Efficient, Portable Implementation Of Asynchronous Multi-place Programs

秋山 茂樹

全体ミーティング (2010/12/8)

Efficient, Portable Implementation Of Asynchronous Multi-place Programs

- 並列プログラミング言語 X10 のサブセット FLAT X10 の設計と実装を提案
  - **非同期通信を用いた SPMD プログラム**を表現可能
  - 効率が良く、シンプルでポータブルな実装
    1. 性能を出しやすいサブセット
    2. C++ **SPMD** + active messaging への変換
  - HPC Challenge Benchmark などで MPI, UPC 実装に匹敵する性能を達成

Efficient, Portable Implementation Of Asynchronous Multi-place Programs

- 著者
  - Ganesh Bikshandi (IBM STG, India)
  - 他
    - IBM T.J. Watson Research Center / India Research Lab
    - Interactive Supercomputing Inc.
- 発表場所
  - ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming 2009 (PPoPP '09)

発表の流れ

1. X10 の概要
2. Flat X10
3. Flat X10 コンパイル戦略
4. Flat X10 ランタイム
5. 評価
6. まとめと future work

X10 とは

- 高生産性を目指した並列プログラミング言語
  - 並列分散プログラミング機構
    - lightweight threads, atomic blocks,
      PGAS constructs, etc...
  - 逐次サブセットは Java like な言語
    - その他 type inference, constraint types のサポートなど

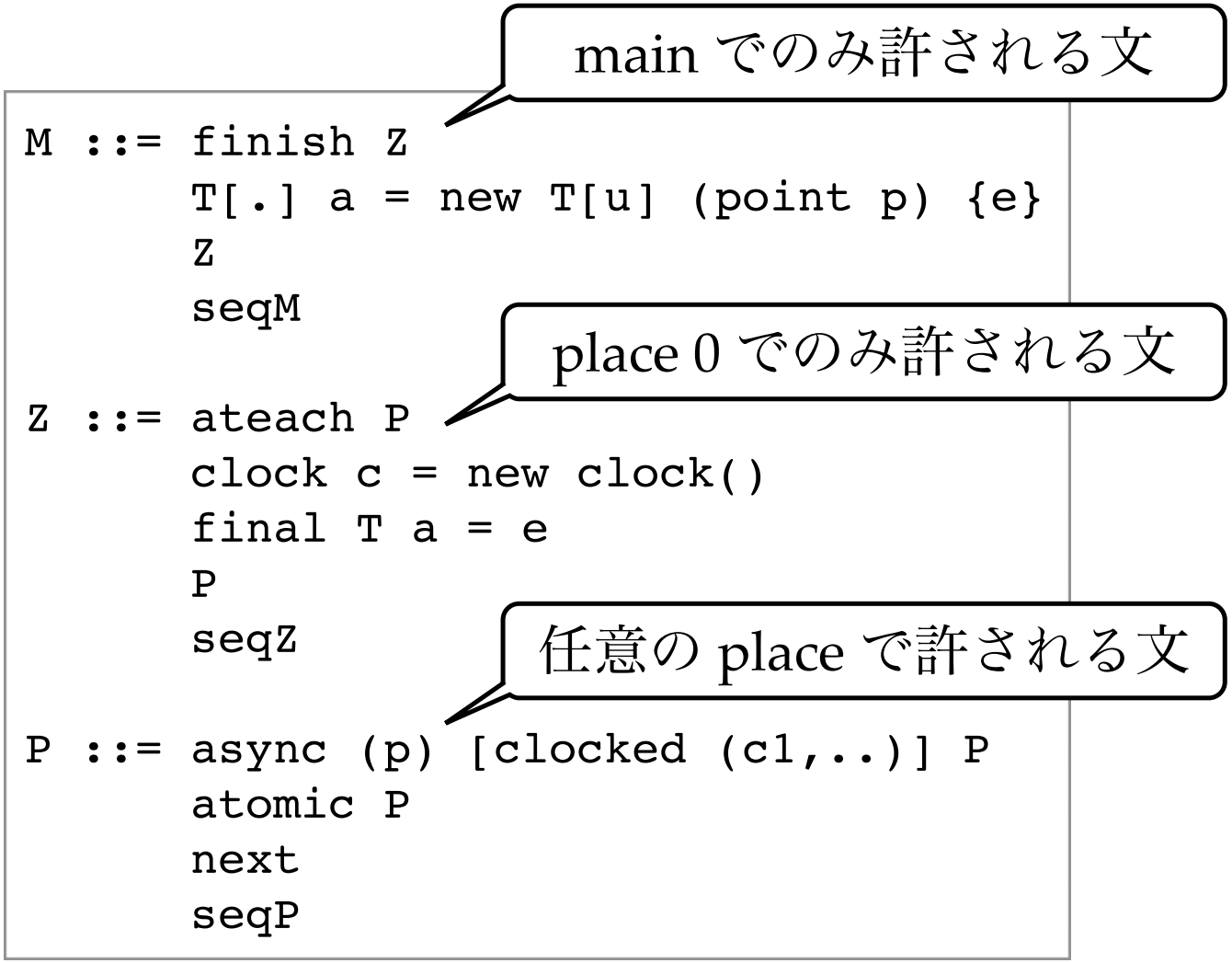
Partitioned Global Address Space

- アドレス空間は共有だが「分割」されている
  - 各分割を place と呼び、place 内で生成された
    オブジェクトは他の place からアクセスできない
  - ただし、remote reference を持つことはできる

FLAT X10 とは

- 非同期通信を用いた SPMD プログラムを記述するのに特化した X10 のサブセット
  - シンプルかつ効率的に実装できるように設計
  - 特徴
    - 各 place はプロセッサ (worker thread) と一対一対応
    - SPMD プログラムの記述に必要ない機構は制限

FLAT X10's syntax



async, finish

- activity の生成と同期

```
finish {
  final int n = 123;
  async (Place.place(0)) {
    Console.OUT.println("n = " + n);
  }
  final int m = 456;
  async (Place.place(1)) {
    Console.OUT.println("m = " + m);
  }
}
```

place 0 に activity を生成

final 変数を使用できる

ブロック内で生成された activity の終了を待つ

finish は発生したすべての例外をまとめる機能を持つ
(rooted exception model)

async のスケジューリング (in FLAT X10)

