HPC向け高水準プログラミング言語X10の評価

佐 藤 芳 樹

東京大学 大学院情報理工学系研究科・JST-CREST
(現在 東京大学 情報基盤センター)

1．はじめに

ハイパフォーマンスコンピューティング(HPC)アプリケーション向けの新しい並列分散プログラミング言語X10の動作を東京大学情報基盤センターOakleaf-FX10上で確認し，実用に足る並列分散機能が有効か，パフォーマンスはスケールするかなどを明らかにする事を目的とした課題に取り組んだ。近年、HPC を利用する科学技術計算は、バイオ、流体工学、金融工学、気象、宇宙工学など多岐に渡っており、また複雑化するソフトウェアの開発生産性が共通課題となっている。2004年からIBM Researchによって開発されているX10は、生産性を10倍向上させる事を目標として命名された比較的新しい並列分散アプリケーション向けのプログラミング言語である。

2．X10とは

X10[1]は,2002年から2010年に推進された米DARPAのHPCSプログラムに基づき,IBMのPERCSプロジェクト[2]の一環で,HPCアプリケーションの生産性向上を目指して開発されたオブジェクト指向プログラミング言語である．X10 の主たる目的は，来るべき将来のエクサスケールコンピューティングを見据え，プログラムの開発生産性と実行性能を両立させるために，並列分散アプリケーションのプログラミングモデルをシンプル化する事である．そのため,X10では,PGAS(Partitioned Global Address Space)プログラミングモデルを採用している．PGAS とは,分散ノード上のメモリ空間を抽象化しつつ，開発者にはデータの局所性を意識したプログラミングを可能にさせるプログラミングモデルである（図1)．MPIのような分散ノード上のメモリに対するデータ転送と計算実行，そのた同期・排他制御を明示的に記述させるような低水準なプログラミングモデルに比べ，それらを高水準に記述できるためプログラムの開発生産性と言った恩恵が得られる．一方，分散ノード上のメモリを透過的に扱う分散共有メモリシステム上でOpenMPやスレッドプログラミングを行う場合とも違い,データ位置を区別し局所性を意識したチューニングも可能である．

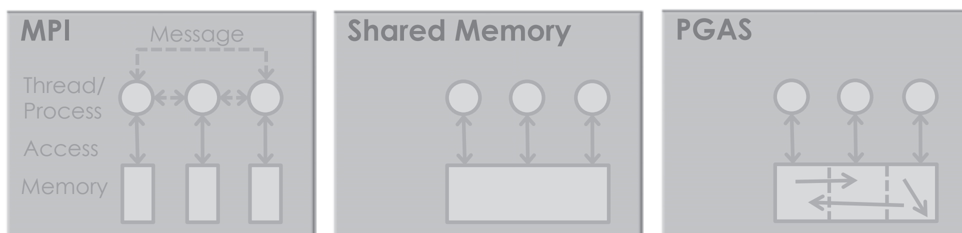


図1: MPI，分散共有メモリ，PGASの比較

X10 のプログラミングモデルは，同様に PGAS モデルを採用している UPC や CAF(Co-Array Fortran)，Chapel に対して，特に同一メモリ上や分散メモリ上での非同期実行に対する言語サポートを強化している点で，APGAS(Asynchronous PGAS)モデルと区別される事もある．

X10 において，計算ノード上の実行プロセスは分散透明なプレースとして抽象化され，各プレースに紐付けられたデータに対するアクティビティの割り当てによって並列分散プログラミングを行う．アクティビティは，非同期的に動作する軽量スレッドを抽象化した X10 の実行単位であり，複数のアクティビティを同一のプレースで並列実行たり，一部のコードブロックを別のプレースで同期・非同期実行させる事もできる．一方，プレース間ではメモリ空間は独立しており，あるプレースに属するオブジェクトは，他のプレースから直接参照する事ができない．そのため，プレース間をまたがったデータアクセスは，オブジェクトの遠隔参照，分散配列や不変データを介して行う．また，複数プレース間でのアクティビティ実行のバリア同期を実現するクロック機能も提供されている．

2.1. X10 言語

X10 言語は，静的に型付けされたクラスベースのオブジェクト指向言語であり，図 2 のように，Java，Scala や C++に強く影響を受けた文法を採用する[3]．

```
1 class HelloWorld {
2     public static def main(args:Rail[String]) {
3         finish
4             for(p in place.places())
5                 at(p)
6                     async
7                         Console.OUT.println(here+" says "+args(0));
8     }
9 }
```

図 2: 分散環境を巡回してメッセージを出力する X10 のコード例

プログラムは典型的なオブジェクト指向言語と同様,フィールドとメソッドを包含したクラス,抽象メソッドを定義したインタフェースに加え，不変データを保持するフィールドとメソッドを包含した構造体によってデータと振る舞いをモジュール化する．図 3 のように，クラスには var 修飾子を付加した変更可能なフィールド(12 行目)と，val 修飾子を付加した不変なフィールド(6 行目)を定義でき,クラスの単一継承とインタフェースの多重実装が可能である.一方,構造体には val を付加した不変フィールド(12 行目)と通常のメソッドを定義できるが，メソッドは継承されない.

```
1 interface Normed {
2     def norm():Double;
3 }
4
```

```
5  class Slider implements Normed {
6     var x : Double = 0;
7     public def norm() = Math.abs(x);
8     public def move(dx:Double) { x += dx; }
9  }
10
11 struct PlanePoint implements Normed {
12    val x : Double; val y:Double;
13    public def this(x:Double, y:Double) {
14       this.x = x; this.y = y;
15    }
16 }
```

図 3: X10 言語でのインタフェース，クラス，構造体のコード例

その他，主な Java 言語との相違点には，メソッドを関数としてファーストクラスで扱え，数値計算によく現れるような配列データの初期化や操作を簡潔に記述できる点や，インタフェースやクラス，構造体に付加したプロパティ（初期化時以外に変更不能なフィールド）を使って型やメソッドに制約を記述できる点，オペレーターオーバーロードや強力な総称型のサポート等が挙げられる．

