ハイパフォーマンス
コンピューティング 65－11
（1997. 3. 6）

MPC++による様々な並列処理プログラミングスタイルのサポートとその性能

二上敦行[†] 松岡聡[††]
石川裕[†††] 佐藤三久[†††]

並列処理の普及のためには，汎用的なハードウェアやソフトウェアをベースとすることが今後望まれ、並列(プログラミング)言語も例外ではない．しかし、同時に並列言語は本来要求される広範な並列プログラミングスタイルへの対応，使いやすさ，実用的な性能を満たすことも必要不可欠である．我々は，既存のオブジェクト指向言語がそのような要求を満たしているかを調べるために、C++で特殊な言語拡張をせず本来の言語機能のみ用いることにより，様々な並列プログラミングスタイルがサポート可能であることを示した．具体的には，C++をテンプレートとインヘリタンスのみで拡張したMPC++をベースに，3つの並列プログラミングスタイルをサポートするテンプレート/クラスライブラリを作成し，ワークステーションクラスタ上で代表的なベンチマークの性能測定を行い、その有効性を検証した。

Supporting Multiple Parallel Programming Styles with MPC++ and their Performance

Atsuyuki Nikami,[†] Satoshi Matsuoka,[††]
Yutaka Ishikawa[†††] and Mitsuhisa Satoh [†††]

For parallel processing to become general, the underlying basis should be advanced commodity technology, and parallel (programming) languages are no exceptions. On the other hand, parallel languages must also satisfy the requirements that inherently stem from parallel processing, such as the support of a wide range of parallel programming styles, ease-of-programming, and high performance. We investigated whether existing object-oriented languages satisfy such requirements or not by showing that C++ can support a wide range of parallel programming styles without special language extensions. More concretely, based on MPC++, which is a parallel dialect of C++ extended using only templates and inheritance, we created a class/template library which support three major kinds of parallel programming styles. We tested its performance with representative benchmark programs of each programming styles on a workstation cluster.

1. はじめに

従来，並列計算/処理のためには，特殊なハードウェアや，特殊な言語，コンパイラ，OSなどが必要とされてきた．特殊なハードウェアは高価であり，特殊なソフトウェアは習熟に時間がかかり，使いこなすのが困難であった．また新技術への対応のために一から設計し直すため，急速な発展する計算機技術への対応が遅れがちであった．

近年，マイクロプロセッサ技術の発展による汎用プロセッサの高性能低価格化と，高速ネットワーク技術の発展により，ワークステーションクラスタが普及してきた．ワークステーションクラスタは，低価格と構成の柔軟さを備えた汎用製品(Commodity)で構成される．

ソフトウェアにおいても，同様な汎用製品が存在するのではないかと考えられる．汎用製品であるためには，適応範囲の広さ，使いやすさが重要な要素である．しかし，それらを重視するあまり性能が犠牲になってはいけない．

本稿では，一般に広く使われているオブジェクト指向言語およびその処理系で，各種並列言語がサポートする主要な並列処理プログラミングスタイルをサポートできるかどうかを論じる．プログラミングのしやすさおよび性能についても検証する．実現不可能と判明した場合には，その原因を究明し解決案を示すこととする．結果として，MPC++を用いて主要プログラミングスタイルをサポートできることを示せた．性能も一定の結果を得ることができた．ただし，十分な成果が得られない場合

† 東京大学 The University of Tokyo
†† 東京工業大学 Tokyo Institute of Technology
††† 新情報処理開発機構 Real World Computing Partnership

もある.

本稿の以降の構成は, 第2章で研究の目的と基本方針を述べ, 第3章で扱った主要プログラミングスタイルを説明し, 第4章でライブラリの実装について説明し, 第5章で性能評価を示し, 第6章でまとめを行なう.

2. 研究目的と基本方針

ある言語が汎用製品 (Commodity) であるための条件としては,

(1) 様々な並列プログラミングスタイルに対応できること

(2) プログラマビリティの良さ (使いやすさ)

(3) ある程度のパフォーマンスを得られること

が挙げられる. 汎用であるためには, (1) は当然満たさなければいけない条件である. また, 様々な並列プログラミングスタイルに対応できるからといって, 特殊な記述を要求したり, 一般的でない文法を持つ使いにくい言語であってはいけない. そこで, (2) のプログラミングのしやすさが要求される. また, (1),(2) を満たすものであっても, 性能が極端に悪く, 全く実用的でないというのでは問題がある (3).

