

4.1.3	PE におけるタスク割り当ての特殊化とデータ転送	22
4.2	PE アロケータ	23
4.3	PE コンテナ	23
4.3.1	単純な PE コンテナ	23
4.3.2	PE プールコンテナ	24
4.3.3	異種混合コンテナ	25
4.3.4	PE コンテナとリダクション処理	26
4.4	タスク操作アルゴリズム	27
4.5	その他高生産性パラダイム	28
4.5.1	for 文の自動展開	28
4.5.2	分散配列	29
第 5 章	ライブラリの評価	31
5.1	PDPTL のモデルの評価	32
5.1.1	STL と PDPTL の双対性の評価	32
5.1.2	STL と PDPTL の記述手法の一致性の評価	34
5.1.3	STL と PDPTL の仕様の一致性の評価	36
5.1.4	設計モデル評価のまとめ	38
5.2	PE クラスの評価	39
5.2.1	ベンチマークによる基礎性能の評価	39
5.2.2	PE クラスのスケール実験	43
5.2.3	ライン数の評価	46
5.2.4	基礎性能のまとめ	46
5.3	分散配列の評価	47
5.3.1	評価環境	47
5.3.2	テストプログラム	47
5.3.3	評価結果	49
5.4	処理・資源の管理手法の評価	51
5.4.1	評価環境	52
5.4.2	テストプログラム	52
5.4.3	評価結果	58
5.5	異種混合環境でのタスクマッピング機構の評価	61
5.5.1	評価環境	61

5.5.2	テストプログラム	63
5.5.3	基礎評価・遠隔呼び出し性能	64
5.5.4	評価結果	66
5.6	評価のまとめ	69
第 6 章	結論	71
6.1	まとめ	72
6.2	課題	72
参 考 文 献		75
発 表 文 献		78

第 1 章

序論

1.1 研究背景

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、さらに今後はクラウドや S.C. (スーパーコンピュータ)、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た。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。またスレッディングが C++ 標準ライブラリに入り、GPGPU (General-purpose Computing on Graphics Processing Units) が C++ に対応するなど C++ の HPC (High Performance Computing) 利用に需要が高まっている。

そのような背景のもと我々は複雑化する並列分散処理環境に対応可能な C++ 用ライブラリの設計と実装を行なっている。このライブラリによって、近年注目されている並列分散処理パラダイムを C++ で利用可能とすると同時に複雑化する環境でのプログラミングスタイルを提案する。

1.2 本論文の構成

本論文は、以下のような構成になっている。

- 第 2 章では、背景技術となる明示的な割り当てと C++ での並列分散処理の現状について述べる。
- 第 3 章では、本稿にて提案する C++ 並列分散処理処理ライブラリの設計に関して述べる。
- 第 4 章では、そのライブラリの実装に関する詳細を述べる。
- 第 5 章では、ライブラリの評価実験について述べる。
- 第 6 章では、まとめと今後への課題について述べる。

第 2 章

背景技術

2.1 背景とそこでの課題

序論にて述べたように、複雑化する計算機環境での生産性向上を目指して様々な検討が行われてきている。主に既存手法では、抽象的なプログラミングモデルを提案し複雑なハードウェアの構成などを隠蔽することで生産性の向上を狙っている。

例えば、OpenMP⁽¹⁾では、プリプロセッサディレクティブによって並列化を指示するモデルを用いている。MPI⁽²⁾ではメッセージパッシングというパラダイムを用いて並列分散処理を記述している。X10⁽³⁾では Place,Activity という抽象的なモデルを用い、データやスレッドの明示的な割り当てによってプログラムを記述していく。

このように既存手法が抱える問題は、「どのような抽象モデルを用い、そのモデルの中でどのようにパフォーマンスを向上させるか」といった問題である。理想的なものは「逐次処理モデルと同一のプログラミングモデル・完全な自動並列分散化によるパフォーマンスチューニング」であるが、自動並列分散化でのチューニングの限界から、プログラミングモデルは徐々に抽象度を落としていっている。

この問題は最終的に生産性とチューニング性のトレードオフの選択問題になる。これらの両立を阻む主要な要因は

- 分散共有メモリへの対応
- ラージスケール環境への対応
- 異種混合環境への対応

の3点がある。

これらの問題点に対して、これまで様々な提案がなされている。分散共有メモリ環境問題には、MPI と OpenMP でのハイブリッドプログラミング⁽⁵⁾や、ソフトウェア分散共有メモリライブラリ⁽⁴⁾を用いる手法などが、ラージスケール環境問題には、プロセスレベルでのワークフロー制御⁽⁶⁾などの資源管理手法などが利用されてきた。残る異種混合環境問題についてはできるだけそのような環境は避け、均質にスケールした環境がよく利用されてきた。しかしながら、エクサスケール環境を目指す上では、SIMD 命令を搭載した計算機や GPU・アクセレレータのような計算資源と通常の CPU との混成環境化が進むと考えられており、そのような環境でのプログラミング手法が課題となっている。

2.2 近年用いられる手法

近年注目されているものとして、スレッドやデータの局所性を指示する言語がある。

2.2.1 分散共有メモリへの対応

X10 では分散共有メモリ問題に APGAS(Asynchronous Partitioned Global Address Space) と呼ばれるメモリモデルによって対応している。APGAS では複数の計算機間で共有メモリを構築するが、ノードごとにメモリが分割されていることをプログラマに明示的に意識させる。そのメモリモデルでデータを共有したり、複数の計算機に跨って配列を割り当てたり、特定の計算機にデータを割り当てることで生産性を向上させている。さらに 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により、データの移動を最低限に抑えることができる。Activity は `acync` によって生成することができるが、このとき `acync at(PLACE)...` にてタスクを実行する Place を指定できる。

2.2.2 異種混合環境への対応

また、Place といった抽象的なインターフェースだけではなく、GPU のようなより具体的な資源に対しても `acync at(gpu)...` のように割り当てが可能で、異種混合環境にも対応している。X10 での資源の管理方法は、計算機資源を抽象的な Place クラスにてリストで繋いで管理しているが、より具体的にメモリの階層構造をあらわすクラスをツリー構造で繋いで資源を管理する手法⁽⁷⁾など資源管理についての研究がなされている。この計算機資源手法や具体的な資源を記述する方法がひとつの課題となっている。

2.2.3 ラージスケール環境への対応

X10 では10万スレッド規模の資源を容易に使い分けできるような言語を目指している。そのために大量の資源とタスクを自動的に運用管理する仕組みを備えている。各 Place は Activity を溜めるキューを持ち、キューが空いた Activity が他の Place のキューから Activity を奪うことで負荷分散をはかるワークスティーリ

ング⁽⁸⁾に対応している．このような処理の管理方法も様々提案されており課題となっている．

2.3 C++での並列分散処理の現状

C++では，古くから NS2⁽¹⁶⁾ や Qualnet⁽¹⁷⁾ といったネットワークシミュレータや，Havok⁽¹⁸⁾ や Bullet⁽¹⁹⁾ といった物理演算エンジンなど様々な計算機応用を多く持っている．マルチコアや GPGPU，クラウドといった計算機環境の変化に伴ってそれらアプリケーションに対する並列分散処理利用の期待が高まっている．

2.3.1 C++用ライブラリ

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⁽⁹⁾，XcalableMP⁽¹⁰⁾)，MPI を拡張してグリッドに対応したもの (Grid MPI⁽¹¹⁾)，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++⁽¹²⁾) が検討されてきた．資源運用に関しては，OpenMP や TBB⁽¹³⁾ 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⁽¹⁴⁾ やプロセスレベルでの資源管理⁽⁶⁾ は検討されているものの，複数ノード間におけるスレッドレベルでの資源管理に関する検討は少なく，ライブラリを組み合わせる自作する例が多い．

また，GPU を利用した計算では古くから，シェーダと呼ばれるプログラムを GPU 上で実行する環境があった．これはベクトル計算をアセンブラで記述するもので生産性が低かったが，HLSL(High Level Shader Language) と呼ばれる C ライクなシェーダ言語が登場した．さらに近年では GPGPU にて，CUDA と呼ばれる C++ に対応したライブラリが登場している．

さらに C++は次期標準からスレッディングをサポートする⁽¹⁵⁾．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はその次の標準への課題となっている．

2.3.2 C++での問題点

このように，現状の C++での並列分散処理は C 言語をベースにしたライブラリを拡張したものであったり，C++プログラム外でプロセス制御するもの，TBB のように C++ベースのライブラリであっても共有メモリ環境用であった．これら様々なライブラリについて学習し，環境を整備し，C++用にラップし，組み合わせてプログラム開発する必要があったが，C++の次期標準の登場とともにそれらライブラリが標準の並列処理ライブラリに統一されたものが望まれる．

2.4 課題のまとめ

今回我々が注目した問題点は，C++の標準に統一された並列分散処理ライブラリが必要であるという点．また並列分散処理の分野では，分散共有メモリ・ラージスケール環境・異種混合環境の 3 種の問題を解決できる必要がある．そのなかで明示的な割り当てモデルが注目され，資源の記述方法・資源の管理方法・処理の管理方法に関してが課題として残っている．

第 3 章

ライブラリの設計

この章では、今回実装した並列分散処理ライブラリ PDPTL(Parallel Distributed Processing Template Library) の設計について述べる。C++11 で標準化された並列処理に合わせた非同期呼び出しインターフェースの設計や、

- 分散共有メモリへの対応
- ラージスケール環境での、非対称性や階層構造への対応
- 異種混合環境への対応

の3種の背景問題に対する解決手法をどのようにC++の体系にまとめるか、また課題となっていた計算資源の記述方法・計算資源の管理方法・処理の管理方法に関してどのようにアプローチするかを述べる。

以下、3.1 節にてデザインコンセプトについて、3.2 節にて異種混合環境への対応について、3.4 節にてラージスケール環境への対応について、3.4 節にて分散共有メモリへの対応について順に述べ、3.5 節にてPDPTLの特徴とまとめについて述べる。

3.1 デザインコンセプト

PDPTL では、複雑化が予想される処理と計算資源の管理方法を体系的に整理し、記述可能なモデルを提案し、C++用ライブラリとして設計する。このモデルを用いて、近年注目されている、データやスレッドを明確に資源に割り当てるモデルがC++上で実装可能となるように設計している。これにより複雑な環境にて環境を隠蔽したプログラミングを可能とする。

処理(タスク)の明示的な割り当てや資源の管理方法は、資源割り当て問題の一種である。C++標準ライブラリにはメモリ資源に対するデータの割り当てに関してインターフェースがまとまっている。これはテンプレートによる階層構造になっている。今回はこれを計算資源(PE)と処理(タスク)の割り当て問題に応用する。

既存の言語が抱えていた問題の一つである具体的な計算資源の記述は、抽象的な計算機クラス(c.f. X10 の Place)のプログラミングインターフェースから、異種混合環境へ対応するための具体的なプログラミングインターフェースにどのようにスムーズに移行するかといった問題に置き換えることが出来る。このライブラリではSTL同様に階層構造を順に登り下りすることによって抽象度の変化に対応する。

以下 STL におけるメモリ資源の管理について説明し，今回設計した PE 資源の管理について順に述べる．

3.1.1 データ構造とデータ割り当て (C++ STL)

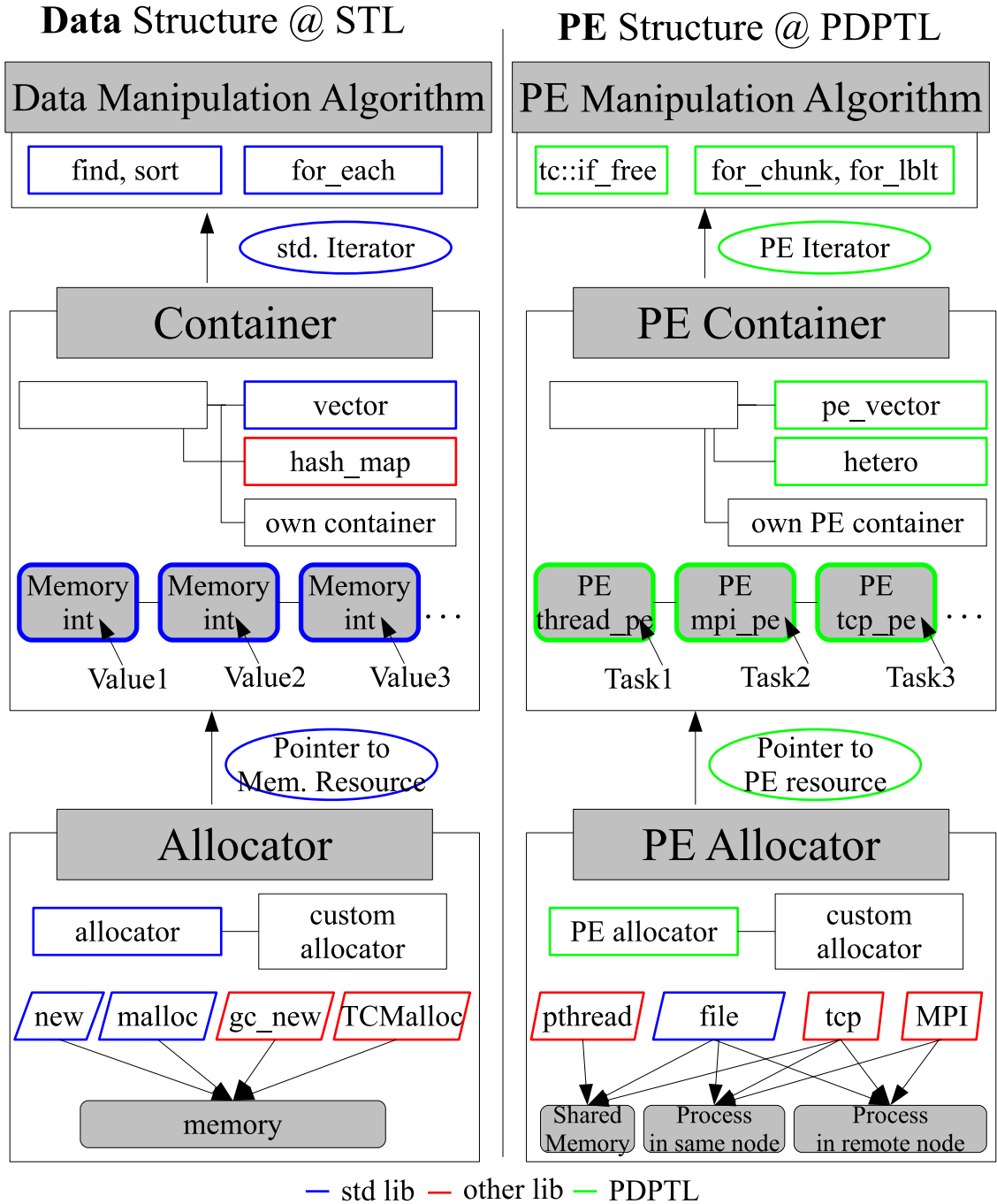


図 3.1: データ構造と PE 構造の比較図

STLでは、メモリ確保・データ構造・データ操作アルゴリズム、と一連の流れがライブラリ化されている。図3.1における左側がその構造を示している。メモリをアロケータによって確保し、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の様々なデータ構造が記述可能である。そのようなコンテナに対してソートなどの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、STLに用意されているアロケータに不満があった場合、独自にカスタムアロケータ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アロケータを実装することができる。このカスタムアロケータは上位レイヤにあるコンテナやデータ操作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コンテナ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。このようにメモリの位置やデータの種別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データ操作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、それに対するデータ操作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。

3.1.2 PE 構造と PE 割り当て (PDPTL)

PDPTLでは、スレッド確保・PE 構造・PE 操作アルゴリズム、と一連の流れがライブラリ化されている。図3.1における右側がその構造を示している。スレッドをPEアロケータによって確保し、PEコンテナによってPE構造を構築する。その際PEコンテナの種類によって、MPI クラスタや異種混合環境などの様々なPE構造が記述可能である。そのようなPEコンテナに対してタスクマッピングなどのアルゴリズムが実装されており、PEに対する操作がおこなわれる。PE操作ではスレッドの遠隔確保が大きなボトルネックになることが多いが、PDPTLに用意されているPEアロケータに不満があった場合、独自にカスタムPEアロケータを実装することが出来る。例えばスレッドプールを用いて高速化したものや、最新の通信ライブラリを用いて遠隔にPEを確保するような高機能化を図ったPEアロケータ