２．２．X10 言語の並列分散機能

X10 は並列分散プログラミングを強力にサポートする言語機能として，特徴的な構文やデータ構造を提供している．例えば，アクティビティと呼ばれる並列タスクを記述するための async や finish 構文，プレースとして抽象化された計算資源へアクティビティを割り当てるための at 構文，アトミック処理を記述する atomic や when 構文などが提供される．図 4 のように，開発者は並列分散構文を用いて，プレースを指定したリモート実行(2 行目)，async 構文によるその非同期化(4 行目)を容易に記述することができる．さらに，finish 構文を用いて非同期処理のバリア同期を指定できるため，複数プレースを利用した SPMD 型処理も直感的に記述できる(8-10 行目)．一方で，複数のアクティビティ間で共有されるデータに対するアトミック処理も async ブロックへの atomic 指定によって明示できる．さらに，async ブロックを clocked 指定し，next や resume 文を使って複数アクティビティ間での連続的なバリア同期も表せる．

```
1  // リモート実行
2  var value = at(place) eval(arg);
3
4  // 非同期リモート実行
5  at(place) async run(arg);
6
7  // SPMD 型リモート実行
8  finish for (p in Place.places()) {
```

```
9      at(p) async run(arg)
10  }
11
12  // アトミックにリモートデータ更新
13  at(ref) async atomic ref() += value;
```

図 4: X10 言語での並列分散構文のコード例

プレースのメモリ空間は互いに独立しており，直接的なオブジェクトアクセスは制限され，プレース間で暗黙的に同期される事は無い．その代わりに，あらかじめグローバル参照(GlobalRef)に入れたオブジェクトに対しては，明示的に at 構文でプレースを移動してアクセスできるようになる．オブジェクトの同期が抑制されるため，典型的な分散共有メモリシステムで問題となるような複製管理のオーバーヘッドを開発者が制御できる．

分散環境に適した特別なデータ構造としては，分散配列(x10.array.DistArray)と呼ばれるプレース間をまたがった特殊な配列が提供されている．分散配列は通常の配列と同様に扱えるため，複数プレースに分散させた配列データへの処理が，データ分割・転送を意識せずに透過的に記述できる．現時点で，分散配列の各要素とプレース間のマッピングは，プログラマが作成する分散(x10.array.Dist)に従って静的に決定され，ロードに応じた配列データの自動再配置はサポートされていない．

２．３．X10 の実行環境

X10 は，Eclipse Public License のオープンソースプロジェクトとして開発され，C++と Java 環境向けに別々にコンパイラと実行時システムが提供されている．C++向けの Native X10 では，X10 プログラムは C++プログラムへ変換され，ターゲットマシン向けのオブジェクトコードが生成される．一方，Java 向けの Managed X10 では，Java プログラムへ変換され Java 仮想マシンがあれば動作させられる．実行性能を重視したアプリケーションでは Native X10 向けの環境を利用し，既存の Java ライブラリを利用したい場合は Managed X10 を利用するような棲み分けが想定されている．通信レイヤーは，TCP/IP ソケットだけでなく，MPI や PAMI に切り替えたり，また CUDA コードを生成する NVidia GPGPU 向けのコンフィギュレーションも提供されている．

図 5 に示すように，X10 のソースコードを解析して生成された抽象構文木から，C++コード及び Java コードを生成する別々のコンパイラが提供されている．C++及び Java コードへコンパイルされた X10 プログラムは，それぞれ C++と Java で実装された XRX と呼ばれる X10 ランタイムを使って async や finish，at のような APGAS 機能を利用する事になる．アクティブメッセージやコレクティブ通信，RDMA 等の通信レイヤーは X10RT と呼ばれるランタイムトランスポートで C で実装されている．その他の主要な X10 コアライブラリは X10 言語で記述され，Native X10 と Managed X10 のそれぞれ向けに C++と Java へ変換されて利用される．

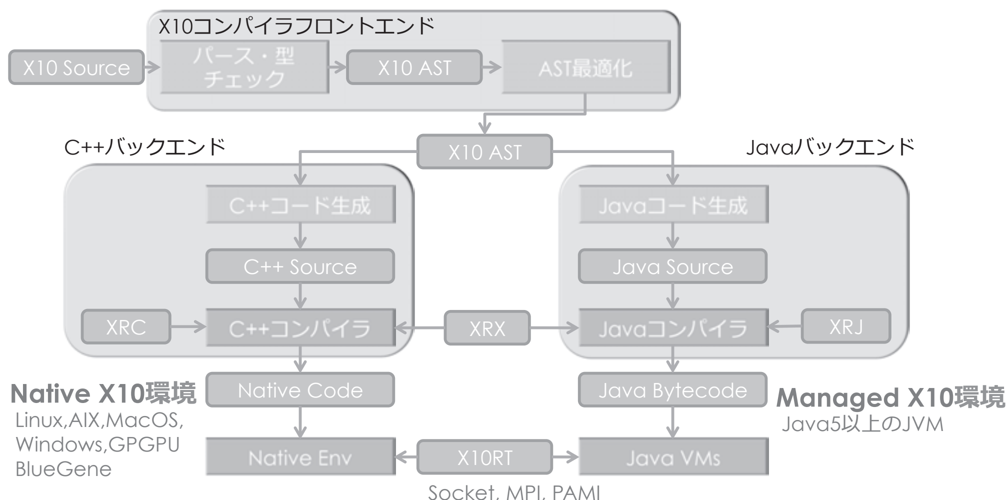


図 5: X10 の実行環境

３．FX10 での課題

東京大学情報基盤センターのスーパーコンピュータ Oakleaf-FX において，X10 を利用するにはいくつかの課題がある．Oakleaf-FX は富士通社が開発した PRIMEHPC FX10（以下 FX10 と呼ぶ）を 4800 ノード（50 ラック）から構成され，理化学研究所の京に搭載されるプロセッサの後継である SPARC64IXfx を採用している．また，計算ノード間が 6 次元メッシュ／トーラスアーキテクチャを採用した Tofu インターコネクトにより接続される．

