2007－OS－106 (10)
2007／8／3

揮発性資源上での並列分散計算を支援するオブジェクト指向ライブラリ

弘中 健[†] 澤井省吾[†] 田浦健次朗[†]

本研究では実行時に動的に計算資源が変化するような環境で有効な，分散オブジェクト指向ライブラリを設計し，これを用いて簡潔な記述で大規模な計算資源で処理を並列するフレームワークを実装した．近年計算資源が着実に増えているが，決まった資源を継続的に使えることは少なく，実行時に追加・削除が出来ることが求められている．このような環境で並列分散計算を手軽に行うライブラリは稀である．既存の多くの実装は各拠点へのインストール，複雑な設定ファイルの記述などを必要とする．その中，大量なデータファイルに対して大規模な計算資源で並列に処理をしたいという要望が多い．このような要望に対し，従来のライブラリでは敷居が高く，生産性も低い．

我々は Python スクリプト言語に分散オブジェクトライブラリ拡張を施した．このライブラリはインストール，設定ファイルの記述などは全く必要とせず，スクリプト言語で透過的に分散環境での計算を可能にする．また，このライブラリを用いて大量のファイルを入力に取り，並列に実行する処理の記述を容易にするようなフレームワークを実装した．このフレームワークでは，ユーザーは希望する処理のみを記述し，ネットワーク，資源の増減などの煩雑さから開放することで高い生産性を与えることが出来る．このフレームワークの有用性を実アプリケーションを用い，評価を行った．

A Distributed Object-Oriented Library for Computation Across Volatile Resources

KEN HIRONAKA,[†] SHOGO SAWAI[†] and KENJIRO TAURA[†]

We propose a distributed object-oriented library that allows computation across computing resources in which node join and leave dynamically. Using this library, we implemented a framework that parallelizes job processing in the presence of joining and leaving nodes. Existing libraries that enable distributed computation requires node-based installation as well as preparing complex configuration files. This is a large overhead for application writers who want to parallelize or distribute computation. Additionally, we have witnessed an increasing demand for frameworks that allow easy parallelization of jobs that process an enormous amount of data. It is important that such applications can be written on-the-fly with very little programming and configuring.

We implemented our library as an extension to Python, a mainstream script language. Our library does not require any installation or configuration files, and allows deployment across large resources with consideration of joining and leaving nodes. In the framework implemented on top of our library, jobs that take large data files may be parallelized with minimal coding, alleviating the user from worries with respect to network programming, and dynamic changes in resources. We evaluated this framework using a real-life application to show its effectiveness.

1. はじめに

近年，数百台規模のクラスタ計算資源が多くの拠点で構築されている．例えば東京工業大学には，TOP500 に名を連ねる 655 ノード，10,480CPU コアのシステム，TSUBAME がある．[17] また，InTrigger プラットフォームは五年間かけて，日本各地の拠点にクラスタ資源を展開する計画を進めている．現在でも 400 ノード以上の資源が稼動している．[9] 今後，これらの計算資源を用いた多分野に及んで大規模計算の促進に繋がることは必死だ．しかし，計算資源が大規模化することにより，計算資源の不均一化がより増している．特に大量の計算資源を活用する場合，すべての計算資源を同時に使用出来ることは稀である．ノードの故障は勿論，計算資源を動的に追加，削除出来ることも迫れれる．計算資源が大規模化することにより，このようなノードの「揮発性」を考慮して使用しなくてはならない．

このような環境で継続的に大規模に計算をすることは困難であり，それを容易にするアプリケーション支援ライブラリも少ない．特にアプリケーションを書く立場の人はネットワーク，並列化に関する記述は極力避けたい上，アプリケーションを実行する手間も多く削減したい．具体的には，分散環境で動作させるために特別のソフトのコンパイル，インストールが全く on the fly に使用出来，高級言語での記述で生産性を高めた支援ソフトが望ましい．この観点からコンパイルの必要がないスクリプト言語での並列・分散支援ライブラリの需要が大きい．

同時に，大規模な計算資源を必要とする応用は近年急速に増加している．インターネットを通じて大量の文章が取得出来るようになったことで，自然言語処理を研究する者からは，