タを実装することができる．このカスタム PE アロケータは上位レイヤにある PE コンテナや PE 操作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 PE 構造を記述した PE コンテナを自分で設計することもでき，既存のタスクマッピングアルゴリズム等をそのまま利用可能である．このようにスレッドの位置や PE の種類を隠蔽して PE 構造を構築し，また PE 構造を隠蔽して PE 操作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 PE 構造と，それに対する PE 操作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 PE 構造に関する知識は体系化し，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．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．

3.2 異種混合環境への対応

3.2.1 具体的な計算資源への割り当て

我々のライブラリでは，X10 における Place と Activity と同様に，計算機資源である PE (processing_element) クラスに対してタスクを割り当てる，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⁽²⁰⁾．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 (コード.2) に合わせたものになる．標準の thread ではコンストラクタで指定したタスクしか実効できなかったが，PDPTL では PE クラスのインスタンスに `talloc` 関数でタスクを割り当てることにより，同一の資源 (スレッド) にタスクを繰り返し割り当てることのできる (コード 11.9-12,) ．

コード 1: :

task allocation interface

```
1 void func1( int a1, int a2){
2     printf( " %d ,%d ", a1, a2);
3 }
4 void func2( int a1, int a2){
5     printf( " %d ,%d ", a2, a1);
6 }
7 void test( processing_element& pe){
8     int arg1=1, arg2=2;
9     pe.talloc(& func1, arg1, arg2);
10    pe.join_task();
11    pe.talloc(& func2, arg1, arg2);
12    pe.join_task();
13 }
```

コード 2: :

C++0x std::thread interface

```
1 void func( int a1, int a2){
2     printf( " %d ,%d ", a1, a2);
3 }
4 void test(){
5     int arg1=1, arg2=2;
6     std::thread th(& func, arg1, arg2);
7     th.join();
8 }
```

PDPTL では、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`、ファイルシステムを経由して呼び出す `file_pe`、非同期に実行せずに呼び出したスレッドがそのまま実行する `self_pe`、軽量スレッドに実行させる `fiber_pe` などがある。これらは、仮想基本クラスとして `processing_element` クラスを持ち、まったく同一のインターフェースによって扱うことが出来る (コード.3)。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。

コード 3: :

派生 PE クラス

```
1 void test(){
2     thread_pe pe1;
3     tcp_pe pe2( " 192.168.0.1 ", 1000); // indicate ip and port
4     mpi_pe pe3(3); // indicate mpi rank
5     pe1.talloc(& func, arg1, arg2);
6     pe2.talloc(& func, arg1, arg2);
7     pe3.talloc(& func, arg1, arg2);
8     pe1.join_task();
9     pe2.join_task();
10    pe3.join_task();
11 }
```

この具体的な PE への割り当てにより、複雑化の原因であった異種混合環境に対応する。能力に合わせて負荷を分散したり、CPU に得意な計算は CPU へ、GPU に得意な計算は GPU へ、といったタスクの割り当てが可能となる。さらに、プログラムを別の計算機環境へ移植するといった異種環境の使い分けにたいしても有効で、計算機環境を記述するタイプの PE コンテナを用いることで容易に計算機環境の変動に追従する。

しかしながら問題点としてタスクと環境に関する前提知識が必要であり、またプログラムのライン数が増加する。

3.2.2 タスクマッピングによる割り当ての自動化

異種混合環境などではタスクのロードバランシングが必要となってくる。PDPTL ではこのような問題は PE コンテナ中にある PE 内で自動的にパラメータ調整してタスクを配分するタスクマッピングアルゴリズムとして用意する。これによって異種混合環境での割り当ての自動化を目指している。コード 4 では for 文のような単一のタスクを繰り返すものを自動的に負荷分散する。

コード 4: :

タスクマッピングアルゴリズム

```
1 void func( int start, int end){}
2 void test(){
3     pe_vector< tcp_pe> pec(20); // pe_vector は PE コンテナ
4     for_test( pec).talloc( & func, 1, 100000);
5 }
```

またタスク割り当ての自動化だけでなく資源管理の自動化についても，利用可能な資源を検出して資源割り当てを行うカスタム PE アロケータを用いることで，環境の変動にも自動的に追従することが可能である．

このようにテンプレート階層構造の上層でのマッピングや下層での資源管理の自動化によって生産性の高いプログラミングも可能とする．

3.3 ラージスケール環境への対応

PE コンテナを用いることで，大量の計算機を用いたプログラミングを容易にする．プログラマは PE コンテナを複数用いることで，大量にある計算機を構造化したりアルゴリズムごとに資源を使い分けることが可能になる (コード.5). 単純なマスタワーカ型の `pe_vector`，単一のタスクをすべての PE で実行する `spmd` コンテナ，PE 間で相互呼び出し可能な `mesh` コンテナなど，PE 構造の典型例を組み合わせてプログラミングすることが出来る．また，タスクスティーリングなどのタスクの管理方法を部分的に切り替えるといったことも，PE コンテナを切り替えることによって容易に行える．

コード 5: :

PE コンテナによる資源の使い分け

```
1 void test(){
2     pe_vector< mpi_pe> pec1(10); // マスタワーカ型
3     spmd< mpi_pe> pec2(20); // SPMD型 (未実装)
4     tree< mpi_pe> pec3(50); // tree 構造 (未実装)
5     alg_func1(pec1);
6     alg_func2(pec2);
7     alg_func3(pec3);
8 }
```

3.4 分散共有メモリへの対応

分散共有メモリへの対応は，`memcpy` のようなメモリ操作 API を遠隔呼び出しするリモートメモリアクセス (4.1.3 節参照) や，PE コンテナと STL のコンテナを複合して設計した分散コンテナなど，テンプレートを用いたライブラリでデータを共有していく．例えば分散配列である `darray` は指定された PE コンテナ内で配列を分担する．`dmirror` では変数を PE コンテナ中のすべての PE 上に生成し保

持する .

コード 6: :

高生産パラダイムの設計

```
1 void data_share(){
2     pe_vector< mpi_pe> pec(10);
3     darray< int> data1(1000, pec); // 要素数と PE コンテナを指定
4     dmirror< int> data2( pec); // PE コンテナを指定
5     data1[10] = 100;
6     data2 = '20';
7     data2. sync();
8 }
9 void func( barrier& wall){
10     wall. join();
11 }
12 void barrier_test(){
13     spmd< mpi_pe> pec(10);
14     barrier wall( pec); // PE コンテナを指定
15     pec. talloc(& func, wall);
16 }
17 void error_test(){
18     fault_tolerance< mpi_pe> pec(3); // PE を 3 多重
19     pec. talloc(& func);
20 }
```

このようにメモリの分散共有性は PE コンテナの中に隠蔽され，高い生産性を狙えると共に，PE コンテナを明示的に組み替えることでデータの分散割り当てをユーザが明示的に意識することも可能になっている .

3.5 特徴とまとめ

PDPTL ではテンプレートの階層構造にて体系化し，C++ライブラリとして標準に合わせて設計した .

- 分散共有メモリへの対応
- ラージスケール環境への対応
- 異種混合環境への対応

といった問題点に対しては，既存の言語同様に明示的なタスクやデータの割り当てや，資源の具体化，処理・資源管理手法を用意して対応する . 既存言語との大きな違いとしては，抽象的なモデルを前提に，これをどのように具体化していくかといった手法ではなく，C++STL 同様に具体的な資源クラスを記述する前提で，それらをどのようにテンプレートによって順次隠蔽していくかという点である . 採

用した STL のモデルでは ,PE 構造やタスク構造を容易に記述可能で ,さらに C++ ユーザの学習コストが低いといったメリットがある .

第 4 章

ライブラリの実装

この章ではライブラリの各層の詳細について、今回おこなった実装について順に述べる。下の層から順に、4.1 節にて PE クラスについて、4.2 節にて PE アロケータ、4.3 にて PE コンテナ、4.4 にてタスク操作アルゴリズムを、そして最後に 4.5 にて、PDPTL を用いた高生産性パラダイムについて述べる。

4.1 PE クラス

X10 における Place にあたる抽象的な計算資源として `processing_element` クラスが抽象クラスとして定義されている。X10 等の言語と大きくことなるのは、PDPTL ではこの `processing_element` の操作をおこなうことも可能ではあるものの、そこから派生した具体的な PE クラスを操作を推奨する。これは双対関係にある STL においては、変数を抽象的な `val, var` といった抽象クラスで操作するのではなく、具体的な特性を持った型やクラスを明示的に指示するタイプセーフな言語であることに由来する。型を指示することによって生産性は落ちてしまうが、型を後述の `template` クラスによって順次隠蔽していくことによって生産性の向上を狙う。さらに C++11 では `auto` を使った自動型推論が利用可能になり、さらなる生産性向上が期待できる。

4.1.1 `talloc` によるタスク割り当て

データに対する値の操作、と対比して PDPTL では PE に対するタスク割り当てを行う。これは `talloc` メンバ関数を通しておこなわれる。例えば `src.1` 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。`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++11 からのサポートになるが、C++98/03 の範囲内のマクロ関数によって可変長引数テンプレートを実現する手法が `boost`⁽²¹⁾ ライブラリなどで用いられており、実装はその手法を用いた。

`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り、PE を拡張する場合 `processing_element` クラスを継承してそのファンクタ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。`talloc` 関数を直接オーバーライドできないのは、C++ がテンプレートメンバ関数の仮想関数化ができないことに起因しており、このように間接的に処理する必要がある。タスク

ファンクタの処理例として、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プールしているスレッドに渡してタスクを実行する。一方 tcp_pe や 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ル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・デシリアルライズした後に実行される。ほかに呼び出し元のスレッドがそのまま逐次的に実行する self_pe や、なにも実行しない dummy_pe、スレッドではなく軽量スレッドに処理を割り当てる fiber_pe、標準ファイル入出力を通してタスクを受け渡しする file_pe などを実装した。

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、引数がシリアル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。シリアルライズには src.7 のような記述が必要で、遠隔呼び出しの通知は src.8 のように記述する。

コード 7: :

Serialization

```

1 class user_class_t{ int a, b; };
2 namespace tpdpl{ // tpdpl は PDPTL の開発名 .
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 }
```

コード 8: :

Preparing for Remote Procedure Call

```

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

4.1.2 talloc 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では、メンバ関数を実行するインスタンスの指定や、戻り値を格納する変数の指定ができない。これを実現するにはファンクタを一つ仲介する必要があるが PDPTL ではこのような機能を実現するインターフェース

として、メンバ関数を呼び出す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 `rtalloc` を用意している。

コード 9: :

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); の非同期実行
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); の非同期実行
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.9 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にはさきほど述べたとおり、戻り値やインスタンスを適切に処理するファンクタを作り、`talloc` を通じてタスクを割り当てる必要がある。

4.1.3 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、`remote_memput`、`remote_memget` 関数が特殊化されており、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。

コード 10: :

Remote Memory Access

```
1 char buf1[256]= " test - test - test ";
2 char buf2[256];
3 char buf3[256];
4 void test(){
5   mpi_pe pe(1); // rank1 に 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.10 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` はそのまま指定スレッドで `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.10 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.)

4.2 PE アロケータ

PE アロケータは, 4.1 節にて述べた PE クラスの確保をおこなう. 特に確保に関して機能が不要ないのならば標準の `std::allocator` を用いる事もできる. 本稿では遠隔呼び出しである `tcp_pe` と `mpi_pe` にて, PE の確保先を限定するようなアロケータを用意している. 例えば, 複数のノードから接続してきた `tcp_pe` をプールして割り当てに使う `alloc_tcp_pe_from_server` や, 特定のノード上に `tcp_pe` を作る `alloc_tcp_pe_at`, 特定の MPI ランク上に `mpi_pe` を作る `alloc_mpi_pe_at_rank`, MPI の各ランクにラウンドロビンに `mpi_pe` を確保する `alloc_mpi_pe_round_robin` などを実装している.

4.3 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` と順に説明をする. 最後に PE コンテナを用いたリダクション処理について述べる.

4.3.1 単純な PE コンテナ

コード.11 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 (4.3.4 節) . また PE コンテナは STL のコンテナ同様イテレータを用いて連続アクセス可能である . これを用いて PE コンテナを隠蔽し , タスクマッピングなどのタスク操作アルゴリズムを記述可能である .

コード 11: :

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> pec(4);
5     join_set js;
6     for( int i=0; i<4; i++){
7         js += pec[ i]. talloc( add, 1, 2);
8     }
9     js. join_all();
10
11     for( pe_vector< thread_pe>:: iterator i= pec. begin(); i!= pec. end(); i++){
12         i-> talloc( add, 1, 2);
13     }
14     pec. join_all();
15 }
```

4.3.2 PE プールコンテナ

コード.12 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) .

コード 12: :

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 }
```

4.3.3 異種混合コンテナ

コード 13: :

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 }
```

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。

コード.13 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。

4.3.4 PE コンテナとリダクション処理

非同期処理を記述する上でリダクション処理を記述可能であるとコード量を削減可能である。PE コンテナはそのイテレータ自体に `talloc` メンバを持っているが、PE クラスの `talloc` メンバ関数とは戻り値が異なっており、`join_set` を返す。この `join_set` は、処理を受け付けた PE とタスク `id`、戻り値を受け取る変数を保持している。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。

コード 14: :

コンテナによるリデュース処理

```

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     tpdpl:: reducer< int> ret=0;
11     ret = pevec. iat(0). rtalloc( & add, 0, 1 ); // at は通常参照を返すが, iat はイテレータを返す .
12     ret += pevec. iat(1). rtalloc( & add, 1, 2 );
13     ret -= pevec. iat(2). rtalloc( & add, 1, 3 );
14     ret( tpdpl:: reducer_operator:: max) = pevec. iat(3). rtalloc( & add, 3, 1 );
15     ret. jreduce();
16     ret( custom_dump.reduce_ope()) = pevec. iat(4). rtalloc( & add, 3, 4 );
17     ret. jreduce();
18 }

```

コード.14 では，10 行目にて `int` 型変数へのリダクション処理を実行する `reducer` `< int >` 型の変数 `ret` が宣言されている．PE コンテナである `pevec` にタスクを割り当てると，`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．

コード.14 の例では PE コンテナが同時に複数の PE にタスクを割り当てていないのでリダクション処理ではなく `future` のような動作になっている．ただし `jreduce` 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．全体に割り当てる PE コンテナが実装されると，`MPI_Reduce` や `MPI_Gather` のような機能が実装可能となる．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．

4.4 タスク操作アルゴリズム

この節ではタスク操作アルゴリズムの一つである，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．呼び出し方法はコード.15 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．

コード 15: :

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 }
```

4.5 その他高生産性パラダイム

PDPTL のライブラリを用いて更に高いレイヤで高生産なプログラミングを可能とするパラダイムを実装している。同期命令や、スレッドを特定のコアに割り当てる `thread_affinity` , 変数の共有などをテンプレートクラスとして実装している。この節では特に、多くの並列分散処理言語で用意される `for` 文の自動展開と、近年注目されている分散配列について述べる。

4.5.1 `for` 文の自動展開

この節では、`for` 文の自動展開機構について述べる。今回実装した `for` 文の自動展開の書式は複数種類ある。コード.16 に実装した 2 種と、実装中の 1 種を示す。

まず始めに TBB などの並列処理ライブラリでよく用いられるスタイルで、繰り返すレンジを引数にとる外部関数やファンクタ定義し、それを指示する方式である。C++スタイルのプログラミングになれたユーザには使いやすい。