汎用であることから, 特殊な機能を言語に追加したり, コンパイラを変更することは望ましくない. そこで, 次の基本方針をとる.

- 言語が許している言語拡張以外の拡張はしない
- コンパイラは変更しない

本稿では, 一般的に広く使われて言語として C++ を取り上げた. C++ を選択した理由として, 言語内の拡張性の高さが挙げられる. 上の基本方針に従って, C++ のオブジェクト指向の標準機能である, クラス, インヘリタンス, テンプレート, オーバーロードなどを使うことにする.

C++ をベースに並列機能を拡張した言語で, 特殊な言語拡張を行なっていないものを簡単に紹介する. Gannon,Chien,Kesselman らによる HPC++[3] は, 並列 STL(Parallel Standard Template Library) を用いて, HPF 流の同期 SPMD を実現するものである. また, 石川による MPC++(Ver. 2.0 level 0)[2] は, グローバルポインタ, スレッド生成, PUT/GET 方式のメモリアクセスの拡張を行ない, 主に並行オブジェクトをサポートしている. HPC++ も MPC++ も, 特定の並列プログラミングスタイルのみをサポートしている.

我々は, MPC++ をベースに, C++ の言語で許されている拡張機能のみを使用して, より広範な並列プログラミングスタイルをサポートできることを示し, 汎用製品としての言語の可能性を検証する.

また, コンパイラは標準的な C++ コンパイラを使用し, 性能評価はワークステーションクラスタ上で行なう.

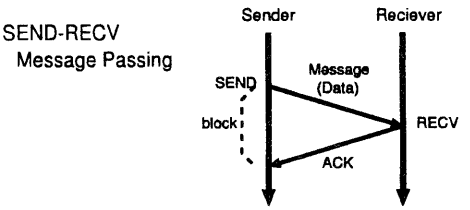

図1 SEND-RECV メッセージパッシング

3. 主要プログラミングスタイル

並列処理において, 主要なプログラミングスタイルとして, 次の3つを取り上げた. これらは, MPC++ や HPC++ が直接にはサポートしていないものである.

3.1 SEND-RECV メッセージパッシング

MPI に代表される SEND-RECV 型のメッセージ通信によるプログラミングパラダイムである. 送信側で SEND, 受信側で RECV をプログラマが明示的に発行し, バッファを経由して通信を行なう. 通信は同期的に行なわれ, メッセージ通信中は処理は block される (図1).

3.2 非同期通信 SPMD

SEND-RECV メッセージパッシングに対し, 非同期通信 SPMD は通信中に処理を block しないのが特徴である. これによって, 通信と計算のオーバーラップが可能になり, 性能が向上する. また, リモートメモリに対し直接アクセスすることが可能である.

非同期通信 SPMD をサポートする言語として, Split-C[4] がある. Split-C は分散メモリ型並列計算機向けに設計された SPMD モデルの C の拡張言語である. グローバルアドレス空間の導入による共有メモリモデルのプログラミングスタイルのサポート, グローバルポインタ, split-phase 代入と呼ばれる非同期通信といった特徴を持つ. split-phase 代入は **put, get** によって行なわれる. **get** はリモートプロセッサへデータの受信要求を発行して処理を終える. 受信要求を受けとったプロセッサはデータを返信する. 一方, **put** はデータ送信の起動をして処理を終える. データはリモートプロセッサの指定領域に格納され, アクノリッジを返す. **get** のデータの受信, **put** のアクノリッジの回収の完了は, **sync** によって行なわれる (図2). その他に **store** という, アクノリッジを返さない **put** がある.

3.3 Cache-coherent 分散共有メモリ

分散共有メモリは, 分散メモリ型の並列計算機上で共有メモリモデルのプログラミングスタイルを実現するものである. 性能向上のため, 各プロセッサはローカルにキャッシュを持つ. このキャッシュは, shared-object に対するアクセスに使用される. 元データとキャッシュ内のコピーのコンシステンシを保つために, メモリマネ

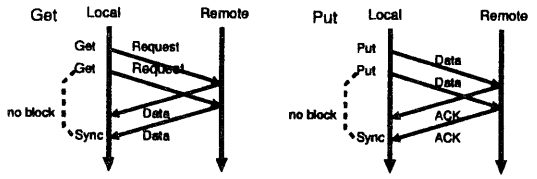

図2 非同期通信SPMD(Split-C流)