- Remote async は
  - データの送信時
  - (ランタイムでの) 明示的な polling 時

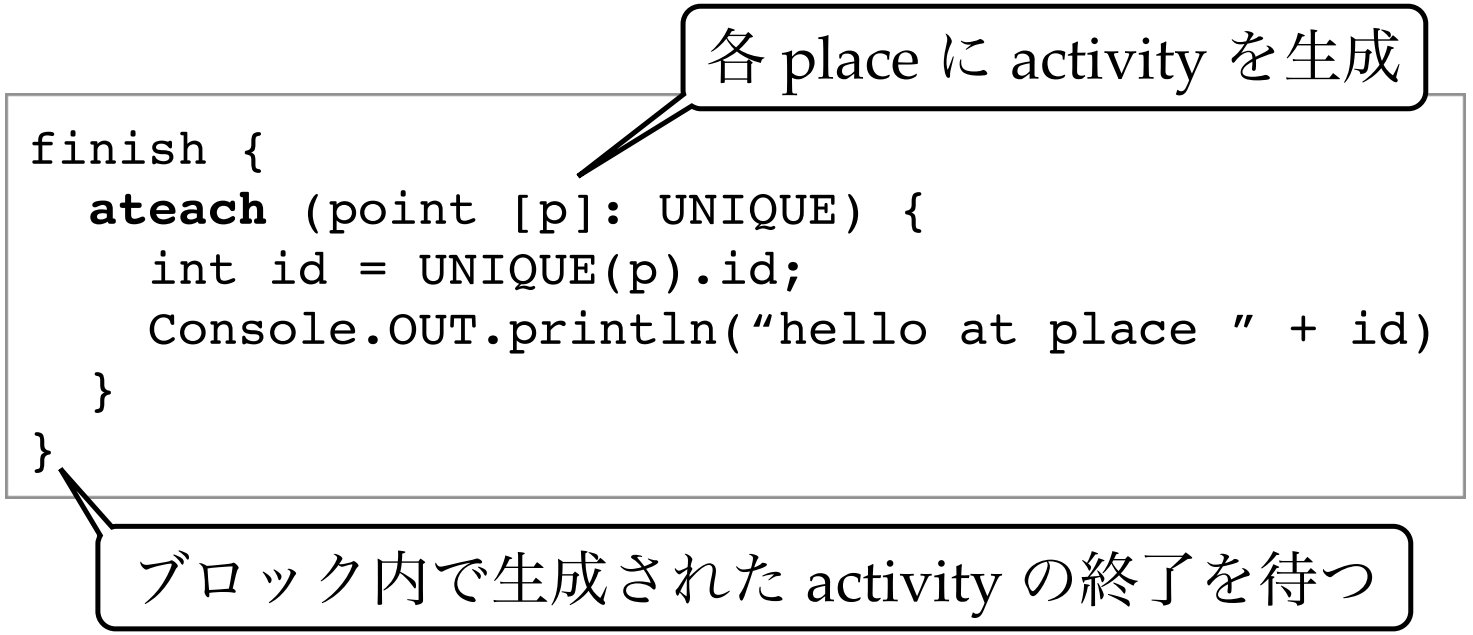
  に**即座に**実行される

async, finish の制約 (in FLAT X10)

- finish
  - main にしか書けない (大雑把には)
  - ネストできない
- async
  - ネストできない
  - body は non-blocking でなければならない
  - 例外を投げてはいけない

ateach

- 各 place に activity 生成

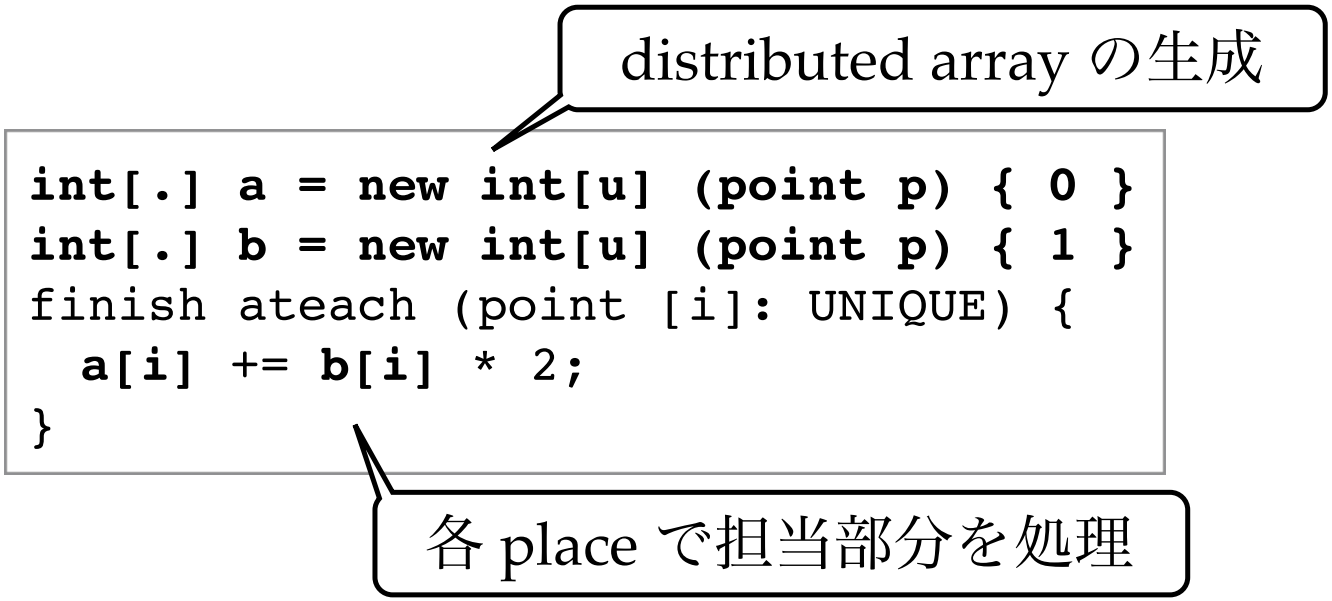


ateach の制約 (in FLAT X10)

- place 0 でしか実行できない
- unique distribution しか許されない
  - unique dist. = すべての place に 1点ずつ分散