† 東京大学情報理工学系研究科
University of Tokyo: Graduate School of Information Science and Technology

言語解析を並列に実行したいという要望がある．また，バイオインフォーマティックスの分野では大量な遺伝子配列を対象に，進化的な配列の類似性，たんぱく質機能の類似性を解析したいという要望もある．このような要望を受けて処理を並列化する場合，分断された入力ファイルは大量にあり，computationally-intensive な処理であり，データ依存は入力ファイル以外にないという傾向がある．また，ある処理が出力するファイルが次の処理の入力となることもある．このような計算のモデルは単純であるが，大規模な計算でかつ，動的に資源が増減する場合は以前として実現は難しい．

これらを踏まえ，本研究では，スクリプト言語でグリッドコンピューティングが出来る分散オブジェクト指向ライブラリ拡張を設計した．また，このライブラリを用い，揮発的な大規模計算資源上で走るアプリケーションの記述を容易にするフレームワークを実装した．アプリケーション記述者は数行のコードを記述することで，希望する処理が必要な入力ファイルを自動的にファイル転送しながら計算資源上で並列に実行される．また，動的に資源を投入することが出来，途中に一部の資源で削除，故障が起きても対応することが出来る．

本論文の構成は以下のようにする．2 章で関連研究を紹介し，3 章で設計した分散オブジェクト指向ライブラリの概要を説明する．4 章ではこれを用いたフレームワークの説明をし，5 章で評価，6 章でまとめる．

2. 関連研究

2.1 分散オブジェクト指向ライブラリ

アプリケーションの記述を容易にするためには，複数のクラスタに跨った大規模計算を支援する分散オブジェクト指向ライブラリが有用だと考えられる．また，このようなライブラリは計算資源を動的に追加，削除を支援することも大切である．

2.1.1 JavaParty

JavaParty はクラスタ上の複数のノードで分散オブジェクト指向プログラミングが出来る Java の拡張である．[10] この拡張では，遠隔オブジェクトへの参照，メソッドコールを通常の Java オブジェクトと同様に，透過的に見せることが出来る．[14] また，既存の JavaRMI ではなく，KaRMI[11] という高性能通信用 RMI ライブラリを用いている．KaRMI はどの通信プロトコルを使うか，また TCP プロトコルの場合はどのポートに listen するか，などを設定ファイルに記述する必要がある．また，分散オブジェクトを実装する上で，Java コンパイラに変更を加えているため，コードを変更するたびに際コンパイル，各ノードに配布というサイクルを踏む必要がある．これはディバッギングをしながら開発するユーザーにとってはオーバーヘッドが大きい．

2.1.2 ProActive

ProActive も JavaParty と同様に大規模な計算資源をターゲットとした Java の分散オブジェクト指向拡張で，1000 台規模での実行の実績もある．[8] 計算資源の構成 (各ノードのホスト名，アクセス方法) を複雑な設定ファイルに予め記述することが必要である．そのため，資源を動的に追加，削除は不可能である．また，ユーザーに設定ファイルの記述を科すことはほぼ不可能である．

2.1.3 スクリプト言語の分散オブジェクト拡張

手軽に分散オブジェクト指向プログラムを書くことを目標とし，スクリプト言語にライブラリ拡張を施した実装もある．Ruby には druby[3] というライブラリが，また Python には pyro[15] という実装が存在する．これらのライブラリを用いると，ユーザーは数行で RMI を利用出来る．容易に分散オブジェクトプログラムが記述出来るという目的に関しては本研究と同様である．しかし残念ながら，これらの実装はクラスタ上計算，ましてや大規模計算資源を想定した実装ではない．

2.2 ワークフローモデル

簡潔な記述で大量のデータの処理を並列分散するフレームワークは，大規模計算資源を使用したいユーザーによって極めて重要である．また，資源を動的に追加，削除出来ることは資源の大規模化を踏まえると必死である．