エンジニアリング上の課題としては，X10 と FX10 は，IBM 社と富士通社が別々に開発したソフトウェアとハードウェアであり，動作サポートが保証される組み合わせでない事が大きい．具体的には， X10 は x86，x86_64 や Power 向けの Linux，Mac，Windows(Cygwin) 向けの Native X10 のみサポートしており，FX10 のアーキテクチャ（Linux/SPARC）向けには実行環境が提供されていない．したがって，XRX や X10RT 等の実行環境を FX10 向けに構築する必要がある．一方，逆に Managed X10 の実行基盤である Java 環境は，FX10 アーキテクチャ向けには提供されていない．そのため，やはり Java 環境を FX10 のアーキテクチャ(Linux/SPARC)向けに構築する必要が出てきてしまう．

さらに別の観点として，シンプルな並列構文と分散処理の抽象化を持つ X10 を利用し，Oakleaf-FX10 の計算性能を十分に活用できるかといった課題もある．例えば，6 次元メッシュ／トーラスアーキテクチャを意識して，X10 のアクティビティのジョブ配置を最適に割り付けられるか，あるいは複数パストランキング，RDMA 通信，Tofu 高機能バリア通信のようなアーキテクチャ固有の機能を利用できるか，が挙げられる．

４．X10 の導入

４．１．導入アプローチ

本課題に取り組むにあたり，まずは X10 を FX10 上に構築する必要があるが，エンジニアリング上の課題があるため，以下の 3 つのアプローチを検討した．

① Native X10 構築

FX10向けのC/C++コンパイラで，X10をソースコードから構築する．Managed X10と比較して，高い実行性能を期待できるが，コンパイラやリンカ等のツールチェインの整合性が問題となる可能性がある．

② Managed X10構築

FX10向けにオープンソースのJava環境を構築する．Native X10と比較して，プログラムの実行性能は劣るが，ScalaやFortressのような別のJVMベースの分散ミドルウェアを活用できるといった副次的な利点もある．一方，もしオープンソースのJava環境にSPARC向けのJust-In-Timeコンパイラが同梱されておらず，X10実行がインタプリタに依拠する場合については，現実的に許容できないオーバーヘッドが発生する恐れもある．

③ ハイブリッドX10開発

当初，①及び②は実現可能性が低いと考え，図6に示すようなNative X10とManaged X10のハイブリッド環境構築を検討した．ハイブリッド環境では，ベース環境としてManaged X10を採用し，分散ノード上で実行されるリモートコードのみがNative X10向けにコード変換される．すなわち，通常のPCサーバをManaged X10を稼働させるクライアント環境としXRXやX10RTを動作させ，アプリケーションプログラム中のatブロックで囲まれたリモートコードをNative X10向けに部分コンパイルする．クライアント環境上で動作するManaged X10は，リモートコードを実行する際に，FX10へリモート実行するように処理をディスパッチする．ディスパッチ処理は，JavaとCのインタフェースであるJNIを利用したり，あるいは別プロセスをフォークしてバッチ処理を起動するようなアプローチが考えられる．当然，C++部分へコンパイルされたX10コードからのX10環境へのアクセス，即ち分散配列，グローバル参照へのアクセスや，アクティビティ生成が必要となる．そのために，Native X10環境の一部（XRX）をFX10上に構築する必要はあるが，その開発コストはX10全体の移植と比較すれば小さい．また，リモートコード内でのデータアクセスをデータフロー解析すれば，XRXを移植せずとも，コード変換によるメソッド引数の追加で代用できる可能性もある．しかし，Oakleaf-FXでは，Managed X10環境の動作可能なログインノードとFX10計算ノードへの通信が，FJMPIのバッチスケジューラに一元管理されているため，リモートコードへのディスパッチはバッチ処理を経由する必要がある．計算ノードとのプロセス間通信が制限されるため，リモートコードとのデータ交換についても，メモリをストレージへ永続化する手間が発生する．

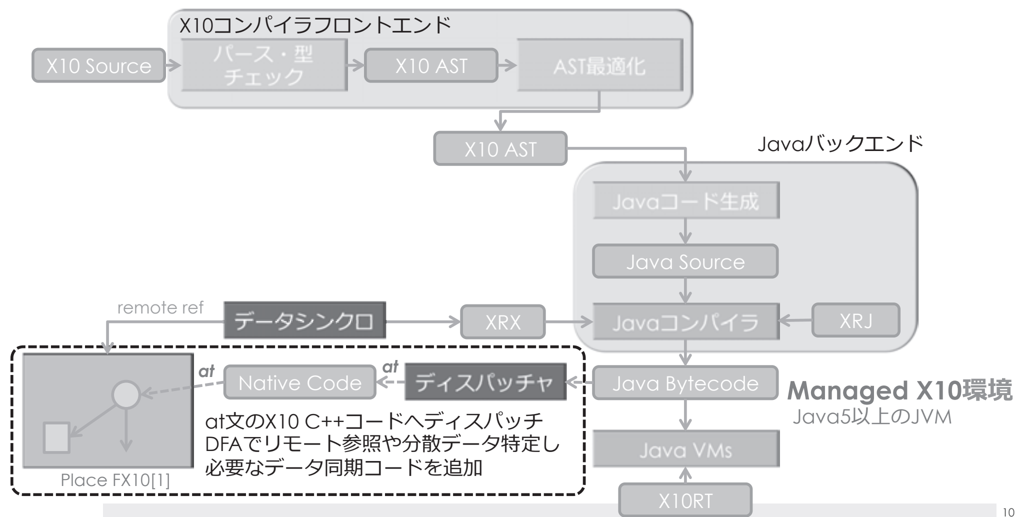


図 6: ハイブリッド X10 環境

本課題では, 3つのアプローチの内, まず①及び②に取り組み, 実現可能性と実現コストのバランスに鑑み③に取り組む事にした.

4.2. 導入状況

現在までに, FX10 上での①Native X10 構築及び②Managed X10 構築が完了した. 具体的には, 下記の成果が挙げられる.