Distributed Arrays

- place をまたがる配列



FLAT X10 における制約:
unique distribution しか許されない

Clocks

- バリア同期

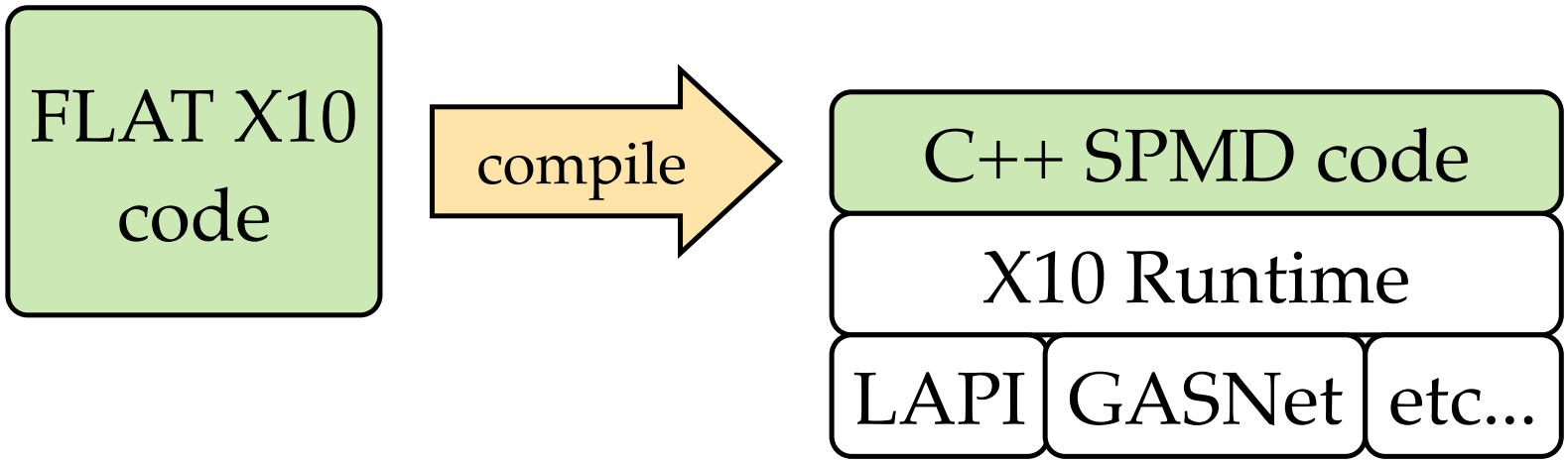
```
finish {
  clock c = new clock();
  async (Place.place(0)) clocked (c) {
    Console.OUT.println(“0-0”);
    next;
    Console.OUT.println(“0-1”);
  }
  async (Place.place(1)) clocked (c) {
    Console.OUT.println(“1-0”);
    next;
    Console.OUT.println(“1-1");
  }
}
```

“clock” の生成

c が登録された activity すべてが next 文に到達するまで待機

FLAT X10 コンパイラ

- FLAT X10 コードを C++ コードへ変換
  - 各プロセスで同じプログラムが動作 (SPMD)



SPMD design

- FLAT X10 の性質をうまく利用して
  C++ SPMD コードへ変換

コンパイル例1 (async 外の逐次ステートメント)

- async/ateach の外側にある逐次ステートメント

```
time = -mysecond();
finish {
  ateach ([p]: UNIQUE) {
    for (i=0;i<N;i++) {
      final int id = ...;
      async (UNIQUE[id]) {
        ...(snip)...
      }
    }
  }
}
time += mysecond();
Console.OUT.println(time);
```

```
if (here == p0)
  time = -mysecond();
finishStart(0);
Exception z = null;
  try {
    for (i=0;i<N;i++) {
      const int id = ...;
      libAsync(...);
    }
  } catch (Exc e) { z = e }
finishEnd(z);
if (here == p0) {
  time += mysecond();
  Console.OUT.println(time);
}
```

コンパイル例1 (finish)

- finishStart と finishEnd の組に変換

```
time = -mysecond();
finish {
  ateach ([p]: UNIQUE) {
    for (i=0;i<N;i++) {
      final int id = ...;
      async (UNIQUE[id]) {
        ...(snip)...
      }
    }
  }
}
time += mysecond();
Console.OUT.println(time);
```

```
if (here == p0)
  time = -mysecond();
finishStart(0);
Exception z = null;
  try {
    for (i=0;i<N;i++) {
      const int id = ...;
      libAsync(...);
    }
  } catch (Exc e) { z = e }
finishEnd(z);
if (here == p0) {
  time += mysecond();
  Console.OUT.println(time);
}
```

コンパイル例1 (ateach)

- 例外処理のみ

```
time = -mysecond();
finish {
  ateach ([p]: UNIQUE) {
    for (i=0;i<N;i++) {
      final int id = ...;
      async (UNIQUE[id]) {
        ...(snip)...
      }
    }
  }
}
time += mysecond();
Console.OUT.println(time);
```

```
if (here == p0)
  time = -mysecond();
finishStart(0);
Exception z = null;
try {
    for (i=0;i<N;i++) {
      const int id = ...;
      libAsync(...);
    }
} catch (Exc e) { z = e }
finishEnd(z);
if (here == p0) {
  time += mysecond();
  Console.OUT.println(time);
}
```

コンパイル例1 (async 内の逐次ステートメント)

※ final T a = e; は除く

- そのまま出力

```
time = -mysecond();
finish {
  ateach ([p]: UNIQUE) {
    for (i=0;i<N;i++) {
      final int id = ...;
      async (UNIQUE[id]) {
        ...(snip)...
      }
    }
  }
}
time += mysecond();
Console.OUT.println(time);
```

```
if (here == p0)
  time = -mysecond();
finishStart(0);
Exception z = null;
  try {
    for (i=0;i<N;i++) {
      const int id = ...;
      libAsync(...);
    }
  } catch (Exc e) { z = e }
finishEnd(z);
if (here == p0) {
  time += mysecond();
  Console.OUT.println(time);
}
```

コンパイル例1 (async)

- クロージャを作成し、libAsync を呼び出す

```
time = -mysecond();
finish {
  ateach ([p]: UNIQUE) {
    for (i=0;i<N;i++) {
      final int id = ...;
      async (UNIQUE[id]) {
        ...(snip)...
      }
    }
  }
}
time += mysecond();
Console.OUT.println(time);
```

```
if (here == p0)
  time = -mysecond();
finishStart(0);
Exception z = null;
  try {
    for (i=0;i<N;i++) {
      const int id = ...;
      libAsync(...);
    }
  } catch (Exc e) { z = e }
finishEnd(z);
if (here == p0) {
  time += mysecond();
  Console.OUT.println(time);
}
```

コンパイル例2 (async 外のif/while)