2.2.1 Hadoop: Map-Reduce Framework

Hadoop[7] は Map-Reduce モデルを実装したオープンソースの Java のライブラリである．Map-Reduce というモデルは関数型言語に存在する map，reduce の処理を手続き型言語に導入したものである．[2] ユーザーは個別に map，reduce という関数を定義することで自分の行いたい処理を記述する．処理は大きく分けて 2 段階に分けられる．まず，map 関数にユーザーが定義する入力データが代入され，データが出力される．そのデータが reduce 関数の入力なり，ユーザーが求める出力が得られる．map，reduce にはそれぞれ大量の入力データが入るが，map,reduce はそれぞれの処理を並列に実行する．処理の並列化はデータ複数の断片に切断し，複数の計算資源に分散させることにより実現される．

しかし，Hadoop を実行するためには，XML の設定ファイルを記述する必要がある．また，すべてのノードで同じパスでライブラリがアクセス出来るようにする，環境変数の設定をする，実行・停止に特別なスクリプトを実行する，などの制約がある．これはアプリケーションユーザーに取ってまだ敷居が高い．また，計算資源はスクリプトを用いた一斉に投入・停止を基本としており，動的にノードを追加することは想定していない．

2.2.2 Martlet

Martlet は並列分散処理用の関数型言語である．[6] Map-Reduce と同様に，map，fold(reduce)，tree など，関数型言語ではお馴染みの関数で，入力の評価を自動並列するものである．今開発中のため，実装の詳細はまだ明らかではない．

2.2.3 UIMA GRID

UIMA は大量のテキストを解析するためのフレームワークである．[4] テキストの処理は多ステージに分けて行われ，ステージ間のインターフェイスを規定することで，各ステージの処理をユーザーが記述し，必要に応じて入れ替えること

が出来る．テキスト解析はUIMA jobとして小さくラップされ，複数の計算資源で並列化される．並列化にはCondorを用いており，バッチキューを用いてジョブが実行される．Condorを使うことにより，ノードの動的追加，削除が可能である．しかし，評価では4台と非常に小さい環境で行われている上，この環境で既にスケーラビリティが低下している．

3. Pythonの分散オブジェクト拡張

本研究では，大規模な計算資源を対象とした分散オブジェクト拡張ライブラリをPythonスクリプト言語で実装した．このライブラリでは，リモートオブジェクトに対するメソッドコールも，通常のPythonオブジェクトへのものと等価に見せることをしている．また，大規模な計算資源を用いる場合問題となる，接続のマネージメント，NATやfirewallへの対応もしている．さらに，通常計算ノード間をTCP接続で繋ぐ場合，互いのIP,portを取得するという作業が煩雑に成りかねないが，設定ファイルの記述などに頼らない提案もしている．

3.1 RMI: Remote Method Invocation

我々の実装したライブラリでは，リモートオブジェクトの物理的位置は透過的に見せている．つまりオブジェクトにメソッドコールでアクセスする場合，呼び出し側と呼ばれるオブジェクトの位置関係に関係なく，通常のメソッド呼び出しとすることが出来る．同期的RMIは勿論，非同期RMIも可能である．

3.2 リモートレファレンスの取得

遠隔にあるオブジェクトにアクセスするには，それに対する参照を取得する必要がある．ライブラリではその方法が幾つかある．参照はメソッドコールの引数として渡された場合，そのメソッド内ではその参照が指すリモートオブジェクトをアクセスすることが出来る．

```
#get remote ref as argument
def do_remote_fib(ro, N):
   value = ro.fib(N) #do RMI
```

また，リモートオブジェクトが自発的に自分へのレファレンスを外部公開することが出来る．具体的には，リモートオブジェクトは自身に任意の文字列の名前で外部公開する．

ある別のノードのPythonプログラムは，このオブジェクトへのレファレンスを取得したい場合，登録されている名前で接続されているノードのネットワークで検索をかけ，その名前に対応するリモートオブジェクトへの参照を得る．この例を図1実行時にこのように参照を得る仕組みがあることは，追加された資源が計算に参加するためには欠かせない条件である．