- X10 2.4.0[1] C++環境(Native X10)の構築及び動作確認
- X10 2.4.0 Managed 環境(Managed X10)の構築及び動作確認
  - OpenJDK 1.6.0_24 インタプリタ及び Server/Client JIT コンパイラ移植及び動作確認
- Tofu 上での OpenMPI Java Bindings 1.7 の構築及び動作確認

つまり, Oakleaf-FX スーパーコンピュータ上では, X10 環境が利用できるだけでなく, Java 環境も利用できるようになった. また, OpenMPI Java Bindings のような, 既存の Java ライブラリを活用した Oakleaf-FX の利用も可能となっている. 既に我々は, X10 環境だけでなく Java 及び OpenMPI Java Bindings 環境を利用して, 既に HPC 向けの単体テストツールの研究, 並列コレクション向けコンパイラや並列グラフ解析向けコンパイラの研究に着手している.

5. 性能評価

本章では, FX10 及び Oakleaf-FX 上での, Java 及び X10 環境の予備的な性能評価実験の結果について述べる. 性能評価実験は, FX10 上での Java 環境のマイクロベンチマークと, 複数台の FX10 を利用した X10 PERCS ベンチマークを行った. また, OpenMPI Java Bindings についても簡単な性能評価実験を行った.

5.1. Java ベンチマーク

Java 環境の性能評価は, 以下 3 つのベンチマークアプリケーションを用いて行った. ①はシ

[1] 2013/10 時点の最新版

[2] http://www.benchmarkhq.ru/cm30/

ングルスレッドでのマイクロベンチマーク，②は実アプリケーションを想定したマルチスレッドでの実行性能を評価するために利用した．

① CaffeineMark 3.0[2]
- Sieve：素数計算
- Loop：ループ処理
- Logic：論理計算
- String：文字列操作
- Method：メソッド再帰呼び出し
- Float：3次元回転による浮動小数点演算

② SPECjvm2008[3]
- compiler：コンパイラ実行
- compress：改良LZWアルゴリズムでのデータ圧縮
- crypto：AES，DES，RSA，MD5，SHA1，SHA256等での暗号・複合化
- derby：JavaのDatabase実装に対する基本的なDB操作
- mpegaudio：mp3デコード
- scimark：FFT，LU，Monte Carlo，SOR，SPARSE等を用いた浮動小数点演算
- serial：オブジェクト直列化・非直列化
- startup：Java仮想マシンの起動
- sunflow：グラフィックスレンダリング
- xml：XML変換及びバリデーション

また，実験環境はFX10計算ノード1台と，比較のためにログインノード1台を利用した．それぞれのハードウェア及びソフトウェアは以下のとおりである．

- FX10計算ノード
  SPARC64IXfx 1.848GHz 12コア，メモリ32GB
  XTC OSカーネル2.6.26.8
  OpenJDK 1.6.0_24-b12
- ログインノード
  Intel Xeon L5640 2.27GHz 6コアx2機，メモリ48GB
  Red Hat Enterprise Linux 6.1 カーネル2.6.32
  OpenJDK 1.6.0_24-b24

図7に①CaffeineMarkの結果を示す．横軸はCaffeineMarkに含まれるプログラム（最右は全体の平均），縦軸は対数表記で表したベンチマークスコアを表す．比較のために，FX10 上では，GNU GCJ 4.4.7 の Java インタプリタと，移植した OpenJDK のインタプリタ実行，C2 JIT実行の結果を併記する．結果より，FX10上のJavaバイトコードのインタプリタ実行は， Intel及びSPARC64IXfx上のJITコンパイラ実行の性能差と比較して極端に悪くなる事が分かる．

[2] http://www.benchmarkhq.ru/cm30/
[3] http://www.spec.org/jvm2008/

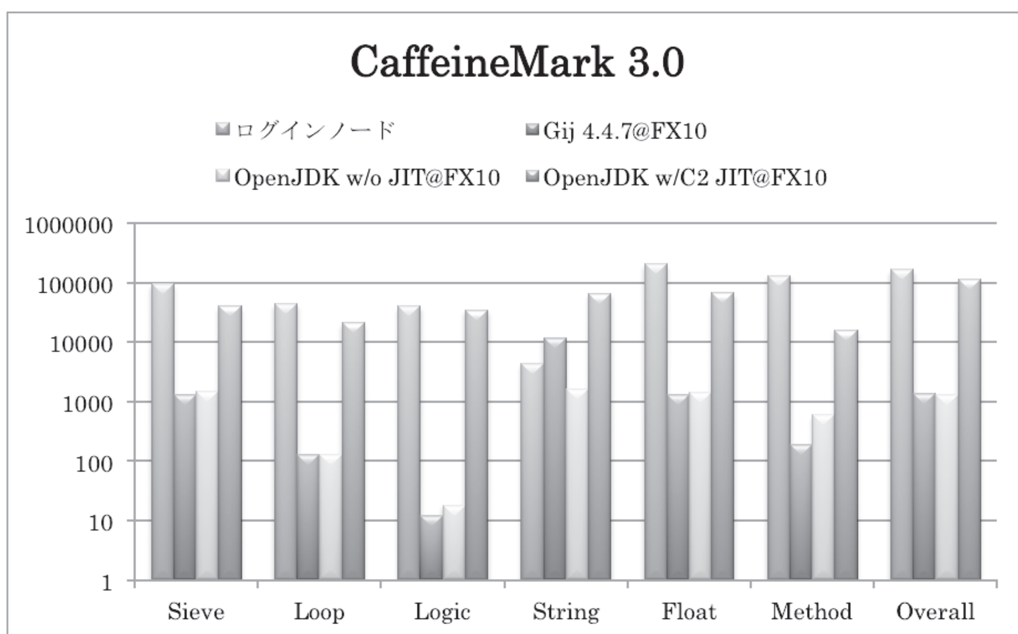


図 7: CaffeineMark ベンチマーク（ログインノード及び FX10 計算ノードでの各 OpenJDK のベンチマークスコア）

図 8 に，Intel 及び SPARC64IXfx 上の JIT コンパイラ実行の結果のみ抜き出した結果を示す．また，比較のために，ベンチマークスコアをクロック周波数当りに正規化した結果を図 9 に示す．ログインノード上での OpenJDK は，FX10 上より全体平均で約 46.5%高いスコアが測定された．特に，浮動小数点演算では 208.1%，メソッド呼び出しでは 726.6%ものスコア差が出ている．一方で，文字列操作に関しては FX10 上の OpenJDK が，32.8%高いスコアが測定された．一方，クロック周波数当りのスコアでは，全体平均で 19.2%のスコア差となる．ほぼ同一バージョンの OpenJDK を用いている事から，主な性能差は JIT コンパイラが生成するアセンブリコードが，SPARC64IXfx において浮動小数点演算用の命令を呼び出せてない可能性がある．また，メソッド呼び出しの性能が FX10 で悪い原因としては，同様に JIT コンパイラが十分に機能しておらず，例えば末尾再帰の最適化など基本的な最適化が施されていない可能性がある．