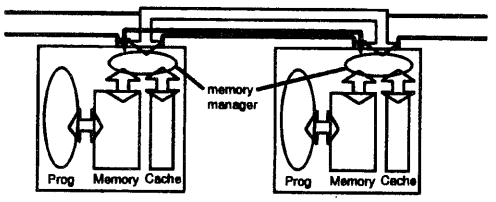

図3 Cache-coherent DSM

ジャ間で通信が行なわれる．コンシステンシが保たれた状態が，cache-coherentである(図3)．

データ書き込み時にキャッシュコピーを更新するwrite-update方式と，無効化するwrite-invalidiate方式がある．

4. ライブラリ実装

MPC++の通信機能とC++の言語拡張機能を使用して，前節に述べたプログラミングスタイルをサポートするためのライブラリを実装した．まず，MPC++の通信機能について簡単に説明し，次に各ライブラリ実装について説明する．

4.1 MPC++

MPC++(Ver. 2.0 level 0)の通信機能には，リモートリード`_mpcRemoteMemRead()`，リモートライト`_mpcRemoteMemWrite()`，同期構造体`Sync`がある．それぞれSplit-Cの**get, store, sync**に対応する．また，MTTL(Multi Threads Template Library)[2]によってグローバルポインタ，スレッド生成，リダクション，バリア同期などの機能が提供されている．MPC++ Ver. 2.0 level 0は，ソースコードをバックエンドのg++に渡し，並列および通信処理のライブラリをリンクする．

4.2 MPI on MPC++

SEND-RECVメッセージパッシングモデルのMPIは，既にRWCによってMPC++上で実装されている[5]．よって，本稿ではその性能評価のみを行なう．

4.3 非同期通信SPMD(Split-C++)

我々は，MPC++上でSplit-Cと同等の機能を持つクラスライブラリを実装した(Split-C++)．Split-Cの持つ大きな特徴であるグローバルポインタ，split-phase代入，spread arrayをサポートする．グローバルポインタは，MPC++ MTTLに同等の機能を持つグローバルポインタが用意されている．

split-phase代入については，`mpcSplitC`クラスを用意した．`mpcSplitC`には，次のメソッドがある．

- `mpcPut(pe,raddr,laddr,size)`
  Split-Cの**put**に対応．リモートメモリへの書き込みと，**put**回数のカウント．
- `mpcGet(pe,raddr,laddr,size)`
  Split-Cの**get**に対応．リモートメモリからの読み出しと，**get**回数のカウント．
- `mpcSync()`
  Split-Cの**sync**に対応．`mpcPut`のACK回収回数，`mpcGet`のデータ受信回数が，それぞれの発行回数に一致するまで待つ．
- `mpcStore(pe,raddr,laddr,size)`
  Split-Cの**store**に対応．リモートメモリへの書き込みと，送信データサイズに関する情報をリモートプロセッサに通知する．
- `mpcStoreSync()`
  Split-Cの**store_sync**に対応．`mpcStore`に対する**sync**待ちを行なう．

split-phase代入は，プロセッサごとに存在する`mpcSplitC`オブジェクトを通じて行なう．**put**の例を次に示す．MPC++のグローバルポインタは，`getPe`，`getLaddr`メソッドによってそれぞれプロセッサID，ローカルアドレスを得ることができる．`mpcSplitC`オブジェクトを`msc`，リモート変数を`rm`，ローカル変数を`lc`とすると，**put**は，

```
#define put(sc,r,l) \
    sc.mpcPut(r.getPe(),r.getLaddr(),
              &l,sizeof(l));
#define sync(sc)    sc.mpcSync();
mpcSplitC msc;
foo() {
  GlobalPtr<int> rm;
  int lc;
                    // on Split-C
  put(msc,rm,lc); // rm := lc;
  sync(msc);        // sync();
}
```

と書ける．Split-Cでの**put**は，`rm := lc;`であり，上記のように簡単な機械的書き換えによってサポート可能である．

次に，プロセッサ間に分散された配列(spread array)を説明する．Split-Cのspread arrayは，

```
double A[10]::[10][10];
```