- 条件計算を p0 で行い、broadcast する

```
finish {
  int i = 10;
  while (i-- > 0) {
    final int x = i;
    async (UNIQUE[i]) {
      atomic {
        Console.OUT.println(
          "x = " + x);
      }
    }
  }
}
```

```
finishStart(0);
Exception z = null;
if (here == p0) i = 10;
while (true) {
  if (here == p0)
    flag = (i-- > 0);
  sendReceive(flag);
  if (!flag) break;
  if (here == p0) x = i;
  sendReceive(x);
  libAsync(...);
}
finishEnd(z);
```

コンパイル例2 (final T a = e)

- 式 e を p0 で計算し broadcast する

```
finish {
  int i = 10;
  while (i-- > 0) {
    final int x = i;
    async (UNIQUE[i]) {
      atomic {
        Console.OUT.println(
          "x = " + x);
      }
    }
  }
}
```

```
finishStart(0);
Exception z = null;
if (here == p0) i = 10;
while (true) {
  if (here == p0)
    flag = (i-- > 0);
  sendReceive(flag);
  if (!flag) break;
  if (here == p0) x = i;
  sendReceive(x);
  libAsync(...);
}
finishEnd(z);
```

コンパイル例2 (atomic)

- 何もしない (1 worker, async の body が non-blocking であるため不要)

```
finish {
  int i = 10;
  while (i-- > 0) {
    final int x = i;
    async (UNIQUE[i]) {
      atomic {
        Console.OUT.println(
          "x = " + x);
      }
    }
  }
}
```

```
void func-async0(args) {
  int x = (get from args);
  Console.OUT.println(
    "x = " + x);
}
```

```
      flag = (i-- > 0);
    sendReceive(flag);
    if (!flag) break;
    if (here == p0) x = i;
    sendReceive(x);
    libAsync(...);
  }
  finishEnd(z);
```

その他

- Clocks
  - すべての clock を無視して、next 式を
    global barrier (GlobalSync 呼び出し) に変換
- Distributed array 生成
  - Syntax sugar (ateach と finish の組に還元)

FLAT X10 ランタイム

- Active messaging (LAPI) を利用して
  - finishStart/finishEnd, globalSync, libAsync, sendReceive

  を実装
  - Active messaging:
    - *gets, puts, active message*

Active messaging

- *gets*
  - Remote processor からデータを受け取る
- *puts*
  - *Remote processor* にデータを送る
- *active message*
  - Remote processor により指定されたコードを実行

finishStart(cs)

```
int finishStart(int CS) {
  if (here == p0) {
    (CS を各childのContinueStatusに書き込む)

    LAPI_Fence;
    return CS;
  } else {
    (ContinueStatusに書き込まれるのを待つ)

    (と同時に、受信したmsgを処理する)

    CS = ContinueStatus;
    ContinueStatus = 0;
    return CS;
}
```

p0: 各 place に
次に実行する文(status)を通知

p0以外: status を受け
取ってそれを返す

finishEnd(exc)

まず global barrier で同期

p0: 子の終了をチェック、
例外を投げていたら
それをまとめて投げる

```
void finishEnd(const Exception* e) {
  LAPI_Fence;
  if (here == p0) {
    (eをbufferに追加);
    (FinishEndカウンタがN-1になるのを待つ)
    (と同時に、到着したmsgを処理する)
    if (buffer is not empty)
       throw MultipleExceptions(buffer);
  } else {
    (次の動作を行う active message をp0に送信)
      (親のbufferにeを追加し、)
      (FinishEndをインクリメントする)
    LAPI_Fence;
  }
}
```

p0以外: 終了と発生した例外を
p0 に通知

その他

- libAsync(p, f, args)
  - place p で f(args) が実行される
  - active message による実装
- sendReceive(x)
  - place 0 から他の place への broadcast
- globalSync()
  - ただの global barrier (LAPI_Fence)？

※ 論文に詳細なし

実験内容

- HPC Challange Benchmark
  - Stream
  - RandomAccess
  - FT
- NAS Parallel Benchmark
  - FT

実験環境

- IBM p575 cluster
  - 16CPUs SMP * 128 nodes = 2048 processes
  - CPU: Power5 1.9 GHz, Memory: 64GB, OS: AIX
- Blue Gene/L
  - “Virtual Node Mode”
  - 1 rack: 2048 processors

HPCC Stream

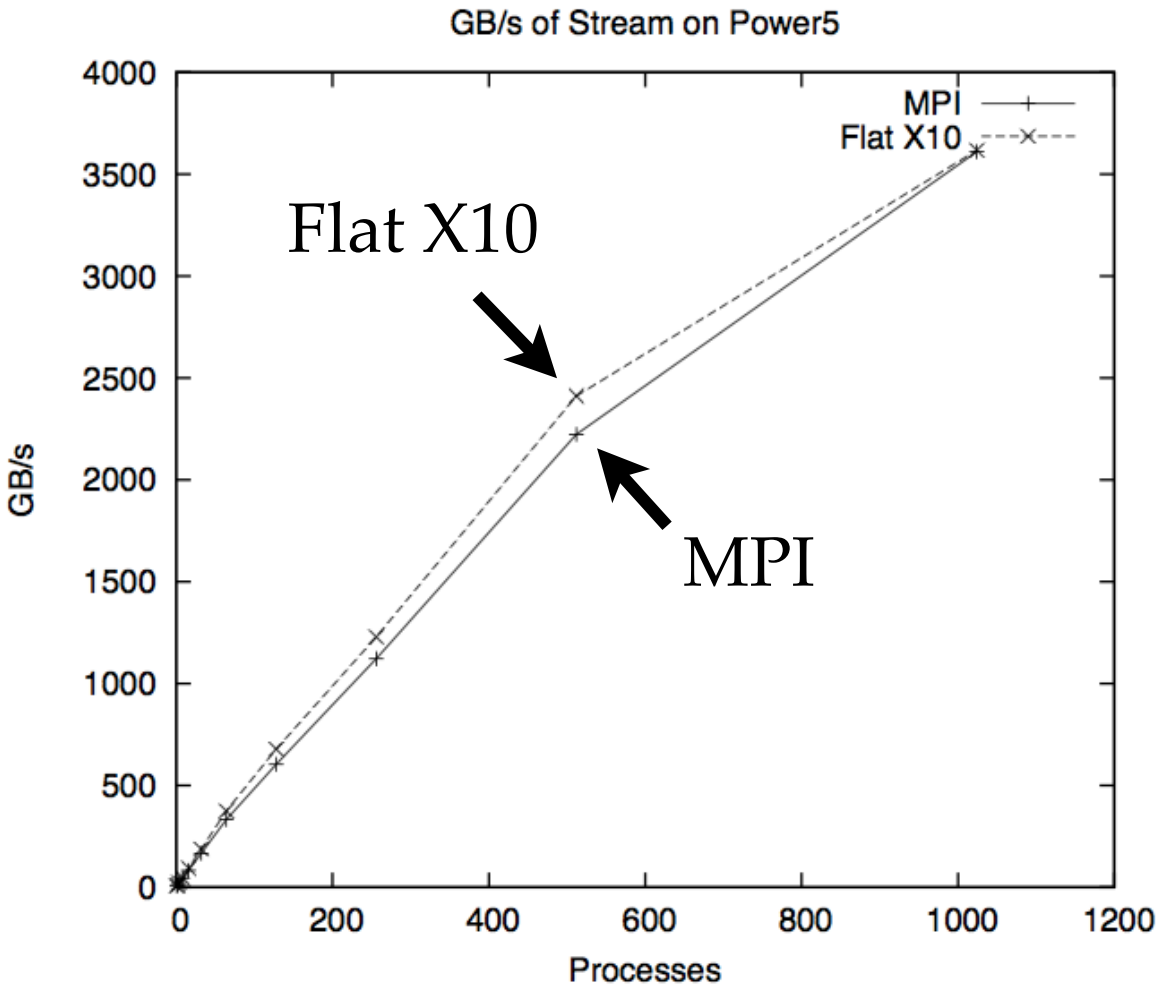
- distibuted array の積和算についてメモリバンド幅 (Gbyte/s) を計測
  - place 間の通信なし

```
ateach (point [p]: a) {
  c[i] = a[i];
  b[i] = alpha * c[i];
  c[i] = a[i] + b[i];
  a[i] = b[i] + alpha * c[i];
}
```