次にマクロ関数とローカル構造体を組み合わせ、自動的に関数内にファンクタを自動生成して非同期呼び出しを行う手法がある。これは既存の `for` 文を拡張した形になり、ファンクタに比べ記述が直感的である。実体は関数なので、`for` 内部で用いたい変数に関してはキャプチャする必要がある。これは `parallel_for` マクロ関数の第二引数で指定しており、型を明示的にキャストする必要がある。第三引

数では，ループカウンタ以外の引数を渡す．

最後に C++11 から導入されたラムダ式と型推論系を用いれば，上述の `parallel_for` を簡易化可能であると考えられる．これにより，明示的なキャストや，`for` 文のあとに必要な `parallel_for_end` といったトレイラが必要なくなる．C++11 はまだ仕様が決定して日が浅いので，今後さらにインターフェースの改善が可能になると考えられ，今後の課題となっている．

コード 16:

for 文の自動展開

```
1 void func( int start, int end, int data){ /* ... */}
2
3 // C++ functor stype for loop parallelization
4 template< class PEContainer>
5 void for_style1( PEContainer& pec){
6     int data = 1;
7     for_even( pec ). talloc( & func, 1, 100, data );
8 }
9
10 // c++98/03 style for loop parallelization
11 template< class PEContainer>
12 void for_style2( PEContainer& pec){
13     int data = 1;
14     parallel_for( ( int i=0; i<100; i++), (( int) i, ( int) data), data ){
15         /* ... */
16     }
17     parallel_for_end
18 }
19
20 // c++11 style for loop parallelization
21 template< class PEContainer>
22 void for_style1( PEContainer& pec){
23     int data = 1;
24     parallel_for( ( int i=0; i<200; i++), ( i, data), data){
25         /* ... */
26     }
27 }
```

4.5.2 分散配列

この節では，複数の計算機資源に対する配列の分散割り当てについて述べる．

コード.17において，分散配列は3行目のように `darray` によって定義される．要素の型をテンプレート引数に渡し，コンストラクタにて要素数と割当先の PE 群が格納された PE コンテナを指示する．これによって PE コンテナの要素間で配列を等分割して保持する．

複雑なマッピングをおこなう場合，6 行目のように PE コンテナの要素の順をユーザが調整する．この作業は STL におけるコンテナの操作と全く同じなので C++ プログラマには直感的に操作可能で，また既存のアルゴリズムやコンテナ操

作手法が用いる事が可能なので学習コストが低くなると考えられる . この例では, pec[0] に 500000 個 , pec[1] に 500000 個, ... , pec[4] に 500000 個と割り当てた後 , また pec[0] に戻って 500000 個 , と割り当てていくことになる .

コード 17: :

分散配列の擬似コード

```
1  template< class PEContainer>
2  void test1( PEContainer& pec){
3      darray< int> data(10000000, pec);
4  }
5
6  void test2(){
7      pe_vector< any_pe> dist;
8      thread_pp pec(5);
9
10     for( int j=0; j<2; j++){
11         for( int i=0; i< pec. size(); i++){
12             dist. push_back( pec[ i] );
13         }
14     }
15     darray< int> data(10000000, dist);
16 }
```

第 5 章

ライブラリの評価

この章では、実装したライブラリの評価を行う。まず評価の前に 5.1 節にて STL と PDPTL との一致性を比較し、設計したモデルの検証をおこなう。次に PDPTL の基礎部分である PE クラスを直接用いた段階での性能を既存の言語と比較する。PE クラスの操作での差を確認した後、より高い層である PE コンテナを用いたパラダイムの評価をおこなう。まず分散共有メモリへの対応策であった分散配列に対応した言語と分散配列について比較する。次にラージスケール環境への対応策であった処理・資源の管理手法のある言語と for 文の自動展開について比較する。最後にこれまで難しかった異種混合環境への対応が PDPTL 可能であることを示す。

5.1 PDPTL のモデルの評価

この節では、設計・実装したライブラリのモデルの検証をおこなう。設計ではライブラリを STL と同様の階層構造にすることで既存の C++ プログラムの学習コスト低減を狙っていた。これが満たされているかについて以下の三種類の検証をおこなう。

- STL と PDPTL の双対性の評価
- STL と PDPTL の記述手法の一致性の評価
- STL と PDPTL の仕様の一致性の評価

以降の節で順に述べる。

5.1.1 STL と PDPTL の双対性の評価

設計では、計算資源を管理する手法を STL のメモリ資源の管理方法と双対性をもって設計した。これがどの程度満たせているかを 3.1.1 節 (データ構造とデータ割り当て) と、3.1.2 節 (PE 構造と PE 割り当て) における記述を比較することでおこなう。各記述の差分評価については subversion 1.6.11 に用いられる svn diff の差分アルゴリズムを用いた (図 5.1)。

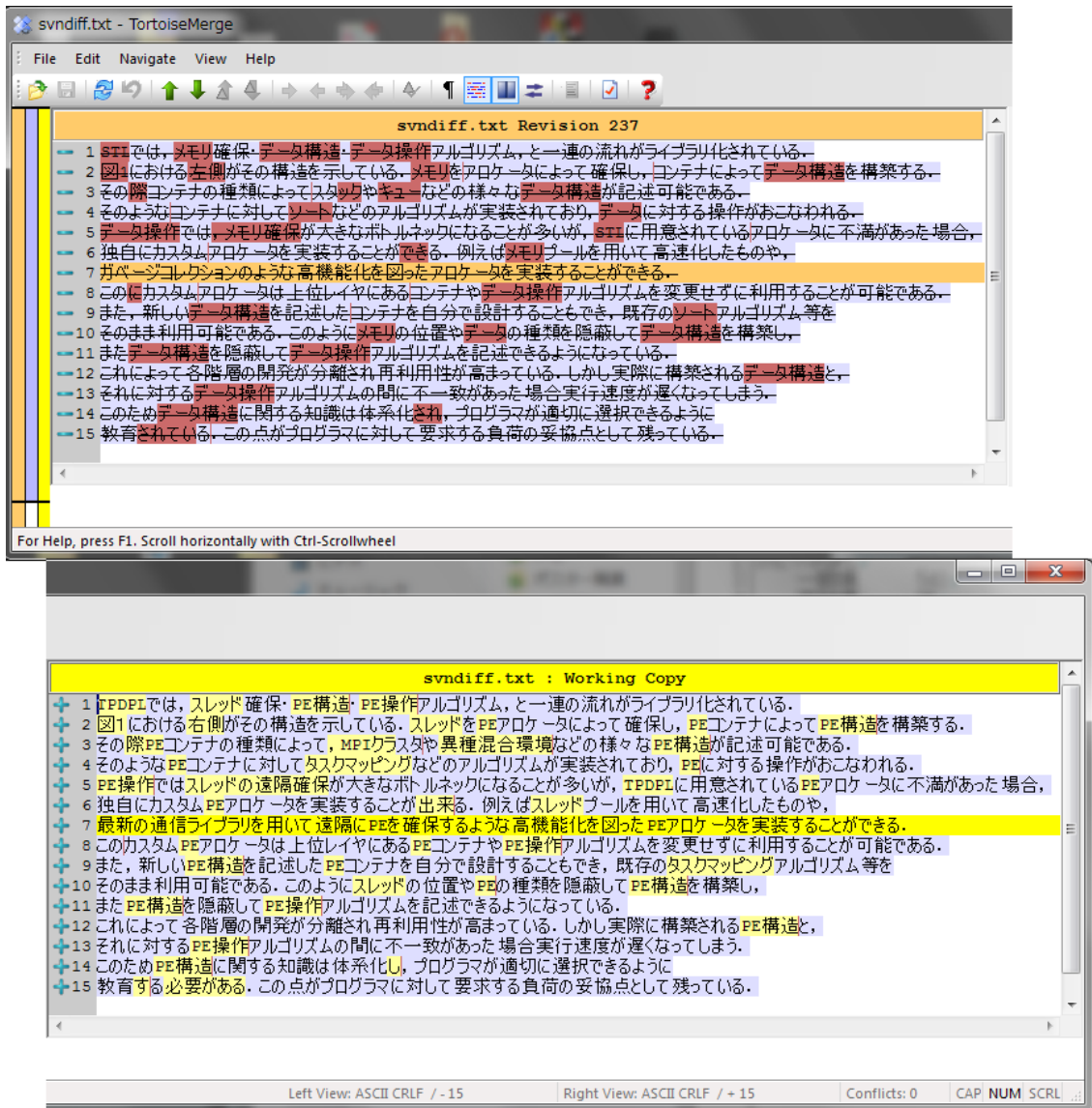


図 5.1: svn diff での比較結果

STL の記述から PDPTL の記述へ変更する時に、記述の 78% の部分に変更が無く、記述に変更があった部分でも 14% の部分が対比表現、7% の部分が例示表現になっており、高い双対性が確認できる。対比表現としては表 5.1 にあるような表現が用いられている。このように、設計の段階では STL との一致性が高く、データと PE の双対性を念頭に持つことで既存の C++ ユーザの学習コストが低くなると考えられる。

表 5.1: 対となる表現例

STL	PDPTL
メモリ確保	スレッド確保
データ構造	PE 構造
左側	右側
アロケータ	PE アロケータ
コンテナ	PE コンテナ
データ操作アルゴリズム	PE 操作アルゴリズム

5.1.2 STL と PDPTL の記述手法の一致性の評価

次に実装したライブラリの一致性であるが、アロケータ、コンテナ、アルゴリズム等の利用方法・標記方法を EBNF にて記述したものを比較する。この記述は C++98 の最終ドラフトである ISO/IEC FDIS 14882:1998(E)⁽²²⁾ を参考にして記述した。以下順に STL, PDPTL と順に記述する。

【STL での記述方法】、

・基本項

identifier :

ISO/IEC FDIS 14882:1998(E) 2.10 節参照

関数やテンプレートの名前など

type-id :

ISO/IEC FDIS 14882:1998(E) 8.1 節参照

int, double などの型やユーザ定義クラスなど

・各層の名前 .

allocator-name :

identifier

container-name :

identifier

iterator-name :

identifier

・イテレータカテゴリ

category-id :

input_iterator_tag | output_iterator_tag |
forward_iterator_tag | bidirectional_iterator_tag |
random_access_iterator_tag

これらは構造体の型であり、イテレータのアクセスの種類を表す。

・各層のテンプレートクラスの記述方法

allocator-id :
 allocator-name < type-id >

iterator-id :
 iterator-name < category-id, type-id, distance-id, pointer-id, reference-id >

container-id :
 container-name < type-id, allocator-name < type-id > >
 コンテナでは下層のアロケータを指定する .
 コンテナは連想コンテナといったコンテナの種類によってはテンプレート引数の書式が変わる .

・アルゴリズムの例

```
template< class InputIterator, class T>  
InputIterator find(InputIterator first, InputIterator last, const T& value);  
アルゴリズムではコンテナの種類によらず ,  
イテレータ操作によって所望する操作をおこなうことで下層の隠蔽がされる .  
この例では , value に一致する最初のイテレータを返される .
```

【PDPTL での記述方法】

・基本項

pe-type-id :
 processing_element 型を継承したクラス
 self_pe, dummy_pe, thread_pe, fiber_pe,
 mpi_pe, tcp_pe, file_pe などの型

・各層の名前

pe-allocator-name :
 identifier

pe-container-name :
 identifier

pe-iterator-name :
 identifier

・イテレータカテゴリ

pe-category-id :
 pe_iterator_tag |
 pe_forward_iterator_tag |
 pe_random_access_iterator_tag |
 pe_master_worker_access_iterator_tag
 PE イテレータの種類をあらわすのに用いられる .

pe-allocator-id :
 pe-allocator-name < pe-type-id >

pe-iterator-id :
 pe-iterator-name < pe-category-id, pe-type-id, distance-id, pointer-id, reference-id >

```
container-id :  
    container-name < pe-type-id, allocator-name < pe-type-id > >  
    PE コンテナでは下層の PE アロケータを指定する .
```

・アルゴリズムの例

```
template< class InputIterator>  
    InputIterator if_free(InputIterator first, InputIterator last);  
    アルゴリズムでは PE コンテナの種類によらずに ,  
    PE イテレータ操作によって所望する操作をおこなうことで下層の隠蔽がされる .  
    この例では , タスクが割り当てられていない最初の PE が返される .
```

このように , アロケータ・イテレータ・コンテナに関しては全く同じ記述方法となっている . アルゴリズムに関しても , イテレータを通じて各 PE にアクセスするという手続きは一致しており直感的である . 設計で提示した STL との双対性を理解した C++ ユーザならば C++ STL を利用するのと同じ感覚でそのまま並列分散処理に関する記述をおこなうことが可能であると考えられる .

5.1.3 STL と PDPTL の仕様の一致性の評価

最後に実装した PE アロケータ・PE コンテナ・PE イテレータが C++ 標準のアロケータ・コンテナ・イテレータが要求するインターフェースを満たしているか比較をする . STL のアロケータに関しては先ほどの FDIS の 20.1.5 節 , コンテナに関しては 23.1 節 , イテレータに関しては 24.1 節にインターフェースの要求が記述されている . 今回実装した `pe_vector` や `pe_iterator`, `pe_allocator` がこれらのインターフェース要求を満たしていることをコード 18,19,20 を用いて確認した . 今回実装したコンポーネントについては正しく C++ 標準の要求を満たしていることを確認できた .

コード 18: :

PE アロケータのインターフェース評価

```
1  template< class X>
2  struct test_allocator_basic
3  {
4      static void test(){
5          X a1, a2;
6          X::value_type v;
7          X::size_type st;
8          X::difference_type d;
9          X::pointer p;
10         X::const_pointer cp= p;
11         X::reference vr= v;
12         X::const_reference cr= vr;
13         X::rebind< tpdpl:: thread_pe>:: other oa;
14         X::rebind< tpdpl:: thread_pe>:: other:: const_pointer u;
15         a1.address( vr);
16         a1.address( cr);
17         p = a1.allocate(1);
18         p = a1.allocate(1, u);
19         a1.deallocate( p,1);
20         a1.max_size();
21         a1 == a2;
22         a1 != a2;
23         X();
24         X c( a1);
25         a1.construct( p, v);
26         a1.destroy( p);
27     }
28 };
```

コード 19: :

PE コンテナのインターフェース評価

```
1  template< class X>
2  struct test_container_basic
3  {
4      static void test(){
5          X a, b;
6          X& r= b;
7          X::value_type v;
8          X::reference vr= v;
9          X::const_reference cr= v;
10         X::iterator i;
11         X::const_iterator ci;
12         X::difference_type dy;
13         a = X();
14         b = X( a);
15         X u( a);
16         a. begin();
17         a. end();
18         a== b;
19         a!= b;
20         a. swap( b);
21         r= a;
22         a. size();
23         a. max_size();
24         a. empty();
25         a< b;
26         a> b;
27         a<= b;
28         a>= b;
29         (& a)->~ X();
30     }
31 };
```

コード 20: :

PE イテレータのインターフェース評価

```
1  template< class X>
2  struct test_iterator_basic
3  {
4      static void test(){
5          X a, b;
6          X& r= b;
7          X u( a);
8          u = a;
9          a== b;
10         a!= b;
11         * a;
12         a-> talloc( pe_vector:: add, 0, 1);
13         ++ r;
14         r++;
15         * r++;
16     }
17 };
```

5.1.4 設計モデル評価のまとめ

以上 3 項の評価によって、設計した PDPTL のモデルが STL と高い双対性を持ち、同一の記述方法でプログラム可能で、その実装が C++ 標準を満たしているこ

とが確認した．これによって C++ プログラムへの学習コストの低減が可能であると考えられる．

5.2 PE クラスの評価

この節ではライブラリの基礎的な特性を評価する．PE クラスの性能評価，PE クラスのスケール実験，PE クラスを用いた実装でのライン数の評価について述べる．これによって最下層の段階での生産性と，実効性能を評価する．