のように定義される．::の右側は各プロセッサ上に確保される配列の大きさを表す．::の左側はプロセッサ間で分割される配列を表し，サイクリックに分割される．プロセッサ数が8とすると，プロセッサ0には A[0][0][0] から A[0][9][9] までと A[8][0][0] から A[8][9][9] までが割り当てられる．

spread array をサポートするために DistArray クラステンプレートを新たに設計，実装した．DistArray は，Init メソッドによる配列の割り当てと，[] および各種代入オペレータのオーバーロードによる配列操作を実現している．配列要素に対する代入，参照などのオペレーションは，オペーレータのオーバーロードにより通常の配列操作と同様に行なうことができる．

Init メソッドは，分割次元数 (:: の左側)，ブロック次元数 (:: の右側と，それぞれの次元の大きさを与えることによって，全プロセッサに配列を確保する．上の例の spread array を書き換えると，

```
DistArray<double> A;
A.Init(1,2,10,10,10);
```

となる．

DistArray を使ったリモートデータへのアクセスは，要素単位で行なわれる．

4.4 Cache-coherent 分散共有メモリ

Cache-coherent 分散共有メモリモデルを実現するためのクラスライブラリも実装した．キャッシュを管理する Cache クラスと，shared-object を扱うクラスを用意した．ただし，現在のところ対応している shared-object は，配列のみである．キャッシュは，キャッシュディレクトリによる管理を行なうオブジェクトベースのソフトウェアキャッシュである．coherency policy として，write-invalidiate を採用している．キャッシュ領域は 1MB で，キャッシュラインは 128 本，FULL WAY である．キャッシュラインサイズは，デフォルトで 256 バイトであるが，変更可能である．

Cache のメソッドは，以下のものがある．

- `isCached(pe,laddr,size)`
  キャッシュにリモートデータが入っているかを調べる．存在するならローカルアドレスを，しないなら 0 を返す．
- `readRemote(pe,raddr,laddr,size)`
  キャッシュにリモートデータが存在するかを確認し，存在すればキャッシュからデータを読む．存在しなければ，リモートプロセッサからデータを読み，キャッシュにコピーをとる．また，オーナーに対しデータ参照の通知を発行する．
- `writeRemote(pe,raddr,laddr,size)`
  オーナーに対し invalidiate の要求を出し，リモートへのデータの書き込みをする．オーナーは，参照先のプロセッサに対し invalidiate メッセージを発行する．
- `writeLocal(dest,src,size)`
  ローカルへのデータ書き込みを行なう．そのデータが参照されていれば，invalidiate メッセージを発行する．
- `clearCache()`
  キャッシュをクリアする．

分散配列 DistArray クラスは，Cache オブジェクトを通してデータにアクセスすることと，Init の指定が異なることを除いて，同じ機能を持つ．

5. 評　　価

本節では，それぞれのプログラミングスタイルを代表するベンチマークのいくつかを移植，実行して，性能評価を行なう．また同時に，プログラミングのしやすさ，移植しやすさについても，評価を行なう．

実行するベンチマークと移植に要したソースコードの変更行数を表1に示す．ベンチマークは，SEND-RECV メッセージパッシングについては，MPI の Scalable BLAS3 テストから行列積 GEMM を，非同期通信 SPMD については，Split-C の examples に含まれる行列積 mm, cannon を，Cache-coherent DSM については，SPLASH2 の fft を使用した．

実装したクラスライブラリの使用により，少量の変更で移植可能であった．

表1 実行ベンチマークと変更行数

	Prog.	変更行数	説明
MPI on MPC++	GEMM	15	行列積 (800,800)
非同期通信 SPMD (Split-C)	mm	~40/700	行列積 (128,128)
	cannon	~30/500	カノン行列積 (64,128)×(128,96)
cc DSM	fft	~90/1400	1024 1D-FFT

5.1 実験環境

実験には，RWC の Sun のワークステーションクラスタを使った．36 台の Sun SPARC station 20/model 71(75MHz,32MB) が，Myrinet(LANai4.0) で接続されている．通信ライブラリを使ったホスト間通信のバンド幅は約 30MB/s である．OS は SCore である．

5.2 基本通信性能

MPC++ の基本リモートメモリオペレーションである _mpcRemoteMemRead, _mpcRemoteMemWrite の発行時間は，それぞれ 14.9,4.8μsec であり，レイテンシは 34.0,22.1μsec である．スループットを図4に示す．立ち上がりが悪く，通信サイズが 80KB 以上ないと通信ライブラリの限界性能に達することができない．8KB 付近の性能低下は，キャッシュ効率の低下が原因と考え

られる.