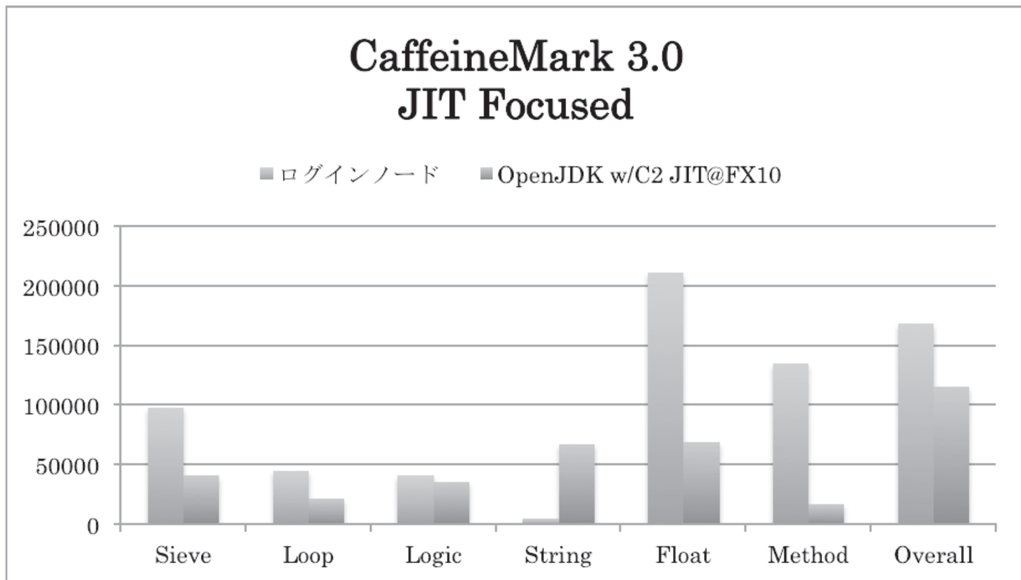


図 8：CaffeineMark ベンチマーク（Intel 及び FX10 上でのベンチマークスコア）

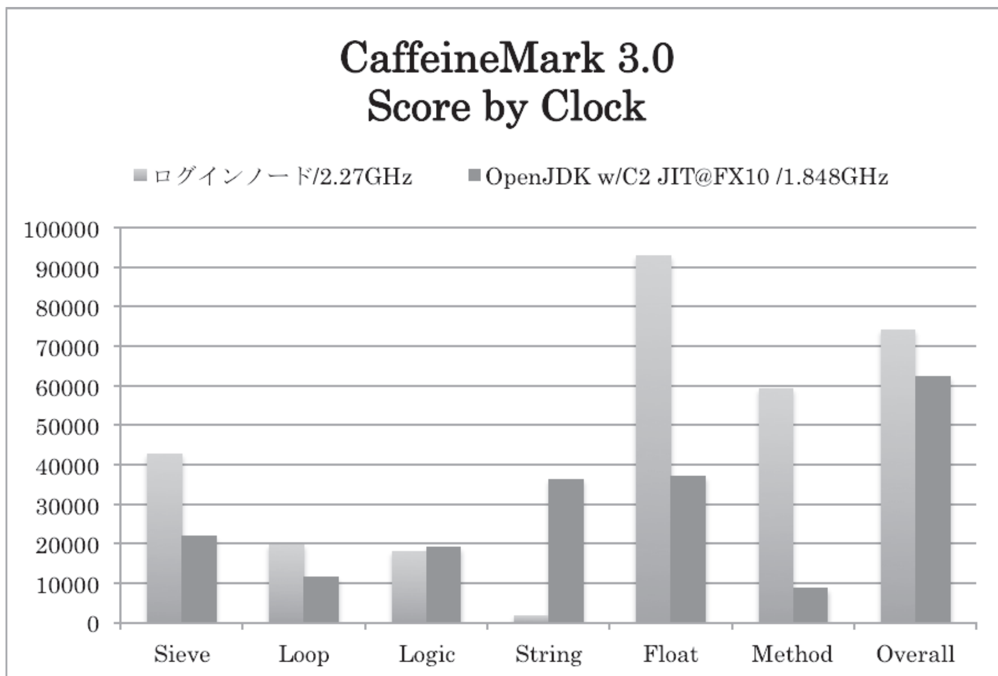


図 8：CaffeineMark ベンチマーク（Intel 及び FX10 上での
クロック当りのベンチマークスコア）

次に，SPECjvm2008 の実験結果を図 9 左に示す（最下段はスコアの全体平均）．SPECjvm2008 は，実践的なマルチスレッドアプリケーションから構成されるため，測定は Intel 及び FX10 のそれぞれの論理プロセッサ数と同じスレッド数で実行した結果を求めた．そのため，図 9 右には，図 8 と同様にクロック周波数当りに修正したベンチマークスコアを 1 スレッド毎に直した結果を示す．ログインノード上での実行は，全体平均で 199.35%の高いスコアが得られた．

一方，1スレッドでクロック周波数当りに正規化した場合，全体平均では33.4%の性能差となった．これは，CaffeineMark 3.0 での全体平均の性能差とほぼ同等である．この結果から，SPARC64IXfx 向けの JIT コンパイラの浮動小数点演算及びメソッド呼び出しの最適化を Intel 向けの JIT コンパイラ並に改善すれば，現実的なマルチスレッドアプリケーションでもアーキテクチャの違いに関わらず一定のクロック当りの性能が得られる事が予想できる．