5.2.1 ベンチマークによる基礎性能の評価

今回実装した PE によるタスク割り当てや，遠隔メモリアクセス等の基本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)⁽²³⁾ を用いた．NPB はいくつかのベンチマークがセットになっておりコンピュータの能力評価に利用される．各ベンチマークは多くが Fortran で記述され，一部 C 言語によって記述されている．また，新規に並列処理言語が登場したときに移植されて性能の比較に用いられている．今回は比較対象として PGAS を用いた並列処理言語である UPC を用いた．UPC は C 言語を拡張した言語で UPC コンパイラによって C 言語コードが出力される．バックエンドには OpenMP と MPI をコンパイル段階で選択可能である．テスト環境は情報基盤センターにある HA8000 を用いた．CPU は 4 コアの AMD Opteron 8356 2.3GHz で 1 ノードに 4CPU 搭載され，主記憶容量は 32GB で NUMA 構造である．OS は RedHat Enterprise Linux 5，コンパイラは gcc version 4.1.2，UPC のバージョンは berkeley_upc-2.12.0，MPI はバージョン 1.2 の MPICH-MX である．NPB にあるベンチマークの内，The Embarrassingly Parallel (EP) と Integer Sort (IS) について，今回開発したライブラリを用い C++ コードに移植した．

EP Benchmark

EP は擬似乱数の生成プログラムである．乗算合同法により一様乱数と正規乱数を生成する．今回用いたの CLASS は A(M=28) であり，131072 個の乱数を生成する．このベンチマークの特徴としてノード間での通信がなく非常に並列化し

やすい．図.5.2 に OpenMP をバックエンドに利用した UPC と `thread_pe` による EP の実行結果を，表.5.2 に MPI をバックエンドに指定した UPC での EP の実行結果を，table.5.3 に `mpi_pe` を用いた EP の実行結果を示す．全特性にてほとんど特性差がないことが確認できる．EP ベンチマークでは並列性が高いので，実行時間の差は呼び出し部に集中すると考えられる．今回差はほとんどみられなかったので，`talloc` を呼び出してからスレッドプールで目的の関数が呼び出されるまでの時間がこのアプリケーションでは実用の範囲内であることが確認できた．同様に `mpi_pe` における遠隔呼び出しの場合でも実行時間に大きな差がなく，遠隔呼び出しのコストは実用の範囲内である．

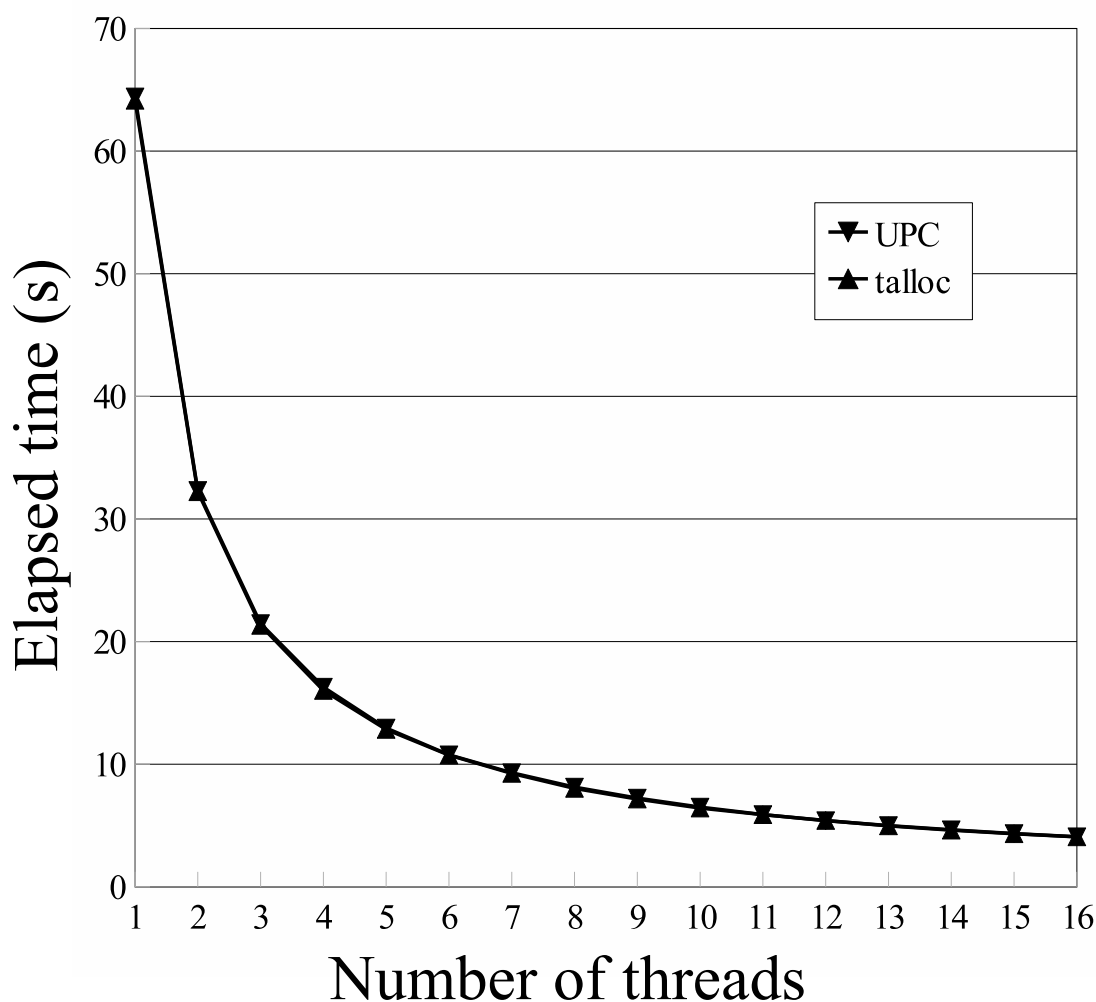


図 5.2: UPC(OpenMP) と `talloc(thread_pe)` での EP の結果

表 5.2: UPC(MPI) での EP の結果 (秒)

		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

表 5.3: talloc(mpi_pe) での EP の結果 (秒)

		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

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。このため並列性は低く、通信が大きなボトルネックとなっている。PDPTL での実装は遠隔メモリ書き込みが用いられている。図.5.3 に OpenMP をバックエンドに利用した UPC と thread_pe による IS の実行結果を、表.5.4 に MPI をバックエンドに指定した UPC での IS の実行結果を、表.5.5 に mpi_pe を用いた IS の実行結果を示す。まず図.5.3 では共有メモリ環境でのテストになるが、ほぼ同等の性能が得られた。特徴として、UPC はスレッド数が少ない場合、PGAS の処理がコストとなるため特性が悪く、逆に talloc ではスレッド数が大きいほど特性が劣化してしまう。talloc の特性劣化はスレッド数が 8 以上で起こっている。これは 2CPU 以上にタスクが割り当てられた場合であるので、CPU 間での通信がボトルネックになっているものと考えられる。スケーラビリティの観点からすると UPC の方が優秀である。

ので、原因を調査し、遠隔メモリ書き込みの性能を向上させることが必要である。次に表.5.4 及び表.5.5 の分散メモリ環境では、UPC(MPI) に比べ、mpi_pe の単一ノード時の特性が悪くなっている。これは mpi_pe 版のベンチマークは通信を前提とし、遠隔メモリ操作を用いて記述しているため、その分劣化しているものと考えられる。そのため複数ノード時には mpi_pe の方が特性が良くなっている。またスレッド数が増えると、UPC(MPI) では制御スレッドが内在するためか処理が終わらなかった。一方 mpi_pe では処理は終わるものの特性の劣化が確認された。これもメモリの遠隔書き込み系がボトルネックになっていることが予想されるが、詳細は現在調査中である。

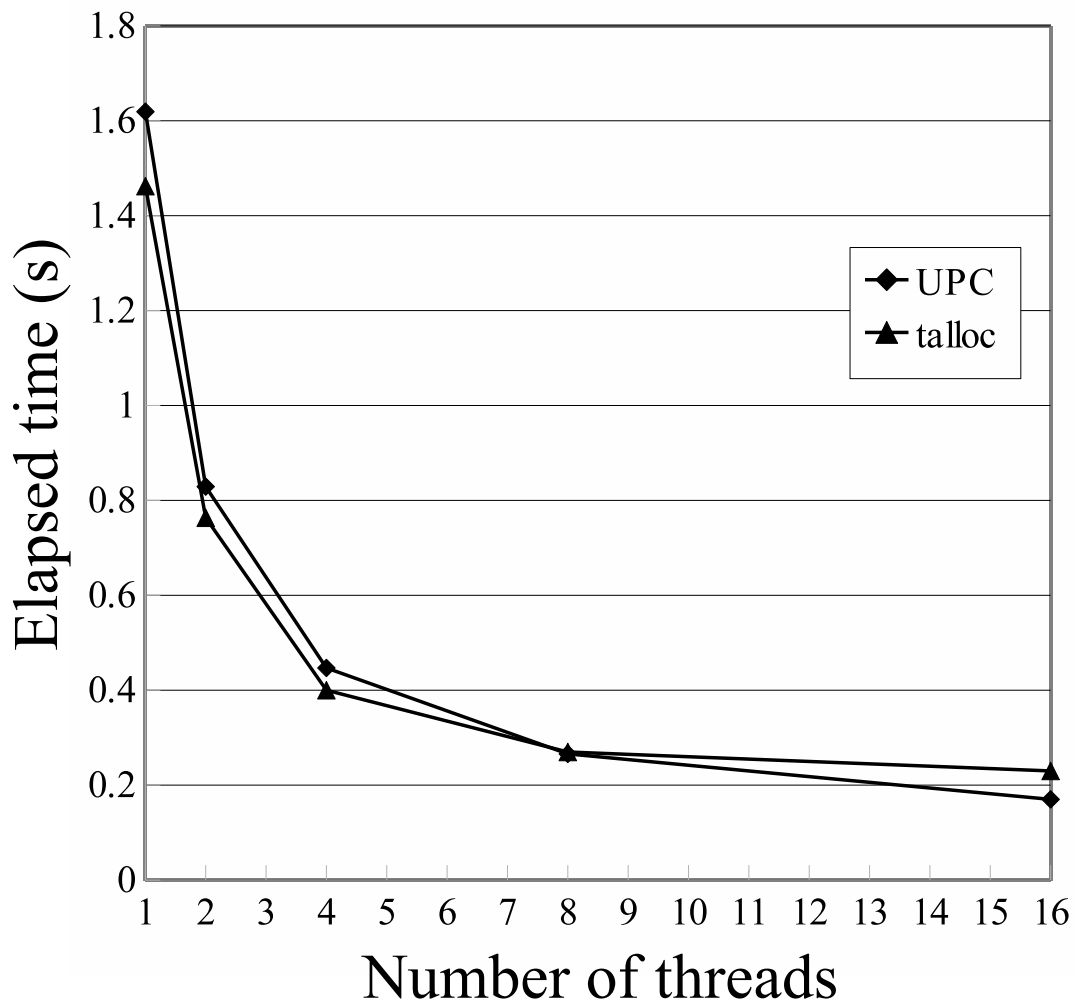


図 5.3: UPC(OpenMP) と talloc(thread_pe) での IS の結果

表 5.4: UPC(MPI) での IS の結果 (秒)

		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 5.5: talloc(mpi_pe) での IS の結果 (秒)

		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

5.2.2 PE クラスのスケール実験

今回実装した tcp_pe 及び mpi_pe によるタスク割り当ての計算ノード数増加に対する性能を評価する．テストプログラムは前節でも用いた EP の class A を用いる．評価環境は StarBED を用いた．ノードは Intel Xeon X5670 2.93GHz 6 コア 2CPU，100 ノードで計 1200 コア利用できる．OS は Debian 6.0.2，コンパイラは gcc4.4.5 となっている．

理論値の見積もり

理想的な EP は PE 数 N が増加するにあたり実行時間 $T(N)$ は短くなる．これは次式によって表される．

$$T(N) = T(1)/N \quad (5.1)$$

しかし実際には割り当てにかかる時間や，通信にかかる時間，戻り値を受け取る時間などが存在する．

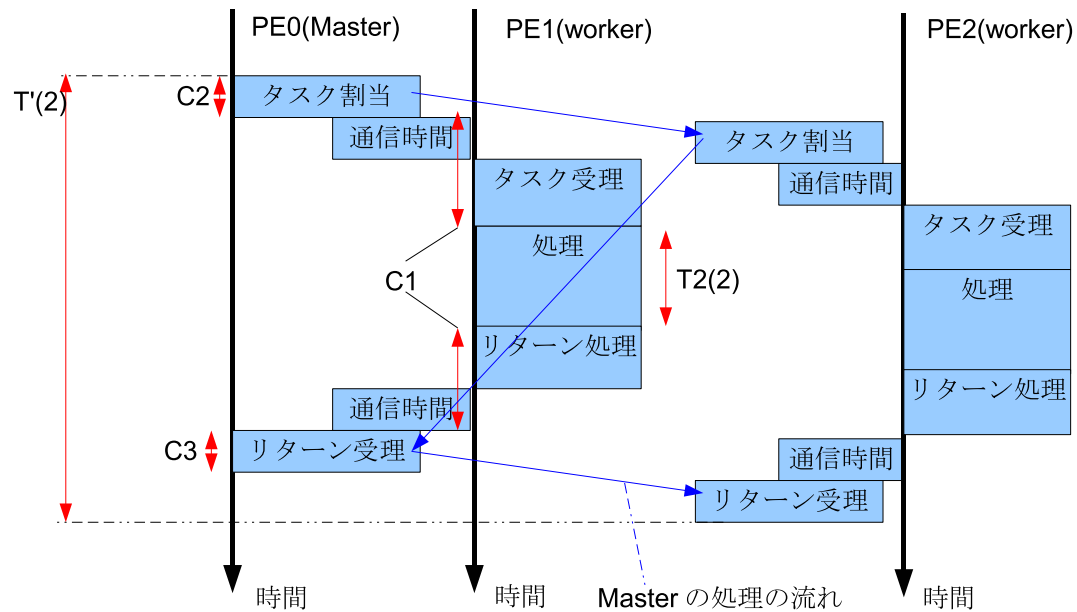


図 5.4: 遠隔割り当て時のオーバーヘッド

図 5.4 は $N=2$ の時の PDPTL での割り当て例になる．割り当てをおこなうマスターは，はじめに PE1 に $C1$ の時間をかけてタスクを割り当て，続いて $C1$ の時間をかけて PE2 にタスクを割り当てる．そして処理が終わり次第帰ってきたリターン受領を $C3$ の時間をかけて処理している．またワーカー側でのタスク受領とリターン処理にかかる時間と通信にかかる時間を合わせた時間を $C1$ とする．ここで， N 個の PE に対して連続的にタスクを割り当てた場合にかかる総合時間 $T'(N)$ は次式のようにになると考えられる．

$$T'(N) = (T(1) - C1 - C2 - C3)/N + C1 + C2 * N + C3 \quad (5.2)$$

各 PE で実際に処理する時間は，一つの PE で実行した時の時間からコストを引いた値を，PE 数で割った時間になる． $T'(N)$ はその値に一回分の $C1$ と $C3$ ，及び N 回分の $C2$ を足した値になると考えられる．表.5.6 に tcp-pe ,mpi-pe における $T(1)$, $C1$, $C2$, $C3$ の測定結果を示す．測定では $C2, C3, T(N)$ 及び， $C1$ の間に位置する各 PE での実行時間 $T2(N)$ を測定し， $C1$ は次式から求めた

$$C1 = T(N) - T2(N) - C2 * N - C3 \quad (5.3)$$

N を 1 から 10 まで変化させ，各値の平均値を求めた

表 5.6: スケール実験におけるパラメータ値

	T(1) [s]	C1 [ms]	C2 [ms]	C3 [ms]
tcp_pe	41.56	0.04	0.01	0.02
mpi_pe	41.56	25.30	0.05	0.03