※ コードはイメージです

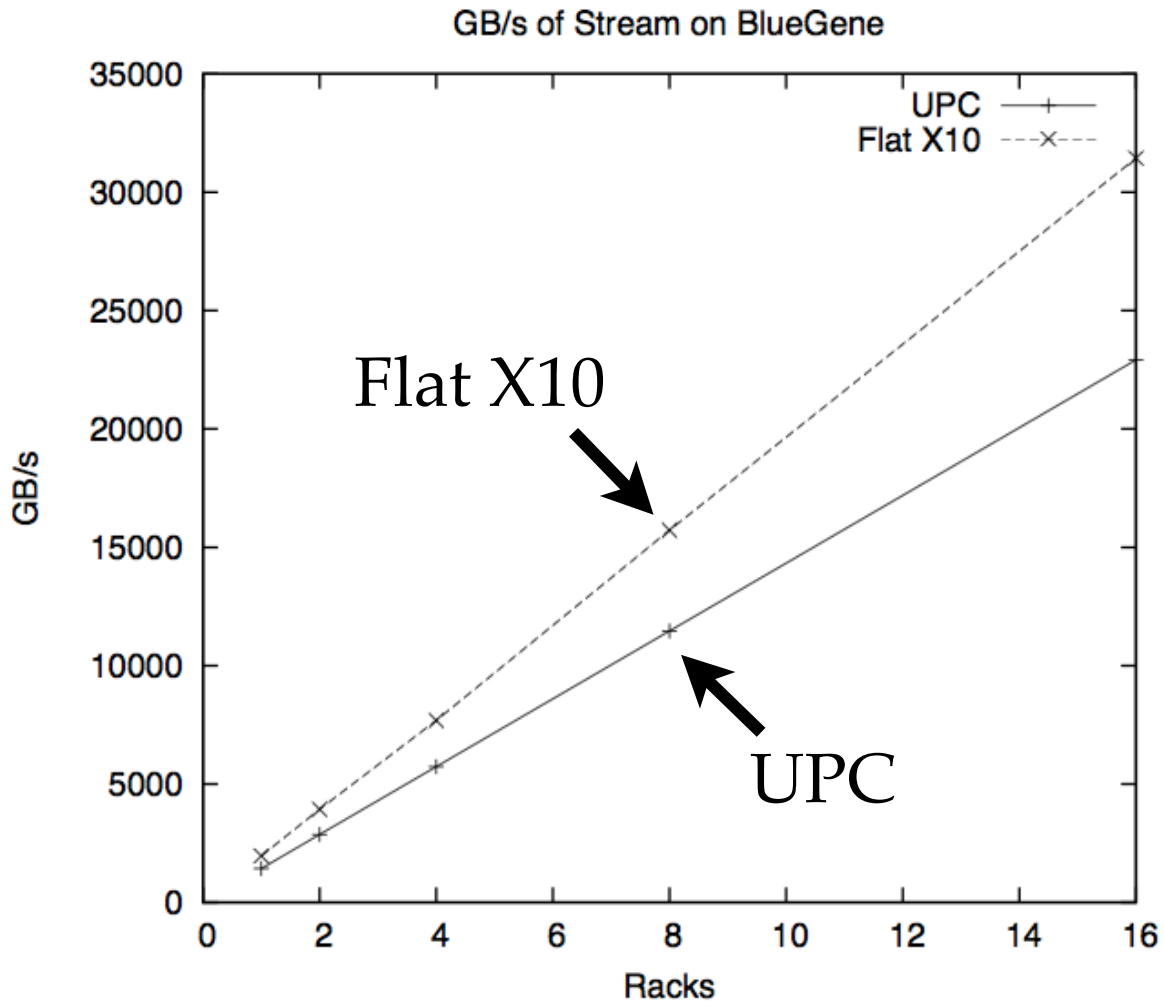
5. 評価

HPCC Stream (on Power5 cluster)



5. 評価

HPCC Stream (on Blue Gene/L)