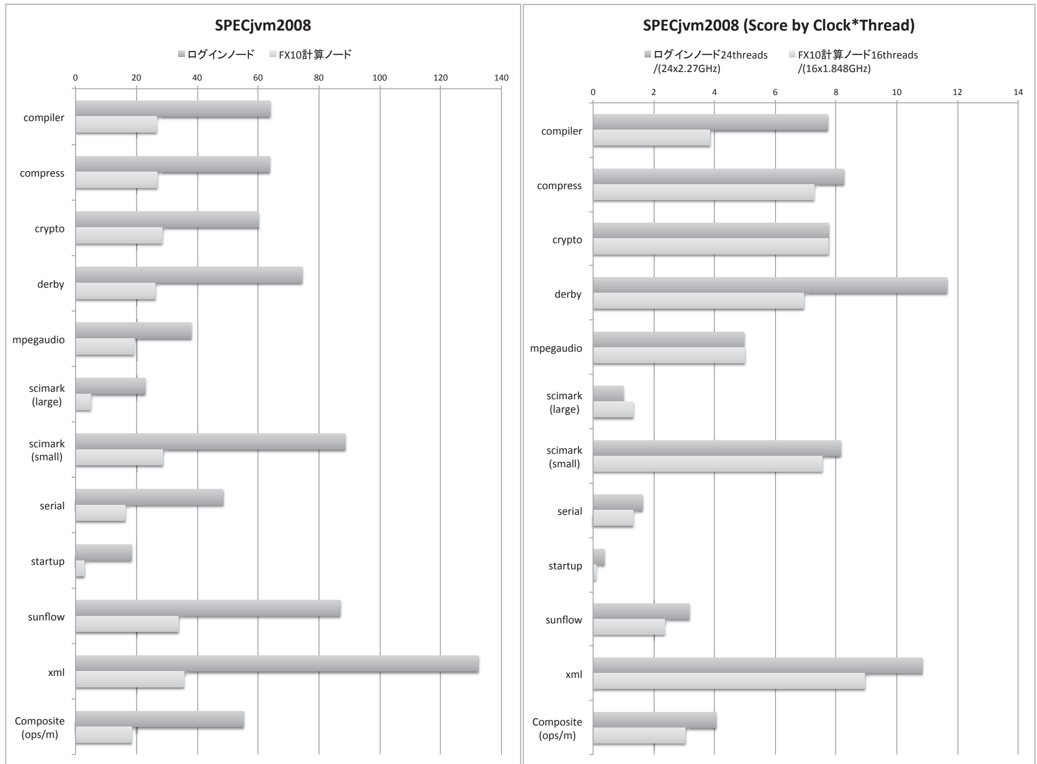


図 9：SPECjvm2008 ベンチマーク（Intel 及び FX10 上でのベンチマークスコア，
及び 1 スレッドのクロック周波数当りベンチマークスコア）

５．２．X10 ベンチマーク

X10 の性能評価には，X10 PERCS Benchmarks[4]を利用した．PERCS Benchmarks には，HPC Class II Challenge 向けに開発された X10 の実装であり，以下に示す規定の 4 つのベンチマークに加えて，5 つのベンチマークが含まれる．

- HPC Class II Challenge 向け
  - Global HPL：連立一次方程式の演算速度
  - Global Random Access：メモリのランダムアクセス速度
  - Global FFT：フーリエ変換による浮動小数点演算速度
  - EP Stream Triad：負荷時のメモリアクセス速度
  - UTS：非均一なツリー探索
- その他のベンチマーク
  - SSCA1：パターンマッチ

➢ SSCA2：不規則なグラフ探索

➢ KMEANS：グラフのクラスタリング

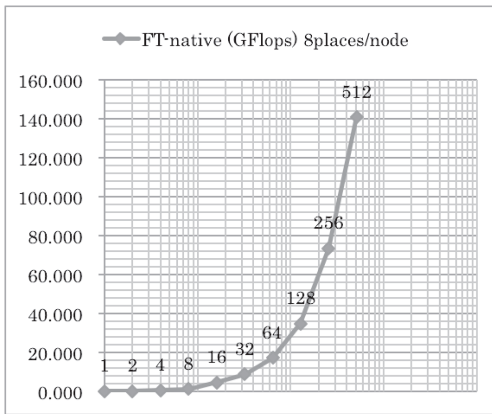
これらの内，HPL，Random Access，FFT は部分的に@Native アノテーションを使って C++で実装された関数を呼び出しているため，Native X10 向けの実装のみ提供されており，Managed X10 向けには Stream，UTS，SSCA1，SSCA2，KMEANS のみが提供されている．しかも，これらのベンチマークについても，Managed X10 向けに最適化が施されている訳ではない．そのため，本実験では，主に X10 ベンチマークが FX10 上でも動作可能であり，ある程度台数効果が得られるかを確認する事が主目的となる．また，Managed X10 は公式には MPI をサポートしていないため，本実験は，X10 2.4.0 に対して，Oakleaf-FX での MPI 実装（FJMPI）を利用できるようにビルドし直して行った．

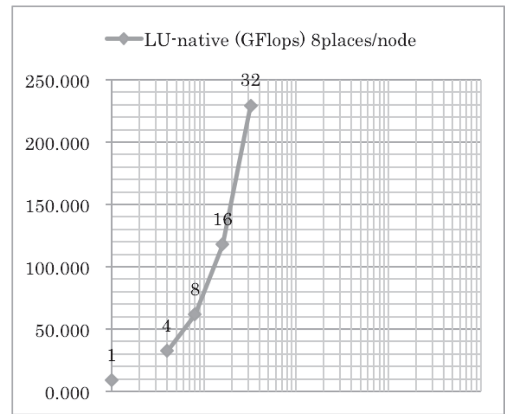
実験には，FX10 計算ノードを 1 台から 512 台まで使い，作成する X10 のプレースを 1 プレースから 1024 プレースまで変更しながら行った．FX10 は，16 コアのプロセッサコアを搭載しているため，実験では 1 計算ノード当り最大 16 プレースを作成し，1 プレース当りのアクティビティ数は 1 スレッドとした．ただし，FFT，HPL，Random Access，UTS については，メモリ不足を回避するよう 1 ノード当り 8 プレースとした．各ベンチマークプログラムのコンパイルオプションには，Native X10 向けは -NO_CHECKS -optimize -FLATTEN_EXPRESSIONS=true -x10rt mpi -cxx-prearg -Xg -cxx-prearg -w -cxx-prearg -Kfast -cxx-prearg -DENABLE_CROSS_COMPILE_FX10 -cxx-postarg -lpthread を付加し，Managed X10 向けには -NOCHECKS -optimize を付加した．また，LU 以外の実行時環境変数に X10RT_MPI_ENABLE_COLLECTIVES=true，FFT と KMeans に X10_CONGRUENT_BASE=0x10000000000LL をセットして実行した．