測定結果

図 5.5 に測定結果を示す．tcp_pe , mpi_pe 共に 1200 コアまで順調にスケールするものの理論値よりもやや劣っている．特に mpi_pe ではコア数が少ない時に大きな性能劣化が生じている．今回の実装では MPI1 をサポートしているが MPI1 では MPI 関数がスレッドセーフではないために通信処理を一つのスレッドで行っている．その結果処理の各所でスレッド間同期が必要になり経過時間が理論値よりも大きく増加しているものと考えられる．今後の実装により更なるコストの削減が課題ではあるが，1200 台程度であれば理論値に比べて 50ms ほどの差で想定通りにスケールすることが確認できた．

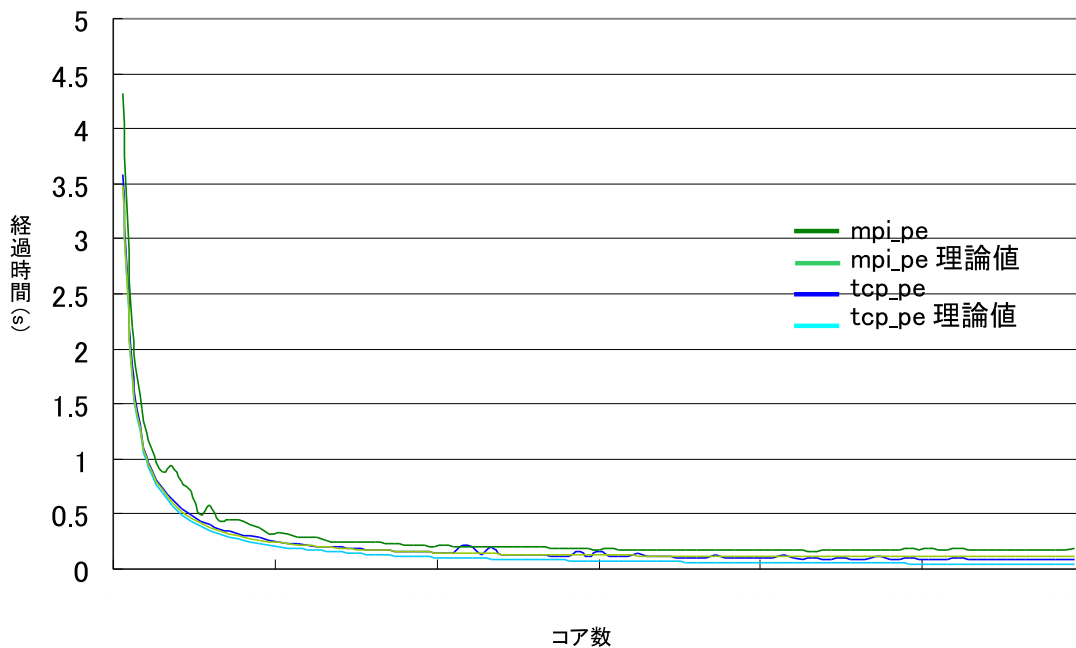


図 5.5: スケール実験結果

5.2.3 ライン数の評価

この節では，PDPTL の PE クラスを直接用いた場合のライン数に関する評価をおこなう．評価したプログラムは，前節でも用いた NPB IS を用いる．NPB の MPI 版と，それから移植した mpi_pe 版の比較，NPB の OpenMP 版とそれから移植した thread_pe 版の比較をおこなう．計測は，インクルード文 (include)，変数・プロトタイプなどのグローバルでの定義 (global)，メインの rank 関数 (rank)，及びその他 (その他関数) と，それぞれ計測した．表 5.7 に結果を示す．

表 5.7: プログラム要素ごとのライン数の評価結果

	MPI	OpenMP	mpi_pe	thread_pe
include	6	5	10	8
global	119	93	131	99
rank	210	155	260	165
その他関数	293	252	370	252
合計	628	505	771	524

表より，mpi_pe は 143 行の増加，thread_pe は 19 行の増加となった．行数の主な増加理由は，mpi_pe 版では全体全通信や，バリア同期，リデュース処理などのパラダイムが未実装だったためその分の行数が増加している．また thread_pe では thread_private な変数の実現のためライン数が増加している．直接 PE を用いるのではなく，上位のコンポーネントを用いたり，よく使われるパラダイムについてはテンプレートとしてまとめることによって，今後ライン数は削減可能であると考えられる．理論的には移植前のコードで用いられているパラダイムが PDPTL 上に実現可能であれば，最終的にライン数はほぼ同等な値まで削減可能である．

5.2.4 基礎性能のまとめ

PE クラスを用いた時点での性能は既存の言語と遜色が無いことが確認できた．今後これを用いた PE コンテナや PE アルゴリズムに関する評価をおこなう．ライン数に関してはこの時点では増加しているが，より上層にてパラダイムを実装することによって削減可能であると考えられ，以後の評価で確認していく．

5.3 分散配列の評価

この章では分散共有メモリへの対応への対応策の一つである分散配列について評価を行う．分散配列に対応した XcalableMP と X10 の二言語と比較を行う．この実験で分散配列を表記するのに必要なライン数と，その実効性能を評価する．

5.3.1 評価環境

評価環境には，二種類の計算機 12 コア 1 台・4 コア 2 台の計 20 コアで構成されている．12 コアのものは，CPU が AMD Opteron 4180 2.6GHz 6 コア 2CPU，OS は Ubuntu10.04LTS．4 コアのものは，CPU は Intel Xeon W3530 2.8GHz で 4 コアで 2 ノード，OS は ubuntu10.04LTS．コンパイラ両者 gcc version 4.4.3, MPI も両者 MPICH2 1.2.1 である．

比較に用いた X10 は x10c++ version 2.2.1 を使い XcalableMP は Omni OpenMP/XcalableMP Compiler の Current Working Copy(omni-3-R400) を用いた．

5.3.2 テストプログラム

実験内容は，配列長 10000000 の int 型配列を計算資源に分散割り当てし，各要素に+1 する操作するのにかかる時間を計測する．この時間の 10000 回の平均を比較する．計算資源は，1 コアから 20 コアまで順に増やし，コア数での変化を観測する．割り当ては 12 コアのノードから始め，13 コア目から 4 コアのノードを投入し，16 コア目から 4 コア 2 台目を用いる．

各言語における分散配列を使用した擬似コードをコード 21,22,23 に示す．

コード 21: :

XcalableMP での分散配列

```
1 # pragma xmp nodes p(*)
2 # pragma xmp template t(0:(10000000-1))
3 # pragma xmp distribute t(block) onto p
4
5 int i;
6 int data[10000000];
7 # pragma xmp align data[i] with t(i)
8 int main( void)
9 {
10     double sum=0;
11     for( int ii=0; ii<10000; ii++){
12         # pragma xmp barrier
13         double s = elapsed_time();
14         # pragma xmp loop on t(i)
15         for( i=0; i<10000000; i++){
16             ++ data[i];
17         }
18         # pragma xmp barrier
19         double e= elapsed_time();
20     }
21     # pragma xmp task on p(1)
22     return 0;
23 }
24 }
```

XcalableMP では 2 行目にてデータやタスクの分散方法を指定しており, 6 行目にて配列の定義, 7 行目にて配列のマッピング指示をしている. また 15 行目にて for 文のマッピングを指示している.

コード 22: :

X10 での分散配列

```
1 public class for_x10{
2     static val D = Dist. makeBlock(1..(10000000));
3     public def this(){
4     }
5     public static def main( args: Array[ String](1)){
6         val data: DistArray[ int] = DistArray. make[ int]( D, 0);
7
8         for( var ii: int=0; ii<10000; ii++){
9             var s: long = System. nanoTime();
10             finish ateach([ i]: Point in D ){
11                 data( i) += 1;
12             }
13             var e: long = System. nanoTime();
14         }
15     }
16 }
```

次に X10 では, 2 行目にてデータやタスクの分散方法を指定しており, 6 行目にて分散配列の定義とマッピングをおこなっている. 10 行目にて for 文の自動分割を行なっている.

コード 23: :

PDPTL での分散配列

```
1 void test1( int s, int e, tpdpl:: darray< int>& da){
2     int* p = & da[ s]- s;
3     for( int i= s; i<= e; i++){
4         ++ p[ i];
5     }
6 }
7 void test2( int s, int e, tpdpl:: darray< int>& da){
8     for( int i= s; i<= e; i++){
9         ++ da[ i];
10    }
11 }
12 template< class PEContainer>
13 void test( PEContainer& pec)
14 {
15     tpdpl:: darray< int> data(10000000, pec);
16     for( int ii=0; ii<10000; ii++){
17         double s = : elapsed_time();
18         tpdpl:: for_even( data). talloc( test1, 0, 10000000-1, data);
19         // tpdpl:: for_even ( data ). talloc ( test2, 0, 10000000-1, data );
20         pec. join_all();
21         double e = elapsed_time();
22     }
23 }
24 }
```

最後に PDPTL では 15 行目にて分散配列を定義している．これにより PE コンテナに含まれる資源に割り当てがおこなわれる．PDPTL では配列へのアクセス方法として，先頭ポインタを取得して直接配列にアクセスする test1 と，darray の添字演算子 ([x]) 越しにアクセスする test2 の 2 種類を用意している．C++STL では vector などの配列では要素の連続性が保証されているため，高速化のために test1 のような手法がよく用いられる．このように高いレイヤーから低いレイヤーへと抽象度を変えたチューニングが既存の C++ と同じようにおこなえる．

5.3.3 評価結果

まず始めに記述性能の考察を行う．この分散配列の定義方法に関してはどの言語も 1 行で記述することが可能でライン数に関しては差がないことが確認できる．

次に図 5.6, 5.7 に実験結果を示す．横軸がコア数で縦軸が経過時間の平均である．

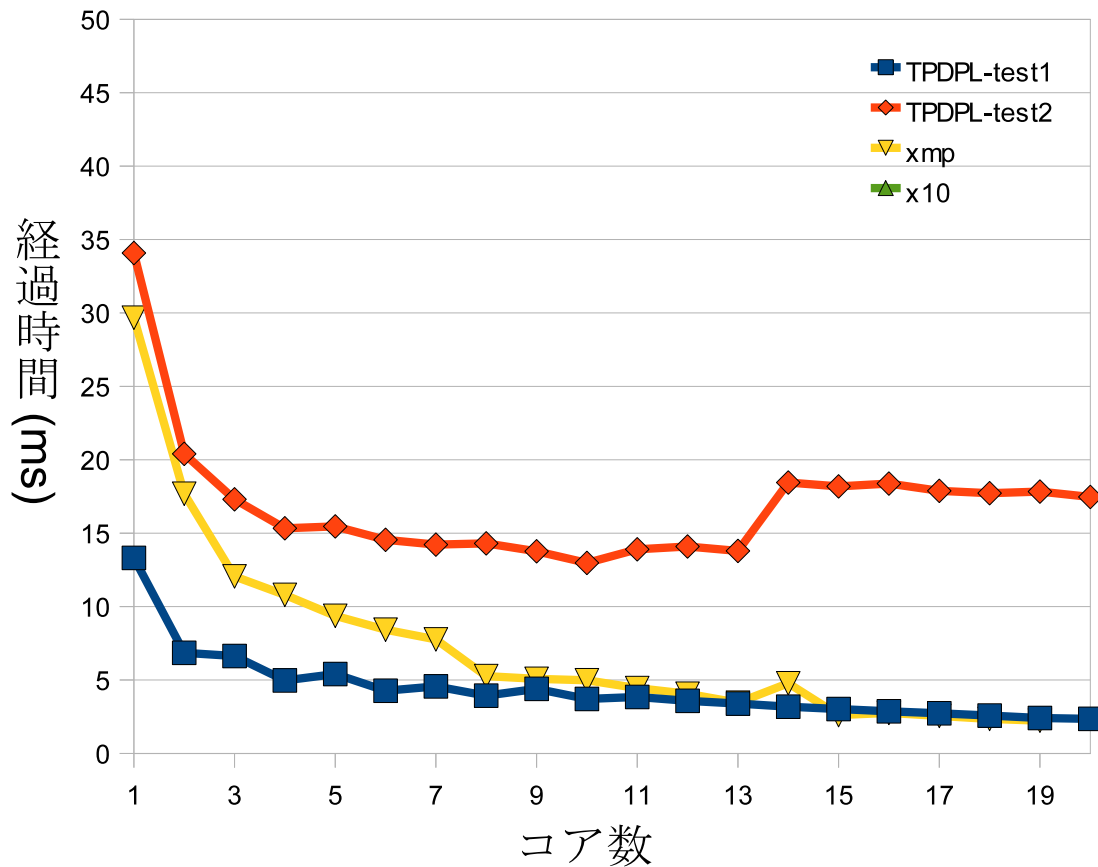


図 5.6: 各言語での分散配列の実行結果 (小レンジ)

PDPTL の test1 による配列に直接アクセスするものが最も速くなった．はじめの 12 コアは共有メモリ上にスレッドが存在しているため，thread_pe で割り当てをおこなう PDPTL が速くなる．XcalableMP の場合は MPI によってプロセスが分割されてしまい，メモリ空間が異なるためその分のオーバーヘッドが発生したものと考えられる．一方でノードが追加される 13 コア目以降，PDPTL では遠隔呼び出しのためのオーバーヘッドが増え，XcalableMP の同期コストよりも経過時間が遅くなる．次に darray のインターフェースを用いた場合は配列にアクセスするインデックスを計算するためオーバーヘッドが増え，XcalableMP よりも遅くなることが確認できる．

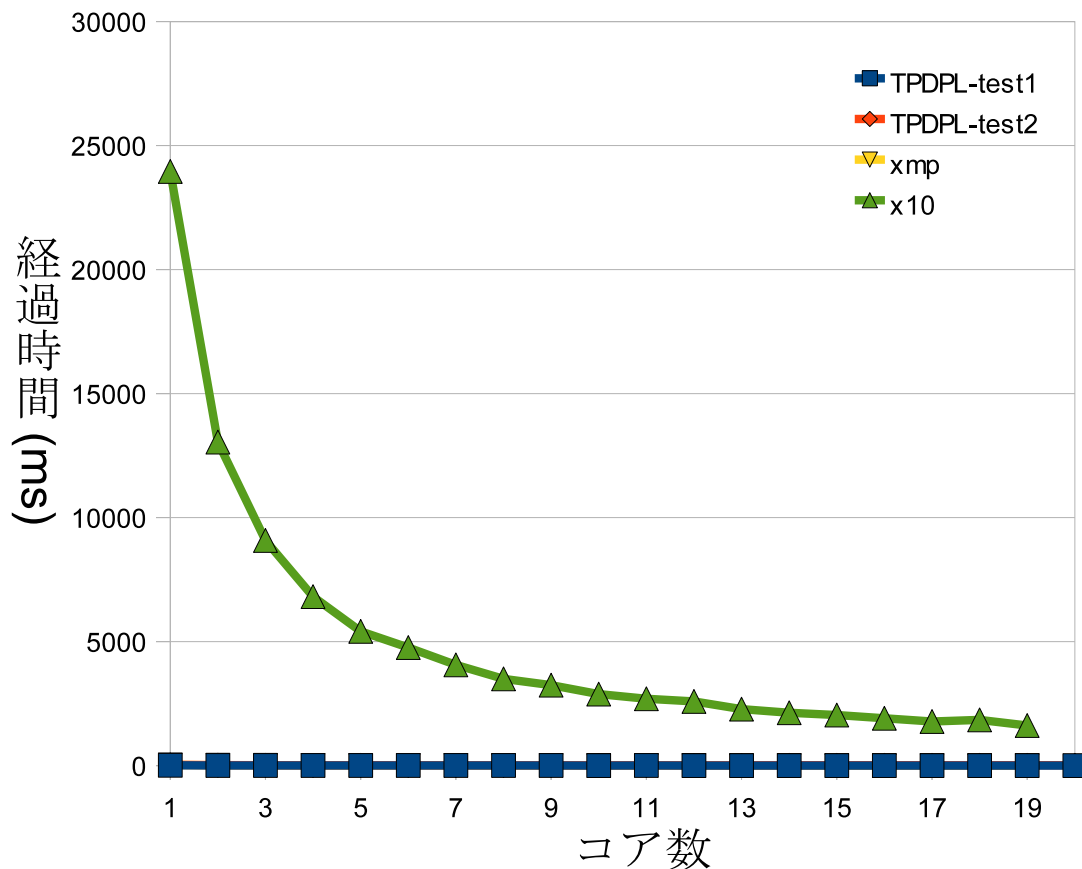


図 5.7: 各言語での分散配列の実行結果 (大レンジ)