HPCC Random Access

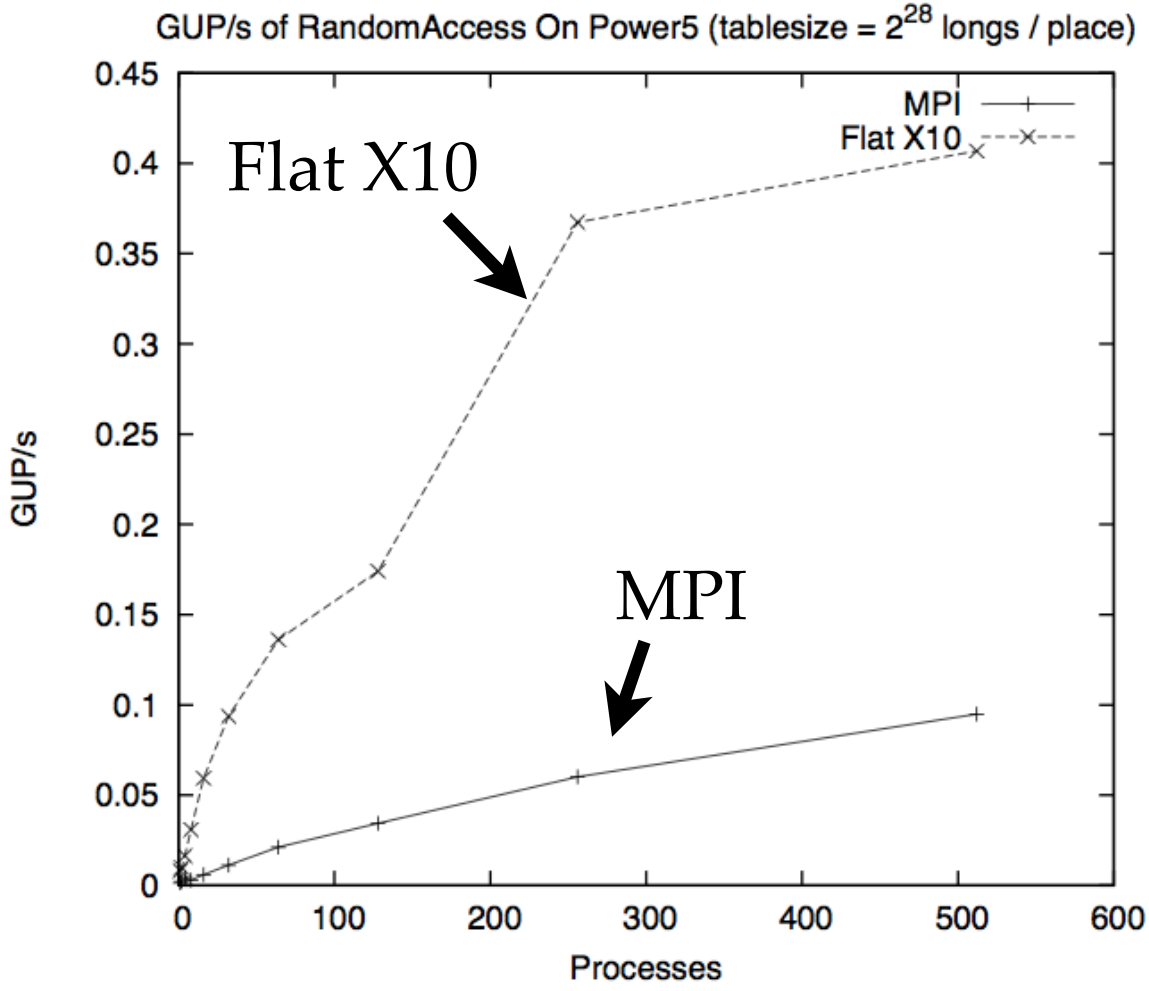
- distibuted array をランダムに XOR で更新
  - Giga Updates Per Second (GUP/s) を計測

```
ateach (point [p]: UNIQUE) {
  finish {
    for (int i=0;i<N;i++) {
      final int id = random();
      final int idx = random();
      async (Place.place(id)) {
        array[idx] ^= id + idx;
      }
    }
  }
}
```

※ コードはイメージです

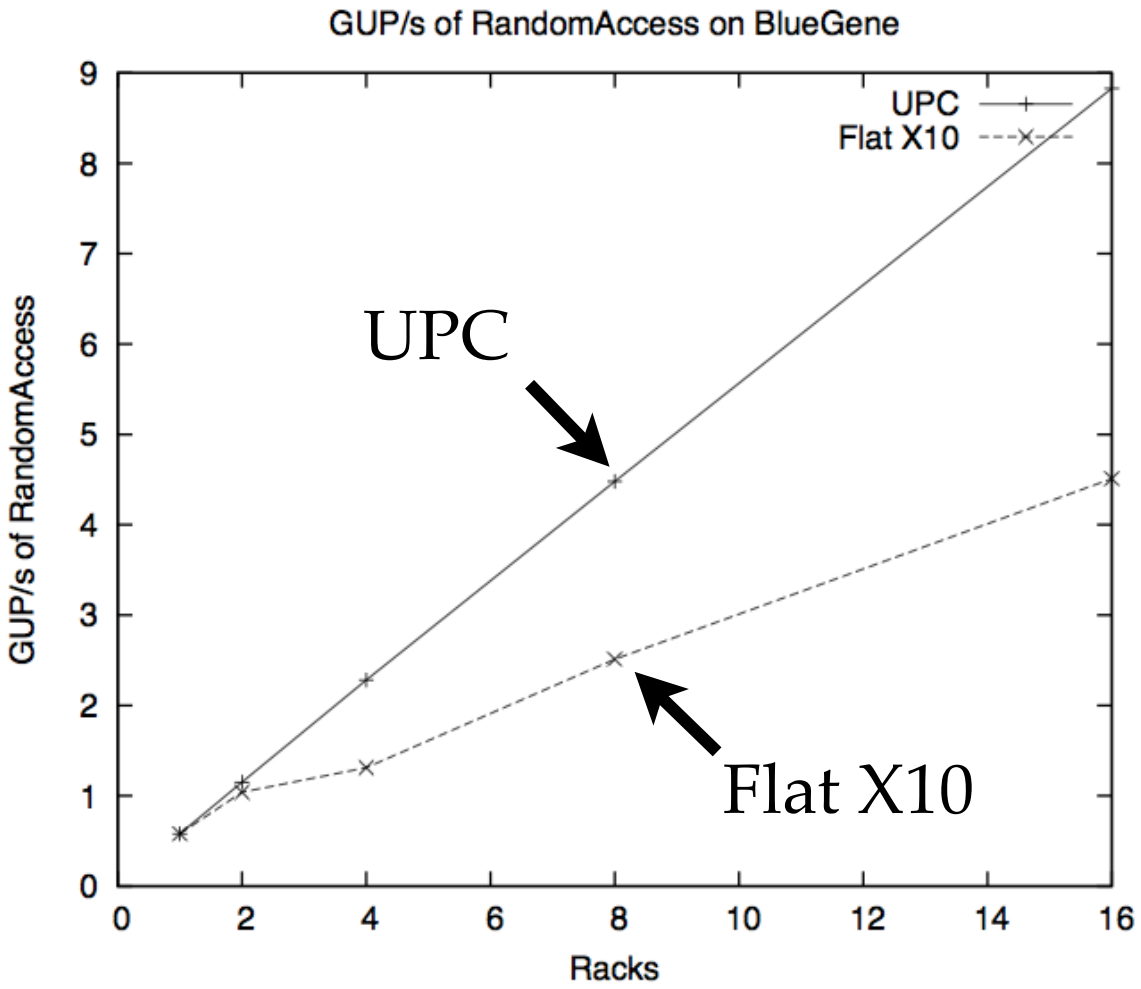
5. 評価

HPCC Random Access (on Power5 cluster)