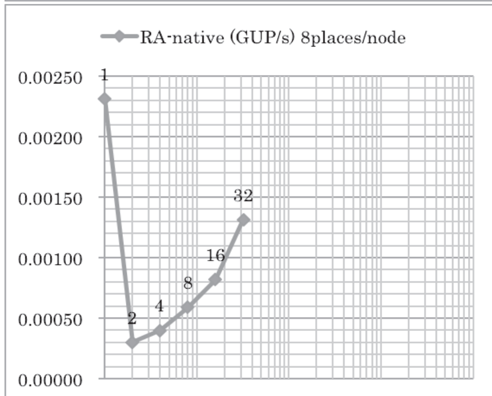
まず，HPC Class II Challenge 向けベンチマークの結果を図 10 に示し，以下にプレース数を増加させた際のベンチマークスコアの変化のグラフを載せる．異常終了や 48 時間を超過するようなケースは，計測不可としている．結果より，Native X10 での FFT，LU，RA，Stream，UTS は，プレース数にスケールしている事が分かる．Managed X10 では，Stream はプレース数にスケールし，かつ Native X10 よりも高いスコアが得られたが，UTS は並列分散の効果が得られなかった．また，LU と Random Access では，64 プレース以上ではプログラムが終了せず，正しくプログラムが動作していない可能性もある．

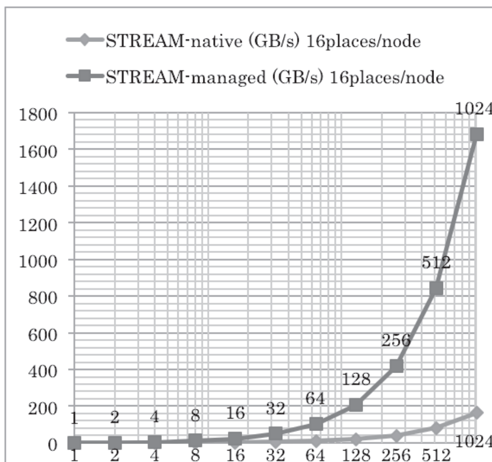
ベンチマークスコア		1	2	4	8	16	32	64	128	256	512	1024
FT-native (GFlops)	8places/node	0.17911	0.3591	0.71868	1.42998	4.61971	8.82557	17.2616	34.9952	73.4877	141	–
LU-native (GFlops)	8places/node	9.21078	–	32.6596	61.7951	118.283	229.394	–	–	–	–	–
RA-native (GUP/s)	8places/node	0.00231	0.0003	0.0004	0.00059	0.00082	0.00131	–	–	–	–	–
UTS-native (M nodes/s)	8places/node	3.5362	7.0625	14.033	27.884	55.603	111.1	221.71	440.61	869.55	1699.7	3247
UTS-managed (M nodes/s)	8places/node	0.0014	6.1777	3.5936	1.2413	1.708	1.7774	8.1568	1.1056	2.0248	2.9943	–
STREAM-native (GB/s)	16places/node	0.17546	0.3178	0.62945	1.23996	2.59029	5.16554	10.3244	20.6183	41.2893	82.305	165.045
STREAM-managed (GB/s)	16places/node	1.73566	3.46431	6.91922	13.6267	26.4125	52.6912	105.934	210.852	422.158	845.894	1686.24