最後に X10 に関しては現状の実装の段階ではまだ性能が出ず，他の言語に比べて非常に遅くなってしまった．(今回の実験では，MPICH2 のバグによって 20 コア目における XcalableMP, X10 のデータが取得できなかった．)

今回実装した分散配列 darray について，記述性能・実装性能共に他の言語と同等であることが確認できた．

5.4 処理・資源の管理手法の評価

この章では処理・資源の管理手法の評価を for 文の並列展開を用いて比較し，その記述性能と実行性能を他の言語と比較する．

5.4.1 評価環境

評価環境には、HaaS としてレンタルした計算機群 (StarBED⁽²⁴⁾,⁽²⁵⁾) を用いた比較対象は、タスクスチーリングを用いる X10(2.2.1), Cilk(gcc4.7) と、for 文の展開をサポートする OpenMP(gcc4.7) や、XcalableMP(omni-3-R400), UPC(berkeley_upc-2.12.0) を用いた。Intel Xeon X5670 2.93GHz 6 コア 2CPU で 5 ノードある。OS は Debian 6.0.2 である。

5.4.2 テストプログラム

テストプログラムは、単純な計算を行うものトイプログラムであり、for 文のループインデックスの値によって負荷の大きさが変わる。負荷の大きさは、k 番目の処理が全体の $1/k$ を占める zipf 分布に従うものとする。その要素全体は 100000000000 個あるが、for 文ではその上位 1000 個を実行する。

自動展開ではワークスチーリングやスレッドプール、PE プールなどの資源管理や、static, guided 割り当てなど言語によっていくつかの割り当て手法がある。以下順に、各言語での例に合わせ擬似コードとともに説明する。

コード 24: :

X10 での for 文の展開

```

1  public class for_x10{
2      static val D2 = Dist.makeUnique();
3
4      public static def zipf( var iii: ULong, var cs: ULong, var t: double, val results: DistArray[
5  Double]){
6          var a: Double = 0;
7          for( var i: ULong= iii; i< iii+ cs; i++){
8              var loop: double=10000000000 UL/ t/( i as double);
9              for( var j: ULong=1; j<= loop; j++){
10                 a += ( i+ j)*( i- j)*( i* j)*( i/ j);
11             }
12         }
13         atomic results( here. id) += a;
14     }
15
16     public static def main( args: Array[ String](1)){
17         // zipf init
18         var tt: Double;
19         tt=0;
20         for( var k: ULong=10000000000 UL; k>=1; k--){
21             tt += 1.0/ k;
22         }
23         val t: Double= tt;
24         var iii: ULong=0;
25         for( var cs: ULong=1; cs<=10 U; cs+=1){ // cs : chunk size
26             val cs2: ULong = cs;
27             val results: DistArray[ Double](1) = DistArray. make[ Double]( D, 0);
28             var result: double;
29             // static alloc
30             time = - System.nanoTime();
31             finish ateach([ i]: Point in D ){
32                 for( var ii: ULong= i; ii<=1000 U; ii+= cs2* Place. MAX_PLACES)
33                     zipf( ii+1, cs2, t, results);
34             }
35             result = results.reduce((( x: Double, y: Double) => x+ y),0);
36             time += System.nanoTime();
37
38             // work stealing
39             time = - System.nanoTime();
40             finish for( iii=1; iii<=1000 U; iii+= cs){
41                 val iiiii= iii;
42                 async zipf( iiiii, cs, t, results);
43             }
44             result = results.reduce((( x: Double, y: Double) => x+ y),0);
45             time += System.nanoTime();
46         }
47     }
48 }
49 }

```

コード 24 の X10 では、29 行目からスタティック割り当てによる for 文の展開が記述されている。ateach にて、各計算機上に Activity を割り当てた後に、1 ずつサイクリックに分割された自分の処理範囲を計算して実行する。このようにスタティックでは事前に計画して割り当てる。一方 38 行目からの記述ではワークスティーリングをおこなって動的に for 文を割り当てる。1000 まで毎回 async によって Activity を生み出し、キューに溜まった Activity が他の Place に移動することが期待されるが、ワークスティーリングに関して今回はライン数の評価のみおこない

実行性能の評価は行わない。

コード 25: :

Cilk での for 文の展開

```
1 void zipf( uint64_t ii, uint64_t cs, std::vector< double>& aa, double t)
2 {
3     for( uint64_t i= ii; i< ii+ cs; i++)
4     { // must declare loop counter here
5         uint64_t me = __cilkrts_get_worker_number();
6         double r=0;
7         uint64_t loop = 10000000000 ULL/ i/ t;
8         for( uint64_t j=1; j<= loop; j++){
9             r += ( i+ j)*( i- j)*( i* j)*( i/ j);
10        }
11        aa[ me] += r;
12    }
13 }
14
15 void test()
16 {
17     double a=0;
18     std::vector< double> aa( rank);
19     double t=0;
20
21     // zipf init
22     for( uint64_t i=10000000000 ULL; i>=1; i--){
23         t+= 1.0/ i;
24     }
25
26     // task stealing
27     for( uint64_t cs=1; cs<=10; cs+=1){
28         aa.assign( rank,0);
29         a=0;
30         s = omp_get_wtime();
31         #pragma cilk grainsize = cs
32         cilk_for( uint64_t ii= start; ii<=1000; ii+= cs){
33             load( ii, cs, aa, t);
34         }
35         cilk_sync;
36         for( int i=0; i< rank; i++){
37             a+= aa[ i];
38         }
39         e= omp_get_wtime();
40     }
41 }
```

コード 25 の Cilk では 31 行目から for 文が始まる。cilk_for によって自動的に展開され、31 行目のプリプロセッサディレクティブによってチャンクサイズを指定している。Cilk はワークスチーリングが実装されているので、1000 個の処理がロードバランスされることが期待されるが、ワークスチーリングに関して今回はライン数の評価のみおこない実行性能の評価は行わない。

コード 26: :

XcalableMP での for 文の展開

```
1 void test()
2 {
3     # pragma xmp nodes p(*)
4     # pragma xmp template t3(1:1000)
5     # pragma xmp distribute t3( cyclic) onto p
6     a=0;
7     double t=0;
8     // zipf init
9     for( i=10000000000 ULL; i>=1; i--){
10         t+= 1.0/ i;
11     }
12
13     // xmp ではチャンク指定なし
14     # pragma xmp barrier
15     s = xmp_wtime();
16     # pragma xmp loop ( i) on t3( i) reduction(+: a)
17     for( i=1; i<=(1000); i++){
18         int loop = 10000000000 ULL/ i/ t;
19         double r=0;
20         for( j=1; j<= loop; j++){
21             r += ( i+ j)*( i- j)*( i* j)*( i/ j);
22         }
23         a += r;
24     }
25     # pragma xmp barrier
26     e= xmp_wtime();
27 }
```

コード 26 の XcalableMP ではでは , 16 行目から展開がはじまる . 5 行目にて事前定義した分散方法に従って for 文が展開される . 動的な負荷分散機構などの実装はまだなくスタティックに分割される . また今回はチャンクサイズは指定していない . スタティック分割なので他のダイナミックなものよりは遅くなると考えられる .

コード 27: :

UPC での for 文の展開

```
1 void test()
2 {
3     // init zipf
4     double t=0;
5     for( uint64_t i=10000000000 ULL; i>=1; i--){
6         t+= 1.0/ i;
7     }
8     a=0;
9     upc_barrier;
10    s = omp_get_wtime();
11    pa[ MYTHREAD] = 0;
12    // UPC ではチャンク指定なし
13    upc_forall( i= start; i<=1000; i++; i){
14        double r=0;
15        uint64_t loop = 10000000000 ULL/ i/ t;
16        for( j=1; j<= loop; j++){
17            r += ( i+ j)*( i- j)*( i* j)*( i/ j);
18        }
19        pa[ MYTHREAD] += r;
20    }
21    for( i=0; i< THREADS; i++){
22        a+= pa[ i];
23    }
24    upc_barrier;
25    e= omp_get_wtime();
26 }
```

コード 27 の UPC では , 13 行目から展開がはじまる . `upc_forall` にて自動的に分割される . UPC では分散配列などのデータマッピングに合わせた自動的な for 文の分割をおこなう . ここでは分割されたインデックス `i` に応じて処理が各ノードで行われる . スタティック分割なので他のダイナミックなものよりは遅くなると考えられる .

コード 28: :

OpenMP での for 文の展開

```

1 void test(){
2     // init zipf
3     double t=0;
4     for( i=10000000000 ULL; i>=1; i--){
5         t+= 1.0/ i;
6     }
7     // static alloc
8     for( cs=1; cs<=10; cs+=1){
9         a=0;
10        s = omp_get_wtime();
11        # pragma omp for schedule( static, cs)
12        for( i= start; i<=1000; i++){
13            uint64_t loop = 10000000000 ULL/ i/ t;
14            double r=0;
15            for( j=1; j<= loop; j++){
16                r += ( i+ j)*( i- j)*( i* j)*( i/ j);
17            }
18            # pragma omp atomic
19            a += r;
20        }
21        e= omp_get_wtime();
22    }
23    // dynamic alloc
24    for( cs=1; cs<=10; cs+=1){
25        a=0;
26        s = omp_get_wtime();
27        # pragma omp for schedule( dynamic, cs)
28        for( i= start; i<=1000; i++){
29            uint64_t loop = 10000000000 ULL/ i/ t;
30            double r=0;
31            for( j=1; j<= loop; j++){
32                r += ( i+ j)*( i- j)*( i* j)*( i/ j);
33            }
34            # pragma omp atomic
35            a += r;
36        }
37        e= omp_get_wtime();
38    }
39    // guided alloc
40    for( cs=1; cs<=10; cs+=1){
41        a=0;
42        s = omp_get_wtime();
43        # pragma omp for schedule( guided, cs)
44        for( i=1000; i>= start; i--){
45            uint64_t loop = 10000000000 ULL/ i/ t;
46            double r=0;
47            for( j=1; j<= loop; j++){
48                r += ( i+ j)*( i- j)*( i* j)*( i/ j);
49            }
50            # pragma omp atomic
51            a += r;
52        }
53        e= omp_get_wtime();
54    }
55 }

```

コード 28 の OpenMP ではスレッドプールという資源管理をしている．また for 文の自動展開についてスケジューリング方法がいくつかある．今回は static, dynamic, guided の 3 種を試した．11 行目, 27 行目, 39 行目よりそれぞれ記述され, どれもプリプロッサディレクティブによって並列展開を指示する．static は等分割, dynamic は終了したものから順に仕事が割り当てられ, guided は始め大きなチャ

ンクで割り当てていき，徐々にチャンクサイズが小さくなる割り当て方式である．今回のタスクでは static よりも dynamic, guided が高速に動作すると考えられる．dynamic と guided では，分割数や処理の粒度によって優位性が変動すると考えられる．

コード 29: :

PDPTL での for 文の展開

```

1  double zipf(uint64_t start, uint64_t end, uint64_t N, double t){
2      double a=0;
3      typedef uint64_t counter_type;
4      for(counter_type i= start; i<= end; i++){
5          uint64_t loop = N/ i/ t;
6          for(counter_type j=1; j<= loop; j++){
7              a += ( i+ j)*( i- j)*( i* j)*( i/ j);
8          }
9      }
10     return a;
11 }
12 void test(){
13     // zipf init
14     double t=0;
15     for(uint64_t i=10000000000 ULL; i>=1; i--){
16         t += 1.0/ i;
17     }
18     // static allocation
19     for(uint64_t cs=1; cs<=10; cs+=1){
20         s = elapsed_time();
21         reducer< double> rd=0;
22         rd += for_static( pec, cs). talloc(& zipf, start, 1000, 10000000000 ULL, t);
23         rd.jreduce();
24         e= elapsed_time();
25     }
26
27     // dynamic allocation
28     for(uint64_t cs=1; cs<=10; cs+=1){
29         s = elapsed_time();
30         reducer< double> rd=0;
31         rd += for_dynamic( pec, cs). talloc(& load2, start, 1000, 10000000000 ULL, t);
32         rd.jreduce();
33         e= elapsed_time();
34         printf( " %d ,%5.12 f \ ms , \ %5.12 f \ n ", cs, ( e- s)*1000, rd.get()); fflush( stdout);
35     }
36 }

```

コード 29 の PDPTL では，22 と 31 行目にて展開している．今回はスタティック割り当てとダイナミック割り当てを実装した．今回のタスクは負荷が均一でないで，ダイナミックが高速であると考えられる．

5.4.3 評価結果

まず始めに，for 文の展開に関する記述性能の考察をおこなう．for 文の展開に必要なライン数に関しては，PDPTL では関数定義分の 1 行増えている．他の言語でも XcalableMP, OpenMP ではプリプロセッサディレクティブによって 1 行増加

している．Cilk や UPC では for キーワードを置き換えるだけなので基本的には行数に増加がない．X10 に関しても for を ateach に変えたり,finish-async を追加するだけなので行数に変化はない．これから分かるように，行数に関しては大きな差が見られず，既存の言語群と遜色が無いことが確認できた．

次に実行結果のグラフを示す．Cilk や OpenMP は単一ノード上でしか実行できないため，1 ノードでの実行結果と，5 ノードでの実行結果の 2 種類を示す．

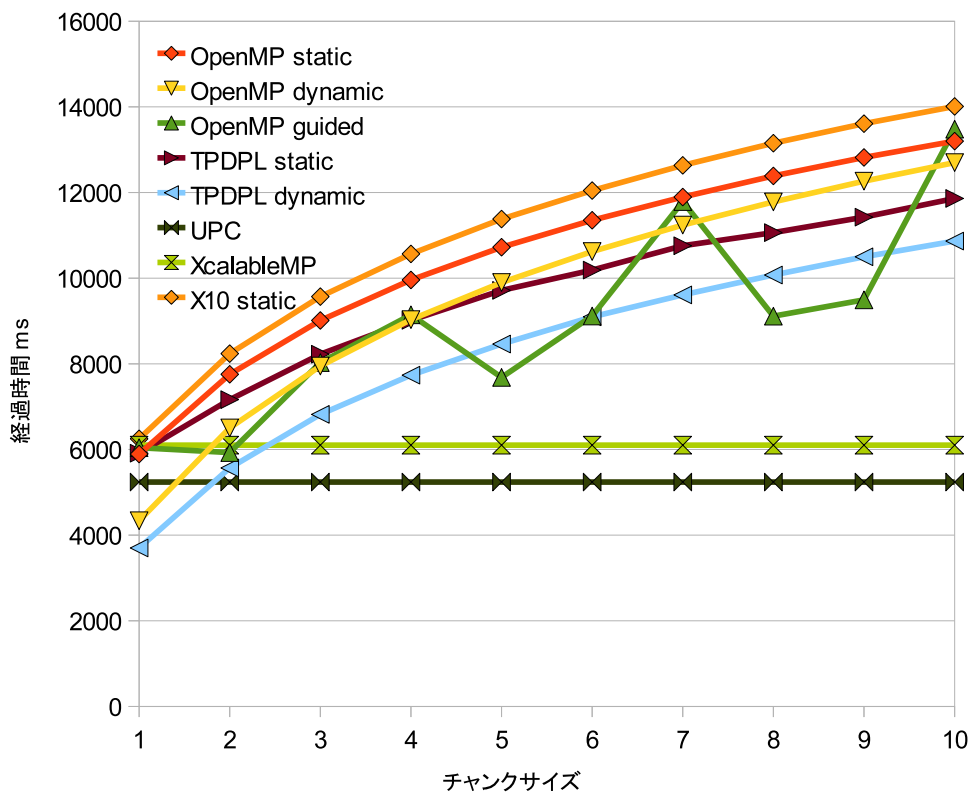


図 5.8: for 文の並列展開 実行結果 (1 ノード)