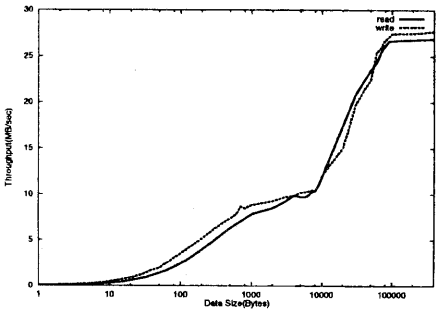

図4 基本リモートメモリオペレーションのスループット

5.3 MPI on MPC++

Scalable BLAS3 テストの **GEMM** を, -O3 オプションでコンパイルした. MPI/MPC++ と MPICH との比較を図5に示す. MPICH は, ネットワークが switched 100BT である. 基本性能では, MPI/MPC++ の方がスループットで3倍以上, レイテンシで5倍以上高性能であるにも関わらず, 実行時間は遅い. 他のベンチマークでは, MPI/MPIC++ の方が速い. 原因を調査中である.

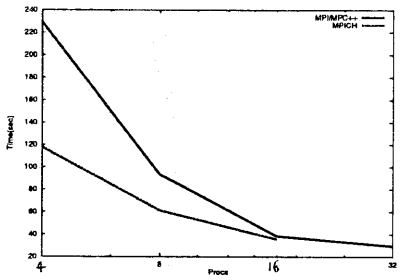

図5 **GEMM**(MPI) (800,800)

5.4 Split-C on MPC++

Split-C の付属 examples に含まれる **mm, cannon** の実行結果を示す. 2つとも -O2 オプションをつけコンパイルした.

- **mm**
  行列積を行なう. 問題サイズは (128,128) 行列 ×(128,128) 行列であり, blocking factor は 16 である. 通信サイズは 2KB であり, 通信パターンは規則的である.
- **cannon**
  カノンの行列操作アルゴリズムにより, 行列積を行なう. 問題サイズは, (64,128) 行列 ×(128,96) 行列である. 通信サイズは, プロセッサ数が 2, 4 のときは 16KB, 24KB, プロセッサ数 8,16 のときは 4KB, 6KB, プロセッサ数 32 のときは 1K, 1.5KB である. 通信パターンは規則的である.

図6,7に, 全体の実行時間, 通信時間, 計算時間の計測結果を示す. **mm** は台数効果がよく現れている. 通信回数が1プロセッサが所有するデータ数に比例して減少するためである. 同サイズの MPI の GEMM と比較して, 大幅に速いことがわかる. 一方, **cannon** は, プロセッサ数の増加に伴い通信時間が増加している. これは, 通信サイズの減少によるスループットの大幅な低下が原因である.

富士通の AP1000+ 上に実装された Split-C[7] でのベンチマーク実行結果との比較を表2に示す. **mm** の結果から, 実行時間で 1.8 倍程度, 通信時間で 1.3 倍程度の遅さで実行されていることがわかる. 専用の並列計算機に対して, この結果ならば十分実用的であると判断できる. 一方, **cannon** は台数を増やした結果, 性能が大幅に低下している. これは, 通信サイズの減少が原因である. MPC++ では, 20KB のスループットは約 18MB/s, 4KB のスループットは約 10MB/s となっている. 一方, AP1000+ 版では, 1KB で既にハードウェアのバンド幅に近い 23 ~ 24.5MB/s を達成している. このネットワークの特性の違いが, 実行時間に大きく反映している.

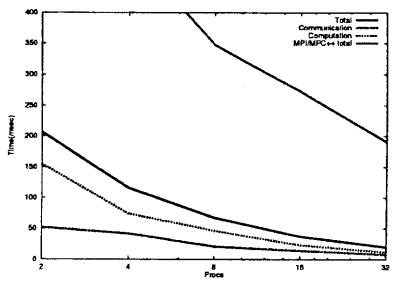

図6 **mm**(Split-C example) (128,128)

表2 Split-C on AP1000+ との比較

Procs	MPC++	on AP1000+
mm		
4	115.7(41.7)	61.7(-)
16	37.0(14.0)	21.2(10.5)
cannon		
4	42(11)	30.2(-)
16	35(24)	9.89(4.6)