5. 評価

HPCC Random Access (on Blue Gene/L)



HPCC FT

- 倍精度 1次元複素フーリエ変換
  - async による計算と通信のオーバラップ
  - ver.1: one sided arrayCopy interface

```
finish ateach (point [p]: UNIQUE) {
  for (int k=0; k<N; k++) {
    perform local transposition of the block k;
    async (k) { Arraycopy(...); }
  }
}
```

※ コードはイメージです

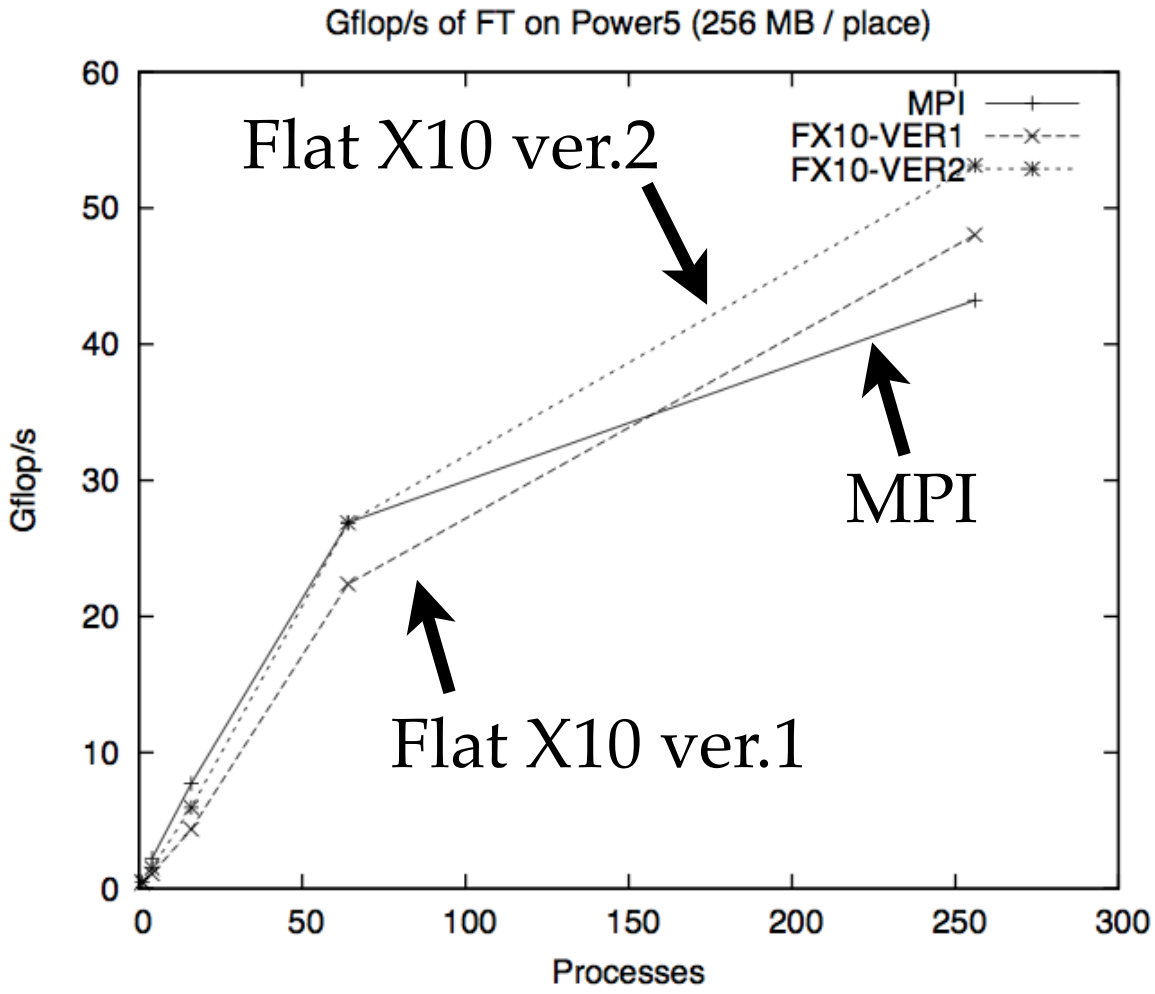
計算 (行列の転置)

通信

  - ver.2: 通信を MPI_Alltoall で置き換え (by hand)