まず図 5.8 では単一ノードでの実行結果を示す．横軸がチャンクサイズで，縦軸が経過時間になっている．ここでチャンクサイズの指定をおこなわなかった UPC, XcalableMP では直線を引いている．また PDPTL では PE コンテナとして thread_pp を用いている．またスタティックとダイナミックでは，負荷分散がおこなわれるダイナミックがより高速であることが確認できる．また，チャンクサイズが大きいほど負荷分散が困難になるため，経過時間が増加していることが確認できる．一方で OpenMP の guided は予測通りチャンクサイズによって結果が変動している．

PDPTL の実行結果についても既存の言語と実行結果に遜色が無いことが確認できる。

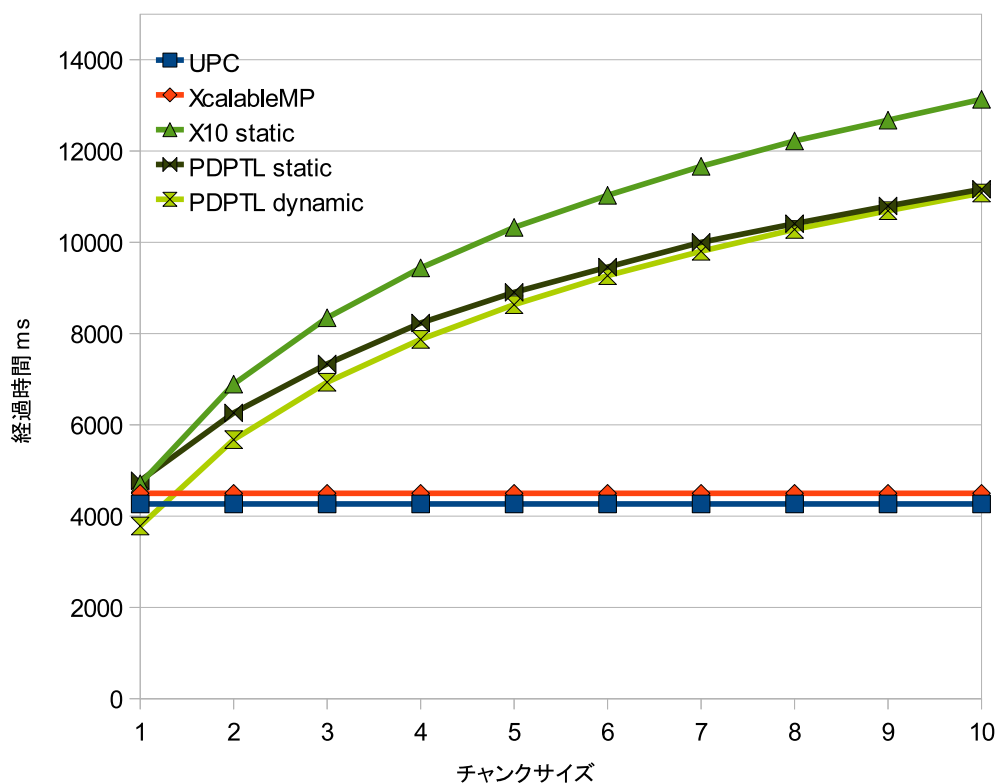


図 5.9: for 文の並列展開 実行結果 (5 ノード)

次に図 5.9 では 5 ノードでの実行結果を示す。PE 数は 60 と 5 倍になるが、実行時間は処理が最も重いものに依存するので 12 コアの図 5.8 と大きな差は出ない。横軸がチャンクサイズで、縦軸が経過時間になっている。ここでチャンクサイズの指定をおこなわなかった UPC, XcalableMP では直線を引いている。また PDPTL では PE コンテナとして tcp-pp を用いている。またスタティックとダイナミックの差異やチャンクサイズによる傾向も単一ノードと同様の結果になった。PDPTL の実行結果についても既存の言語と実行結果に遜色が無いことが確認できた。

5.5 異種混合環境でのタスクマッピング機構の評価

この節では手動でタスクを明示的に割り当てるのが難しい環境で、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、他の言語での構築が実現できないので比較は行わず、実装したライブラリの動作確認と問題の発見を目的とし、今後複雑な計算機環境でのタスクマッピングの検討が可能であることを確認する。

5.5.1 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(StarBED) 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 Super-computer(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 5.10 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ー型となっており、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。）

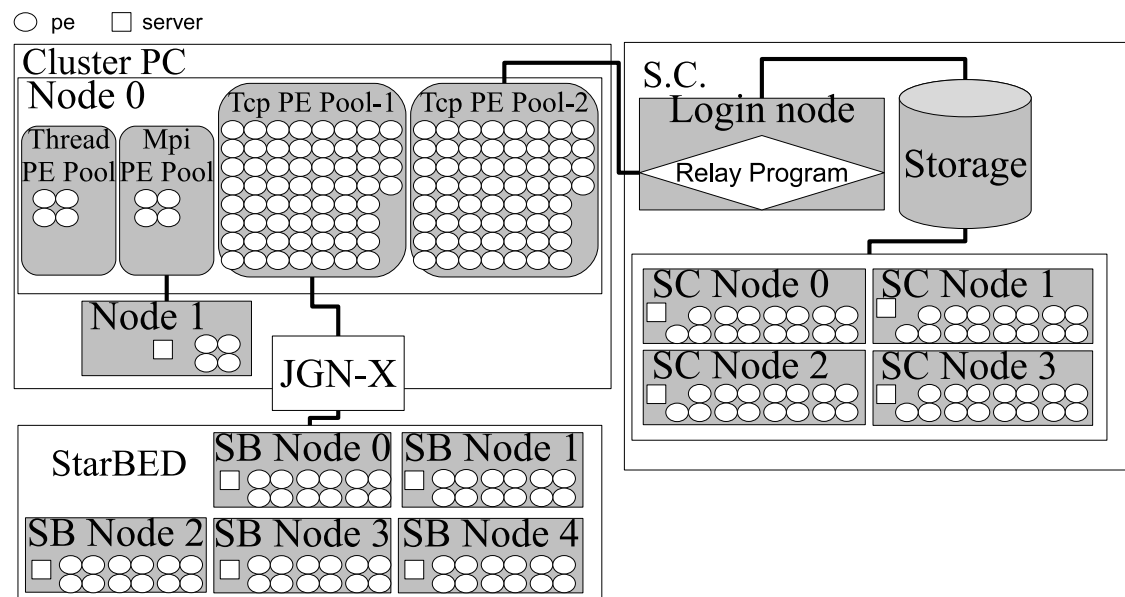


図 5.10: 実験環境上での PE の配置図

マスタノードで用いた PE コンテナは，コード.30 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ル化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⁽²⁶⁾ 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．

コード 30: :

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 (4 PE) 確保  
3 pec. pec0. full_assign();  
4 // mpi.pe で他ノードの物理 CPU 分 (4 PE) 確保  
5 pec. pec1. full_assign();  
6 // S.C. 60 PE 確保  
7 pec. pec2. assign(60);  
8 // StarBED 60 PE 確保  
9 for( int i=0; i<60; i++){  
10     pec. pec2. assign( ip[ i], port[ i]);  
11 }
```

5.5.2 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(コード.31) . 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 100000000 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大きくなるよう for 文のループ回数を設定した.

コード 31: :

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         }  
7     }  
8     return a;  
9 }  
10 uint64_t load_double( int64_t start, int64_t end){  
11     uint64_t a=0;  
12     for( uint32_t i= start; i<= end; i++)  
13         for( uint32_t j=0; j<=1000; j++){  
14             double ii=( double) i, jj=( double) j;  
15             a += (( jj+ ii)*( ii- jj)/( ii* jj)*( ii/ jj);  
16         }  
17     }  
18     return a;  
19 }
```

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 $100000000/128$ 回繰り返し、clock 方式では、各々の PE には $100000000 \times \text{クロック} / (\text{クロックの総和})$ 分だけ、test 方式では、各々 $100000000/\text{時間}/(\text{時間の逆数の総和})$ 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高い負荷分散効果が見込める。

5.5.3 基礎評価・遠隔呼び出し性能

まず始めにマスタノードである研究室の Node0 から、石川県に存在する Star-BED への遠隔呼び出し性能、東京大学情報基盤センターに存在し、WAN-NFS を介した S.C. への遠隔呼び出し性能を確認する。実験では、処理を何も行わない dummy 関数を Node0 から呼び出し、戻って join() する行為を 10 回行う時間を測定した。

図 5.11, 5.12 に結果を示す。各図は横軸がノード当たりのコア数で、縦軸が経過時間になり、パラメータとしてノード数がある。

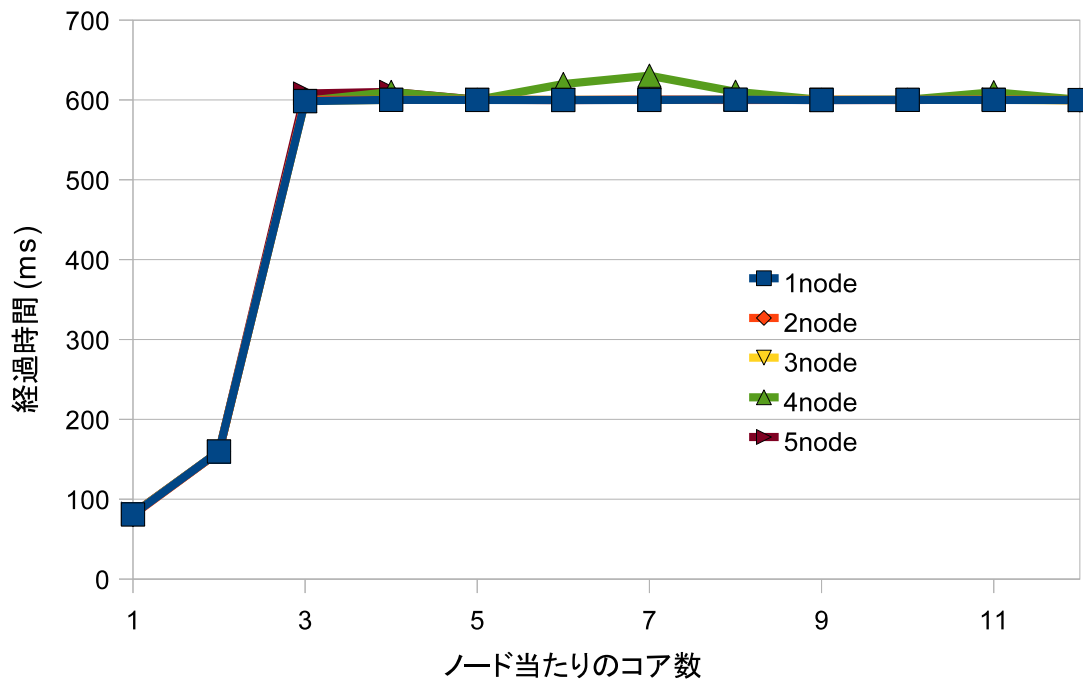


図 5.11: S.B. への遠隔呼び出し性能

S.B. は平均すると 3 ノード以降には安定し、1 回あたり 60ms の時間がかかっている。1,2,3 とスレッド数が増加するに従って線形に経過時間が増加しない理由は、今回の実験で用いた `tcp_pe` の実装では、接続先が同じ `tcp_pe` は全て同じ TCP セッションを通じて通信が行われる。そのため同じ目的地へ数回に分けてパケットが転送されると、ACK の処理などで遅延が発生する。その遅延の間に後続の遠隔呼び出し命令の `Send` が終わるので、以後一定になるものと考えられる。これでは応答性能に不公平が発生してしまう。これを解決するには複数の呼び出し命令を一度に送るか、`tcp_pe` の実装を変更し、セッションを張ることで応答性能が向上することが確認されている。

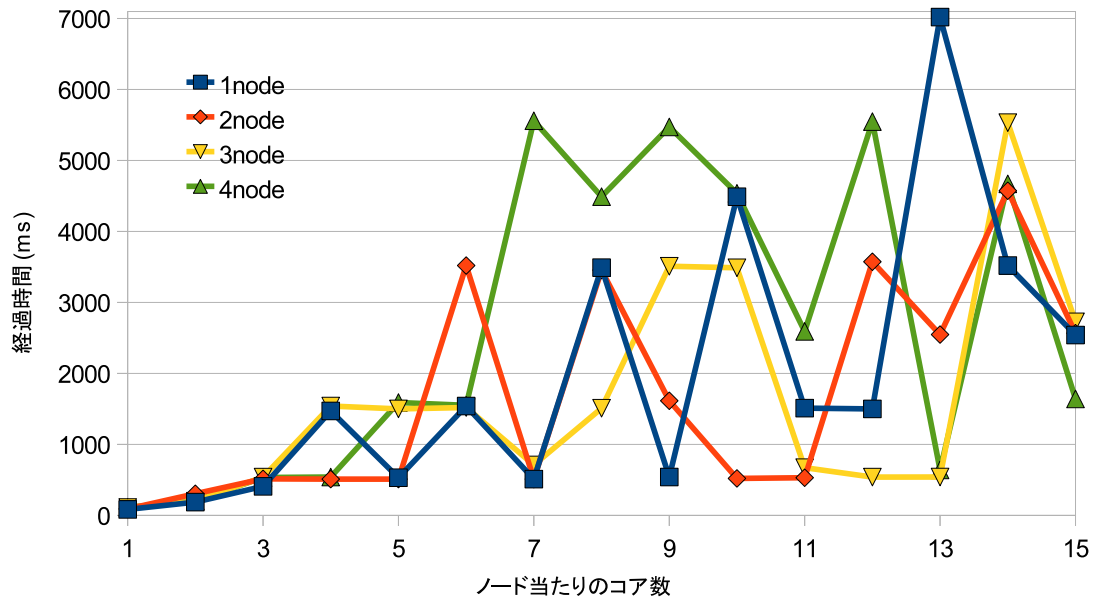


図 5.12: S.C. への遠隔呼び出し性能

次に S.C. への割り当てだが、同様に 1 回 50-60ms ほどの辺りに性能限界面が存在すると思われるが、WAN や NFS からくる外乱が大きいために、一回の割り当て時間は 50-700ms の間で大きくばらついてしまっていることが確認できる。今回の実験で用いたタスクマッピングアルゴリズムは呼び出しにかかる遅延を考えずにタスクをマッピングしているため、このような外乱がどう影響を観測していく。