3.3 オブジェクトマイグレーション

各リモートオブジェクトの実体はネットワーク内のあるノードのPythonプロセス上に存在することになる．ただし，オブジェクトの物理的位置が固定されるということは大きな制約になることがある．例えば，あるノードのPythonプロセスが頻繁に遠隔のオブジェクトにアクセスする場合，通信遅延が非常に大きくなる．ここで，そのオブジェクトと手元のノードに移動させることが出来れば，計算の効率は多く向上する．我々のライブラリでは，リモートオブジェクトを動的に再配置することを許している．具体的には，以下の構文を用いる．

```
# AT REGISTERING PROGRAM
# make ro accessible with public name
ro.register("remote_foo")

# AT RETRIEVING PROGRAM
#search for reference
#with this public name
ro = RemoteRef("remote_foo")

#rmi using the obtained reference
ro.foo()
```

図1 リモートレファレンスの登録・取得の例
Fig. 1 Example code of a remote reference being registered and retrieved

```
#ro located at some node
ro.move(pid) #globally unique process id
#ro is now located at node: pid
```

オブジェクトマイグレーションは，今後脱退するノードを考慮する上で，必須の条件ということが言える．

3.4 オーバーレイの構築・ルーティング

大規模な計算資源を用いる場合，すべてのノードが互いに接続を張ることは望ましくなく，実際不可能のことも多い．計算に参加する台数Nに対して，一般に$O(N^2)$のコネクションを必要とするので，メモリに大きな負荷をかける上，ノード間のL2スイッチのセッションテーブルを溢れさせることすらある．また，一部private network，firewallを用いるクラスタをある場合，全対全の接続は物理的に不可能である．

そのため，我々のライブラリでは，計算に参加しているノード達でオーバーレイネットワークをTCP接続で構築し，その上でメッセージのルーティングを自動的に行う．つまり，RMIなどで通信する2つのエンドポイントが直接繋がっている必要はなく，複数の中間ノードを挟みオーバーレイ上で繋がっていれば，経路を自動的に見つけ，RMIを実行可能にしている．互いに接続が張れない2ノード間も，それぞれが接続出来る中間ノードを通し，通信を可能にしている．また，ルーティングはモバイルネットワークなどで用いられているAODV[13]を用いることで，計算途中にノードの追加，削除によるオーバーレイトポロジーの変化にも対応することが出来る．

3.5 動的参加

我々のライブラリを使用するPythonプログラムを，既に

```
node A: REGISTER
ENDPOINT: (foo000.bar.ac.jp, 12345)
PATH: '/bar'

node B: REGISTER
ENDPOINT: (foo001.bar.ac.jp, 34567)
PATH: '/bar'

node C: REGISTER
ENDPONT: (alpha030.kiwi.ac.jp, 65432)
PATH: '/kiwi'

node D: LOOKUP  PATH: '/bar'
  ---> [(foo000.bar.ac.jp, 12345),
        (foo001.bar.ac.jp, 34567)]

node D: LOOKUP  PATH: '/'
  ---> [(foo000.bar.ac.jp, 12345),
        (foo001.bar.ac.jp, 34567),
        (alpha030.kiwi.ac.jp, 65432)]
```

図 2 エンドポイントの登録，取得の例
Fig. 2 An example of registering and retrieving endpoints

起動しているノードの集合に加え，途中から計算に参加させることが出来る．まず，プログラムを接続するには，TCP 接続を張る必要がある．これには既に起動しているどれかのノードが待ち受けている IP アドレス，ポートのペアが必要になる．一般に遠隔にある計算資源の IP アドレスを取得することはユーザーには大きな負荷である．また，ポートは一般には固定されていないので，事前に知ることはとても難しい．関連研究で挙げた，JavaParty，Hadoop などは事前にホスト名，ポートなどを明記した設定ファイルの記述を求めるが，ネットワークに詳しくないユーザーには無理な要望である．

高速並列計算のライブラリとしてよく知られている MPI の実装などでは，各ノードに立ち上がるデーモン[1]，またはグリッドシェル[16] を用いて一斉に起動する時に IP，ポートのペアのエンドポイント交換をする．ただし，これらでは動的にノードが参加することは不可能である．