単位: msec, () 内は通信時間

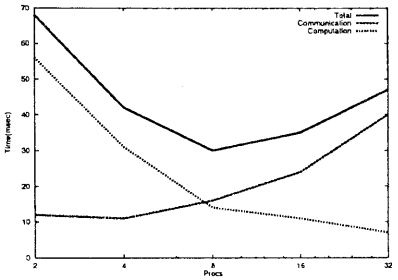

図7 **cannon**(Split-C example) (64,128)×(128,96)

5.5 Cache-coherent DSM

SPLASH2 の complex 1D-FFT の実行結果を図8に示す. 通信部分を最適化したバルク転送版, キャッシュを OFF にしたものも併記する. いずれも -O2 オプションにてコンパイルした. キャッシュ OFF では通信は 8 バイト単位で行なわれ, キャッシュ ON では 256 バイト単位で行なわれる.

キャッシュ ON と OFF と比べると, キャッシュによる性能向上が見られる. しかし, 台数が増えキャッシュのヒット率が低下すると, 通信のオーバーヘッドによる性能の逆転が発生する. また, キャッシュ ON とバルク転送を比べると, 2 倍以上の実行時間の差が存在する. これは, キャッシュ実現のためのソフトウェアオーバーヘッドである. キャッシュ ON, バルク転送で台数効果が見られないのは, 台数増加による通信回数の増加で打ち消されたためである. キャッシュのオーバーヘッドの削減による性能向上の余地がある.

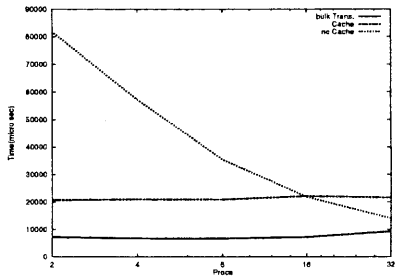

図8 **fft**(SPLASH2) 1024 complex

5.6 考　　察

ベンチマークの測定実験の過程で, [] のオペレータをインライン展開したものがコンパイラで最適化されず, 効率低下が起こっていることが確認されている. 具体的には, 多次元配列の添字処理, loop-invariant 処理である. C++ のコンパイラの最適化はまだ不十分である.

6. ま と め

並列計算/処理における, ソフトウェア面での汎用製品 (commodity) の可能性を検証するため, MPC++/C++ をベースに主要並列処理プログラミングスタイルへの対応を試みた. 結果として, C++ の持つ言語拡張機能のみを用いて, MPI, 非同期通信 SPMD, Cache-coherent DSM といったプログラミングスタイルに対応可能であることを示せた. また, 実装したクラスライブラリを使用して, それぞれのプログラミングスタイルの主要ベンチマークを移植したところ, わずかな機械的な変更によって移植可能であった. それらのベンチマークの性能は, 非同期通信 SPMD モデルについては比較的良好な結果を得た. その他の 2 つについては, ライブラリの改良の余地がある.

現状のオブジェクト指向言語に与えられた言語機能だけで, 各種の並列処理プログラミングスタイルに対応できる可能性は十分あると考えられる. ただし, 実装したライブラリには, さらなる最適化が必要である.

今後の課題としては, ベンチマークを増やすこと, クラスライブラリを完成させ改良することが挙げられる. また, ネットワーク性能の向上などハードウェア面での支援が期待される. 配列の添字処理, loop-invariant などのコンパイラレベルでの最適化が必要とされるものについて検証し, それらを組合せた利用を考える必要がある.

参 考 文 献

1) Y. Ishikawa and et. al.: "MPC++," In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pp. 427–466. MIT Press, 1996. To be published in 1996 Spring.
2) Y. Ishikawa: "Multiple Threads Template Library – MPC++ Version 2.0 Level 0 Document – Document Revision 0.1." Techinical Report TR96012, RWC, September 1996. `http://www.rwcp.or.jp/people/mpslab/mpc++/mpc++.html`
3) D. Gannon and et. al: High Performance C++ `http://www.extreme.indiana.edu/hpc++`
4) D. E. Culler and et. al.: "Parallel programing in Split-C," *Supercomputing '93*, Nov 1993.
5) F. B. O'Carroll: MPIXX User's Guide `http://www.rwcp.or.jp/people/ocarroll/usingmpixx.html`
6) S. C. Woo and et. al: "The SPLASH-2 Programs: Characterization and Methdological Considerations." *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.
7) 小林ら: AP1000+ における Split-C の実装と実行性能の評価. *Hokke2 '95 HPC*研究会, 1995.