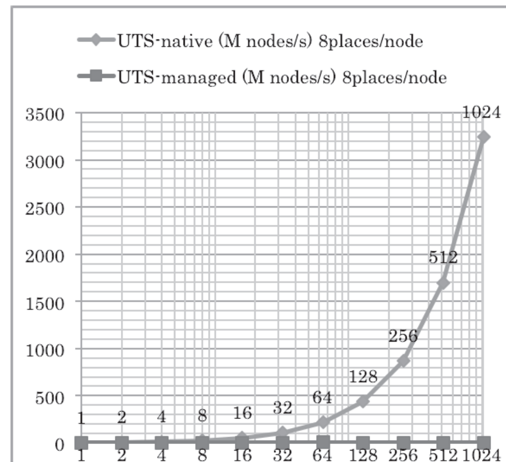
図 10：HPC Class II Challenge 向けベンチマークの結果











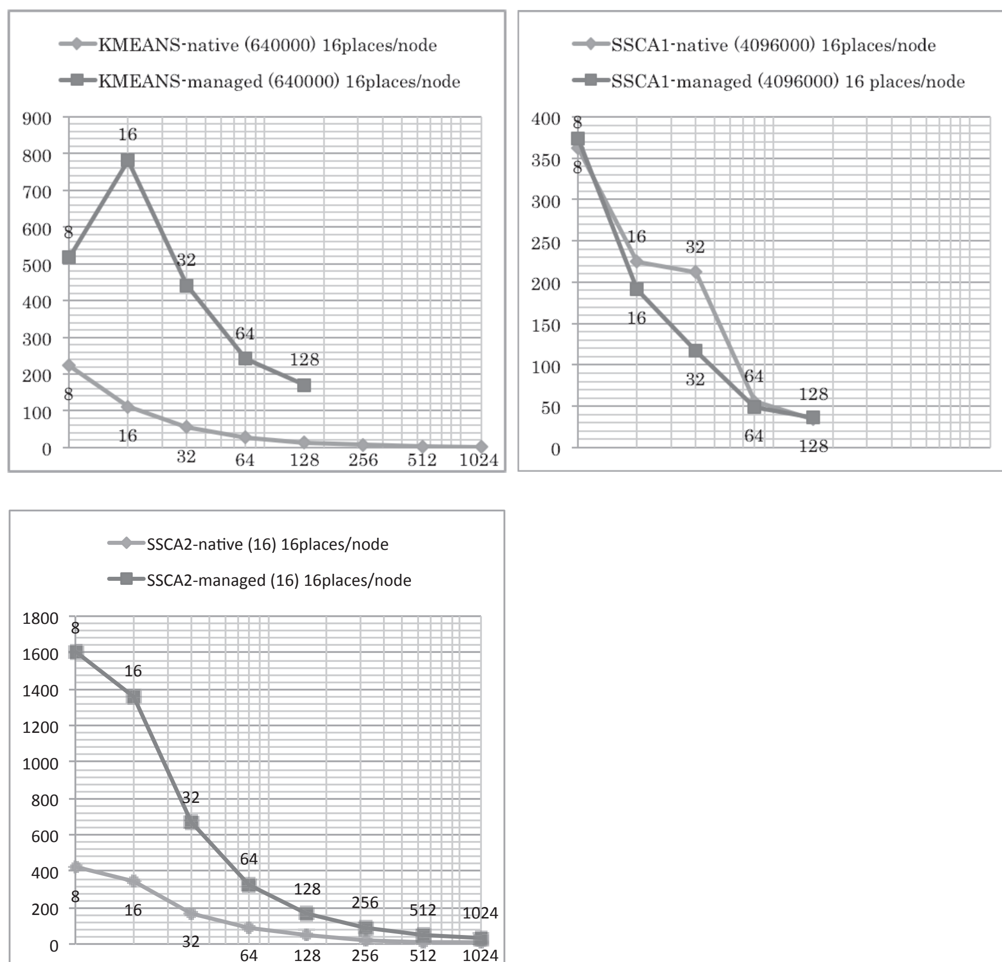
次に，X10 PERCS Benchmarks に含まれる，その他のベンチマークの結果を図 11 に示し，以下にプレース数を増加させた際のベンチマークスコアの変化のグラフを載せる．KMeans, SSCA1, SSCA2 は，ベンチマークスコアの計算式が提供されないため，KMeans では点の数を 640,000, SSCA1 では問題の長さを 4,096,000，SSCA2 では頂点数を 2 の 16 乗，に固定した際の実行時間を測定した．48 時間を超過するケースや，問題サイズが小さ過ぎるケースは計測不可としている．

結果より，Native X10 及び Managed X10 において，プレース数に対して実行時間が減少している事が分かる．しかし，Managed X10 では，KMeans では 256 プレース以上，SSCA2 では 1024

プレースで計算が終了せず，正しくプログラムが動作していない可能性もある．

問題サイズ固定時の実行時間		8	16	32	64	128	256	512	1024
KMEANS-native (640000)	16places/node	224.139	112.782	56.4248	28.3655	14.3585	7.51583	3.99605	2.22658
KMEANS-managed (640000)		519.114	783.562	442.056	243.385	171.641	–	–	–
SSCA1-native (4096000)		363.104	225.35	212.904	56.2486	34.8475	Too small	Too small	Too small
SSCA1-managed (4096000)		374.754	191.934	117.98	49.3966	36.8376	Too small	Too small	Too small
SSCA2-native (16)		420.124	343.941	170.872	88.0621	46.1907	24.677	14.1433	8.9248
SSCA2-managed (16)		1600.06	1363.7	666.943	330.034	164.166	90.9827	49.4603	34.0714

図 11: その他のベンチマークの結果



6． まとめと今後の課題

本課題の成果としては，東大情報基盤センターOakleaf-FXに，Java環境及びX10環境を移植し，予備的な実験から実用性を評価した事である．また，OpenMPI Java Bindings も移植し，

MPJ Benchmarks での簡単な評価も実施している.

特に，移植した Java 及び OpenMPI Java Bindings を利用すれば，スーパーコンピュータ向けの研究開発の迅速なプロトタイピングが可能となると期待される．実際に我々は，ステンシル計算用の Java 言語拡張及び並列 Java コレクション向けコンパイラ開発[5]，HPC アプリケーション向けの単体テストツールのプロトタイプ実装[6],並列グラフ解析向けのフレームワークのプロトタイプ実装，等々，既にいくつかの研究プロジエクトを開始している.

X10 に関しては，今回十分な性能評価を行えなかったが，ただし，Native X10 では一定の台数効果を確認でき，ネィティブインタフェースを利用すれば，FX10 に固有の最適化機能も呼び出せる可能性を確認した．一方，Managed X10 に関しては，MPI に非対応である事や，Java 以外のネィティブライブラリを呼び出せない事等，実用上のいくつかの課題が存在する事も確認した．一方で，並列分散アプリケーションのスケーラビリティと実行性能，開発生産性を両立する X10 環境は，社会シミュレーションやバイオインフォマティクスのように，実行性能よりも開発生産性やスケーラビリティが重要視される分野のアプリケーションに対して，スーパーコンピュータの活用機会を開く道具となる事が期待される.

参 考 文 献

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object- oriented approach to non-uniform cluster computing. In OOPSLA, 2005.

[2] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, K. Yelick, S. Alam, R. Campbell, L. Carrington, T.-Y. Chen, O. Khalili, J. Meredith, and M. Tikir. DARPA's HPCS Program: History, Models, Tools, Languages. In M. V. Zelkowitz, editor, Advances in COMPUTERS High Perfor- mance Computing, volume 72 of Advances in Computers, pages 1 - 100. Elsevier, 2008.

[3] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, X10 Language Specification Version 2.3, 2012.

[4] O. Tardieu, D. Grove, B. Bloom, D. Cunningham, B. Herta, P. Kambadur, et al. X10 for productivity and performance at scale. A Submission to the 2012 HPC Class II Challenge, 2012.

[5] 宗桜子，佐藤芳樹，千葉滋，処理の差異と順序を考慮した並列コレクション向け Java 言語拡張，並列／分散／協調処理に関する『北九州』サマー・ワークショップ，2013

[6] 穂積俊平，佐藤芳樹，千葉滋，HPC アプリケーション向け計算網羅性・計算順序をテストするツール，並列／分散／協調処理に関する『北九州』サマー・ワークショップ，2013