3.5.1 PortMapper

本ライブラリでは設定ファイルの記述を無くす，また動的なノードの参加を可能にするために，エンドポイント情報を管理する HTTP サーバーを立てることを提案している．自分への接続を可能にしたいノードは，この HTTP サーバーに自分のエンドポイント情報を登録する．逆に他のノードに接続したいノードは，この HTTP サーバーを問い合わせ，エンドポイント情報を取得し，接続を張る．この目的に使う HTTP サーバーは既存の apache などにソフトには依存せずに，我々のライブラリ内で提供し，任意のノードで任意のポートに立てることが出来る．

HTTP サーバーは内部では階層的なディレクトリを構成している．こうすることで，エンドポイント情報を階層的な集合に整理することが可能だ．図 2 で説明する．node A, B, C がそれぞれのエンドポイントを`'/bar'`, `'/bar'`, `'/kiwi'` に登録した場合，node D が問い合わせをする．node D は `bar` というドメイン内のノードにしか興味がない場合，問い合わせのパスに`'/bar'` と記述すると，その階層以下に登録されてエンドポイントのみ取得することが出来る．また，登録されているすべてのノードの情報を取得した場合，ルートである`'/'` を指定し，`'/kiwi'` を含めたすべてのエンドポイント情報を取得することが出来る．

一般に，各ノードが接続したい相手は「クラスタ A のノード達のどれか」，などとある特徴を持った集合である．また，一般的には，興味ある集合はネットワークの構成上，階層的に包含関係にあることが多い．そのため，階層的にエンドポイント情報を整理することが望ましい．

3.5.2 HTTP サーバーであることの利点

HTTP を扱えるソフト，ライブラリなどが多く存在することから望ましいプロトコルとして採用した．ライブラリでは Python の HTTP 接続ライブラリをラップして，容易に希望するエンドポイント情報を取得する API を提供している．それ以外にも，ユーザーが希望すれば汎用のブラウザなどで HTTP サーバーをアクセスし，GUI で情報を得ることも容易に出来る．

HTTP サーバーのエンドポイントは勿論，別途の方法で取得する必要があるが，ユーザーには軽い制約だと考えられる．まず，自分の接続したいノード達のすべてのエンドポイントを知る必要がない代わりに，一つの URL に置き換わるという点がある．また，何百とある計算ノードのポートは固定番号には出来ないが，一つの HTTP サーバーは分かりやすい番号に固定出来ることが多い．

4. フレームワークの実装

以上説明したライブラリを用い，揮発的な大規模計算資源上で走るアプリケーションのためのフレームワークを実装した．大量のファイルをワークフローで処理するようなアプリケーションに対して，ユーザーの記述を容易にし，ファイル転送を含めた処理の並列化を行う．また，ノードの増減への対応は勿論，中間ファイルを保持するノードが脱退した場合でもバックアップを取ることで対応する．

4.1 概　　要

マスタープログラムを実行する一つのノードと，ワーカープログラムを実行する多数のノードによって構成される．マスター，ワーカーはそれぞれ遠隔からでもアクセス出来る分散オブジェクトとして表現され，互いに RMI を用いて通信をする．ワーカーはライブラリで提供する portmapper に問い合わせ，マスターがあるノードに接続し，計算に動的に参加することが出来る．マスターは参加しているワーカーに向けてジョブを分配し，ワーカーは与えられたジョブの処理を実行する．ジョブの投入は実行時に，ジョブを実行するワーカーからも RMI で受付，FIFO に処理される．計算から脱退するワーカーはプログラムを終了させることにより完了する．一連のフローを図 3 に示す．

ユーザーは，テンプレートとして与えられたジョブクラ

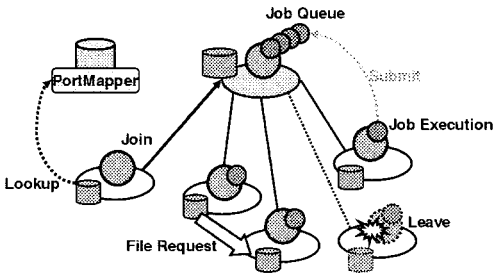

図 3 フレームワークのフロー
Fig. 3 The overall flow of the framework

スを継承したクラスを実装し，処理の入出力ファイルを指定する．ジョブの投入には，ワーカープログラムと同様に，portmapper に問い合わせを行い，マスターに接続し，RMI でジョブオブジェクトの投入を行う．