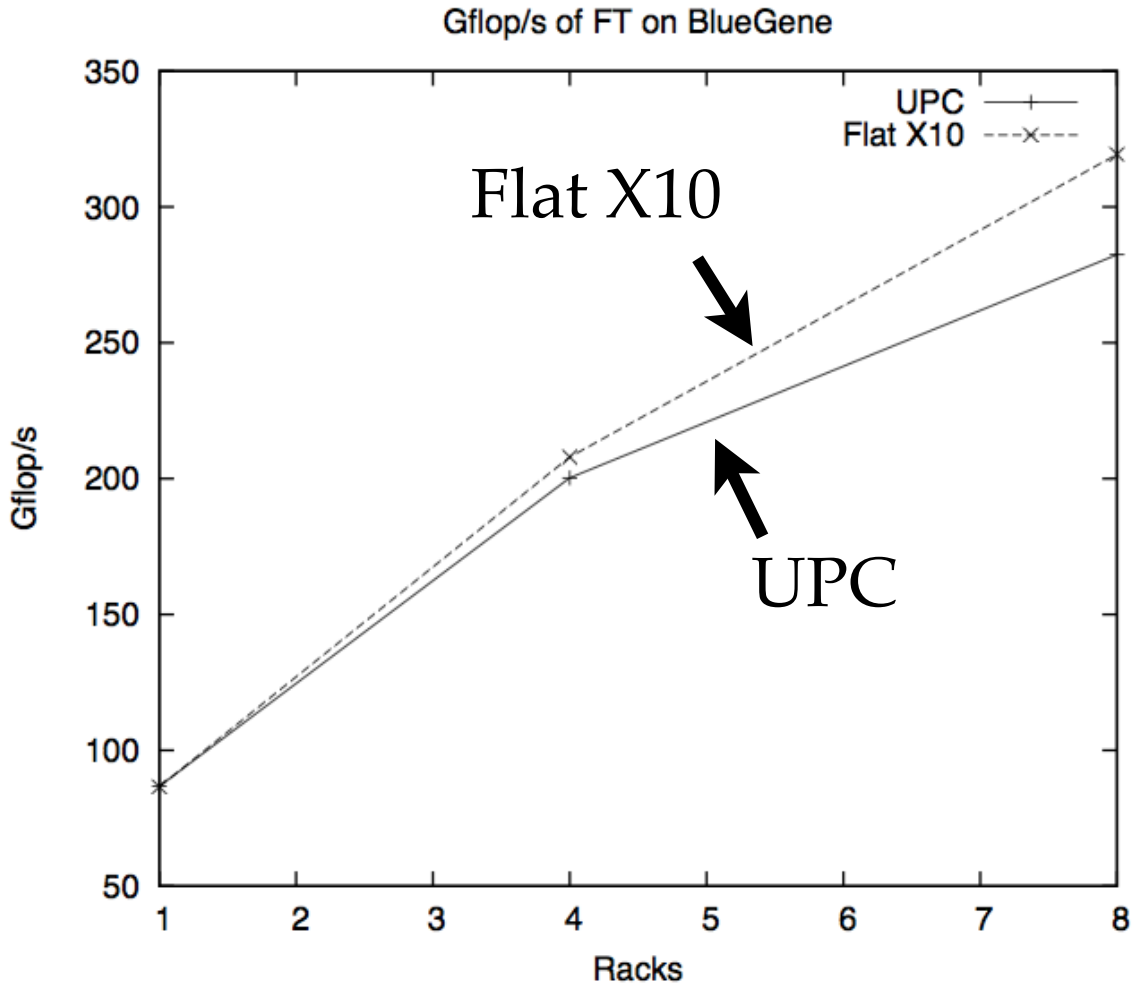
5. 評価

HPCC FT (on Power5 cluster)



5. 評価

HPCC FT (on Blue Gene/L)

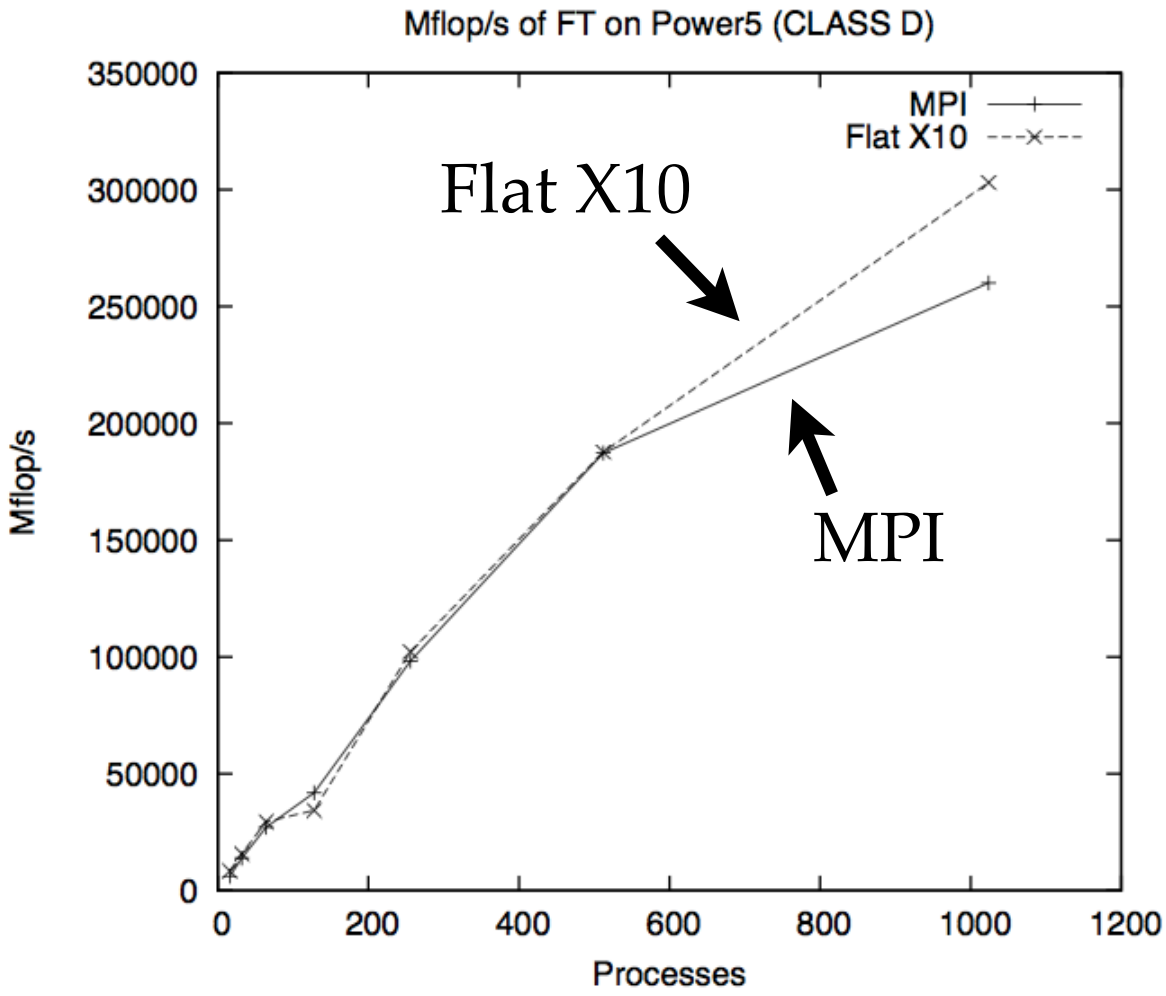


NPB FT

- 3次元フーリエ変換
  - async による計算と通信のオーバーラップ

5. 評価

NPB FT (on Power5 cluster)



まとめ

- X10 のサブセット FLAT X10 を設計・実装
  - SPMD + asynchronous messaging を表現するのに十分な表現力
  - シンプルかつ効率的なコンパイル戦略
  - シンプルかつ効率的なランタイム
- Power5 cluster と Blue Gene/L で評価
  - MPI, UPC に匹敵する性能を達成

Future work

- Place 内で 複数の activity をスケジューリングするためのフレームワークの導入
  - 複数 worker thread のサポート
  - Work stealing scheduling の実装 (cf. Cilk)
- Non-global barrier operations (clocks) とネスト可能な finish の導入