5.5.4 評価結果

図 5.13 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。

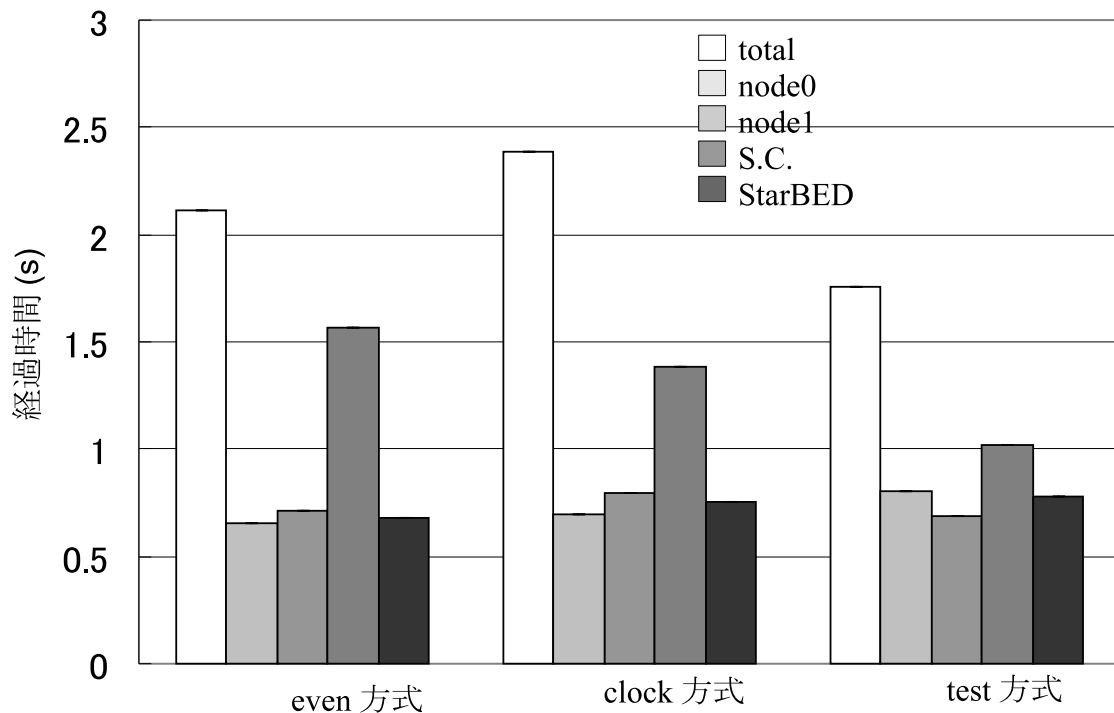


図 5.13: even,clock,test 三方式による load_int マッピングの実行結果

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。

また、今回の実験では遠隔呼び出しに 100m から 1000m ほどの時間がかかる。各図の色のついた棒グラフは、遠隔呼び出しで到着してから、リターンを返すまでの時間の平均なので目安にしかないが、その最悪値から更に環境ごとの通信遅延がたされたものが白い棒の値に近くなる。そのため、クロック取得やテスト実行時間の取得のために2回遠隔呼び出しをする clock,test 方式では、白い棒と色ついた棒の差が平均的に高くなっていることがわかる。さらに高い負荷分散効果を目指すためには、通信遅延の変化に動的に対応可能なマッピングアルゴリズムが必要になると考えられる。

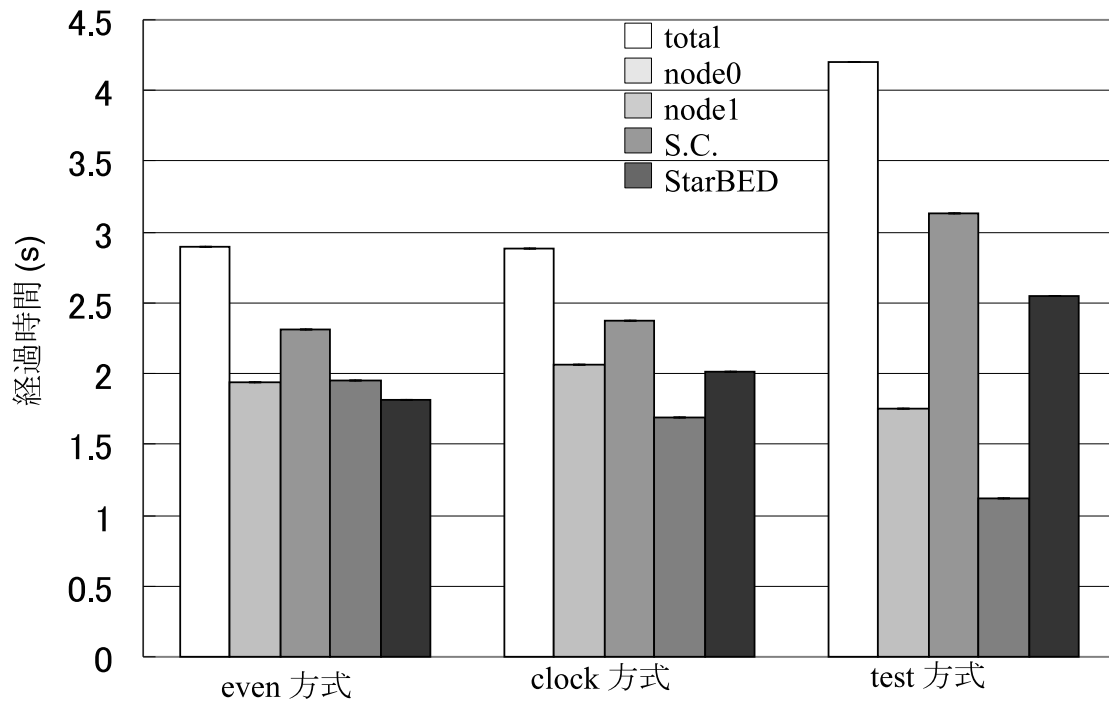


図 5.14: even,clock,test 三方式による load_double マッピングの実行結果

次に図 5.14 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．

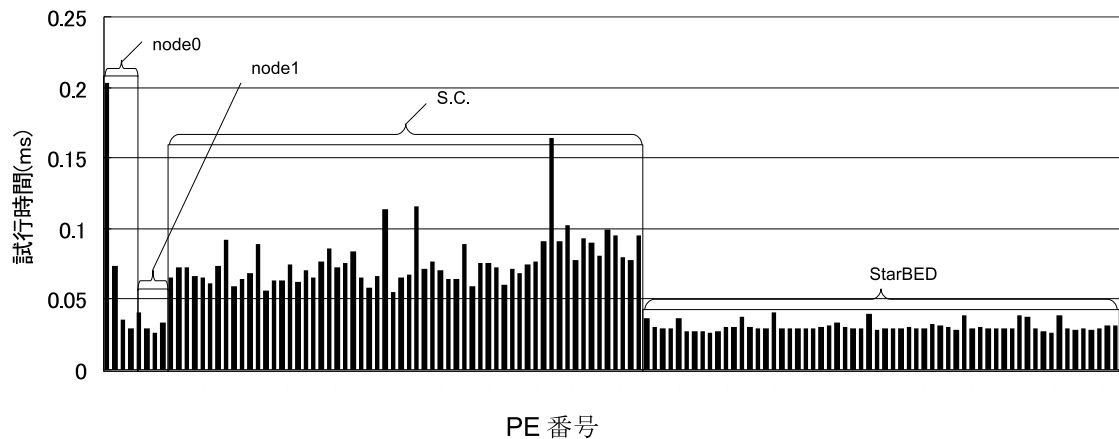


図 5.15: load_double での test 方式での試行時間

図 5.15 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 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。

この実験により、複雑な異種混合環境への自動タスクマッピングについて、PDPTL を用いて今後検討していくことが可能であることが確認できた。また今回実装した単純なマッピングアルゴリズムにより、幾つかの実装上の問題点の解決や通信遅延を考慮した割り当てが必要であることが確認できた。

5.6 評価のまとめ

実験結果より得られた点を以下にまとめる。

今回設計したライブラリのモデルが C++STL と高い一致性があることを確認

した．その実装も C++STL が要求するインターフェースを満たしており，既存の C++プログラムの並列分散処理プログラミングへの移行コストを低減可能であると考えられる．

基礎的な PE の性能は既存の並列分散処理言語とほぼ同等なものとなった．スケール性能に関しては tcp-pe と mpi-pe にて 1200 コアまではほぼ想定どおりにスケールすることが確認できた．ライン数に関しては既存言語に比べて増加してしまった．これには上層での既存のパラダイムの実装が進む事によって低減可能と考えらる．下層の実装性能の確認ができたため，これ以降の上層のコンポーネントの設計や評価の基礎に利用する．

分散メモリ環境への対応策の一つの分散配列については，実効性能・ライン数共に大きな差は見られなかった．また，ラージスケール環境への対応策であった処理・資源の管理手法についても実効性・ライン数共に既存の言語と遜色がないことが確認できた．最後に異種混合環境でのタスクマッピングについては，ライブラリの設計通りに動作することが確認でき，今後複雑な計算機環境でのタスクマッピングの検討が可能であることが確認できた．またその負荷分散性能については，S.C. への割り当てや通信遅延の最適化に問題があることが判明し，今後の検討課題である．

第 6 章

結論

6.1 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．我々は，その中からデータやタスクを明示的に計算機資源に割り当てるパラダイムに注目し，C++特有のテンプレートの階層構造を応用し，C++用のライブラリ (PDPTL) として設計し，一部実装した．

評価によって，ライブラリが C++STL と高い双対性を持つことを確認し，ライブラリの基礎的な性能を確認し，より上層の分散配列の性能や，処理・資源の管理手法の実装が他の言語と遜色が無いことを確認した．最後に異種混合環境での自動チューニング機構について動作確認をおこない，今後様々な高機能マッピングアルゴリズムの検討が可能であることを示した．

これらにより分散共有メモリへの対応，ラージスケール環境への対応，異種混合環境への対応が従来の言語同様に可能であると同時に，データ構造とアルゴリズムのように，並列分散処理で近年注目されているパラダイムや資源管理方法をライブラリとして体系化できることを示した．今後最終的に PDPTL を用いることによって C++並列分散処理アプリケーションの開発効率が向上するものと考えられる．

6.2 課題

今後の課題として，今回実装できなかった様々な既存のパラダイムを実装すること，例えばワークスチーリングなどのタスクや計算機資源の管理手法，共有変数，バリア同期などのパラダイムを実装することなどがある．他にも，高機能なマッピングアルゴリズムや，資源の途中参加や離脱にどのように対応するか，フォールトトレランスなどのエラーハンドリングをどのようにおこなうのか，資源の自動検出機構の設計・実装，デバッグ手法やモニタリング手法の設計などがある．またより下層に近くで，gpu-pe などの PE クラスを実装して適応範囲を広げる必要がある．また評価として，上層を用いたベンチマークを実装し性能や生産性を評価する必要がある．また実際のアプリケーションへの適応や，実際に C++ユーザに公開して学習コストを評価する必要がある．

この他に，現在は実行コードをすべての端末上に配置して実行するスタイルであるが，ソースコードを送って実行するようなスタイルがある．このような機構

を実装するか、もしくは JavaRMI のように VM を用いた遠隔呼び出し系が必要になる。後者であるならば現在検討が進む LLVM(Low Level Virtual Machine) 等を用い、自己反映機構、実行時コード生成機構を組み込んでいくかといった問題がある。

また、現在マッピングアルゴリズムや処理・資源管理手法の検討・検証は主に実際の計算機上でのベンチマークのみであるが、環境や言語の違いから単純比較するのが難しい。理論的評価・計算機によるシミュレーション・実計算機上での実行、と段階的に開発が可能となる必要がある。この先様々な計算機環境のモデル化・プロファイリングを行っていく必要がある。環境のモデル化とともに、プログラムのスライシングなどを用いた実行時間予測アルゴリズムも必要になってくる。

また、今回はメモリ・データとスレッド・PE に関する資源管理手法を双対性を持って設計したが、この他にネットワーク資源の割当問題についても対比的に表現可能であるか検討する必要がある。特に近年では通信にコアがメッシュ状に配置された TILE プロセッサや、6 次元メッシュトラスのような複雑な通信トポロジが複雑なものや、OpenFlow のようにルーティングが制御可能な装置が注目されている。通信資源の確保・管理系に関しても考えた設計が必要となる。これによって、通信・記憶・処理の分野に存在する並列分散性問題の体系化や、一般化に関する検討をおこなっていく。

最後に、HPC 利用だけでなく P2P 分散処理システムや、スマートフォンやセンサを用いたデータ集約と集中処理、ユビキタスコンピューティングなどに適応していくことがあげられる。

謝辞

本論文の作成・研究にあたり貴重な御指導，様々な御助言を賜り、有意義な御討論をしていただいた中山雅哉准教授に心から感謝の意を表します。また，本研究を進める上で様々な角度から多くの助言を与えてくださった若原恭教授，関谷勇司准教授，小川准教授，妙中雄三助教，宮本大輔助教に心より感謝いたします。また出張や研究室運営にて日ごろからお世話になりました若原研究室秘書 吉澤女史、川崎女史に深く感謝いたします。

日々の授業や研究にて活発な意見交換をしてくださった同期の山崎弘太郎氏，新島有信氏を始め，CNL のメンバーの皆様に感謝の意を表します。

研究のために必要不可欠である，高品質な計算機環境を提供してくださった東京大学情報基盤センタースーパーコンピューティング部門の皆様，またとても利便性が高く高性能な計算機資源を提供して下さり，さらに手厚いサポートや研究に対する助言をしてくださった StarBED プロジェクトの皆様，活発な議論やアイデア・コメントをして頂いた WIDE プロジェクトの皆様に深く感謝いたします。

最後に，人生の相談をしてくださった姉に，日々の健康な生活を支えてくださった母親に，コンピュータや電子工作など技術の基礎を幼少の頃より指導してくださった父親に心より感謝いたします。

参考文献

- (1) OpenMP <http://openmp.org/wp/> (2012)
- (2) The Message Passing Interface (MPI) standard <http://www.mcs.anl.gov/research/projects/mpi/> (2012)
- (3) 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)
- (4) 緑川博子, 飯塚肇: ユーザレベル・ソフトウェア分散共有メモリ SMS の設計と実装, 情報処理学会論文誌ハイパフォーマンズコンピューティングシステム, Vol.42, No.SIG9(HPS 3), pp.170-190 (2001, August)
- (5) Tuji, M. Sato, M., “Performance Evaluation of OpenMP and MPI hybrid Programs on a Large Scale Multi-core Multi-socket Cluster, T2K Open Supercomputer, ” Parallel Processing Workshops (ICPPW), p206-213, 1109/ICPPW.2009.73, 2009
- (6) 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
- (7) 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
- (8) Raghavan Raman. M.S. thesis, “Compiler Support for Work-Stealing Parallel Runtime Systems,” Department of Computer Science, Rice University, May 2009.

- (9) 草野和寛, 佐藤茂久, 佐藤三久:Omni OpenMP コンパイラの性能評価, 情報処理学会論文誌, vol.42, No.4, p802-811, 2001 年 4 月
- (10) 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.
- (11) 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.
- (12) Yutaka Ishikawa , Atsushi Hori , Mitsuhsa 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,
- (13) Threading Building Blocks web site, <http://threadingbuildingblocks.org/> (2011)
- (14) 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
- (15) The C++ Standards Committee <http://www.open-std.org/jtc1/sc22/wg21/>
- (16) The Network Simulator - ns-2 <http://www.isi.edu/nsnam/ns/> (2012)
- (17) Qualnet (SCALABLE Network Technologies) <http://www.qualnet.com/content/> (2012)
- (18) Havok Physics <http://www.havok.com/> (2012)
- (19) Bullet Physics Library <http://bulletphysics.org/wordpress/> (2012)
- (20) 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
- (21) Boost C++ Libraries <http://www.boost.org/>

- (22) Programming language [Langages de programmation-C++], ISO/IEC FINAL DRAFT INTERNATIONAL STANDARD 14882:1998(E)
- (23) NAS Parallel Benchmarks, RNR-94-007 <http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf> (2012)
- (24) StarBED Project <http://www.starbed.org/>
- (25) Toshiyuki Miyachi, Ken-ichi Chinen and Yoichi Shinoda:StarBED and SpringOS: Large-scale General Purpose Network Testbed and Supporting Software, Valuetools 2006, Pisa, Italy, ISBN 1-59593-504-5, Oct, 2006.
- (26) JGN-X <http://www.jgn.nict.go.jp/>

発表文献

- (1) 山崎健生、中山雅哉:開発中の並列分散処理ライブラリ (C++) の紹介,
WIDE Project CAMP 2010 Autumn
- (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 Sympo-
sium Series, Vol.2011, p.82
- (3) 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
- (4) 山崎健生, 宮元大輔, 中山雅哉: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, HPCS2012 シンポジウム論文集 IPSJ Symposium Se-
ries, Vol.2012, p.15-22
- (5) 作成物
C++用並列分散処理ライブラリ Template Parallel Distributed Processing Li-
brary
(全 60266 行・本体 30885 行・テスト 3435 行・ベンチマーク 23047 行・解析ツ
ール 2478 行)(公開準備中) [http://www.cnl.k.u-tokyo.ac.jp/~yamasaki_
takeo/TPDPL/Top.html](http://www.cnl.k.u-tokyo.ac.jp/~yamasaki_takeo/TPDPL/Top.html)