このモデルでは，パスの設定，コンパイル，設定ファイルの記述はない．また，計算させたいノードでワーカープログラムを起動，終了させることで計算資源を動的に変えることも出来る．ユーザーはジョブクラスを継承した自分のアプリケーションの実装に専念出来る．

4.2 仮　　定

このフレームワークにおいて，マスタープログラムを実行する計算ノードについて，いくつかの仮定をしている．

- マスターノードはワーカーノードのどれからも接続可能である
- マスターノードは故障しない

4.3 ユーザーの記述

既に述べたように，ユーザーはテンプレートとして用意された Job クラスを継承してユーザーが行いたい処理に合わせて変更を加える．具体的には，Job クラスの `run` メソッドをオーバーライドし，適当な処理に置き換えることによって実現される．

4.3.1 入力ファイルの指定

また，ユーザーが定義するジョブは多くの場合，ファイルを処理の入力として取る．このため，処理に必要な入力ファイルを指定する API を Job クラスに用意している．これによって指定されたファイルは処理の実行前に実行ノードへ自動的に転送される．指定には，ジョブを投入する前に以下のコードのようにする．

```
job = CustomJob(args)
#specify input file
job.add_input(prot, path, host)
```

`prot` はファイル取得のために用いる転送プロトコルの指定を求める．これには 2 つのプロトコルが許されている．

'http': wget によって HTTP で指定された `path` からファイルを取得する

'tcp': ファイルを保持するノードのポート `host` へ TCP 接続し，そのノードの `path` のファイルを取得する．

4.3.2 追加ジョブ投入

ある Job が再帰的に子 Job 達を産み出すというような処理も一般的には多い．このような時には，ジョブの処理の一環として子 Job をマスターノードに投入することが必要である．そのために，ジョブ実行時に RMI でマスターにジョブ投入が可能である．具体的には以下のコードのようにする．

```
#pass a list of new jobs
self.master.add_jobs(list_of_jobs)
```

4.3.3 出力ファイルの指定

現在のジョブが出力するファイルが次に実行されるジョブの入力ファイルとなる場合が多々ある．そのような場合，ユーザーに出力するファイルをジョブ実行中に指定することを求めている．具体的には次のように指定する．

```
self.add_output(path)
```

こうすることで，ジョブ終了時に出力ファイルがマスタープログラムに返される仕組みになっている．

4.4 故障への対応

このモデルではワーカーノード達は自由に脱退したり，故障したりすることも考慮している．マスターは定期的にワーカー達に ping を RMI で発している．これに対して十分の時間応答がないワーカーは脱退，もしくは故障したと見なし，それに投入したジョブを別ワーカーに再投入することをしている．

4.5 ファイルの転送

ワーカーはあるジョブを実行する時に，必要なファイルが指定されていれば，自動的にファイルの取得を試みる．既に説明したように，ファイルの転送方法には，TCP，HTTP のプロトコルが用意している．

4.6 ワーカーの脱退，NAT，firewall がある場合の転送

別のジョブの実行に必要なファイルを保持したまま，ノードが故障，脱退する場合がある．具体的には，あるノードで処理されたジョブがファイルを出力し，このファイルを入力とした子ジョブを投入した後，ノードが脱退する場合である．子ジョブが実行される時に，脱退したノードからファイルの取得に失敗する．同様に，一部のノードが private network 内，または firewall の内側にいることが考えられる．すると，外部にいるノードはそれらのノードからのファイルの取得に失敗する．

このようにファイルの取得に失敗した時，図 4 に示すようにマスターノードからファイルの取得を試みる．出力ファイルはすべてマスターに返されるので，マスターが故障しない限り，ファイルの取得に成功する．また，各ノードは自分の出力ファイルをパーシステントな HTTP サーバーにアップロードすることも可能である．この方式だと，HTTP プロト

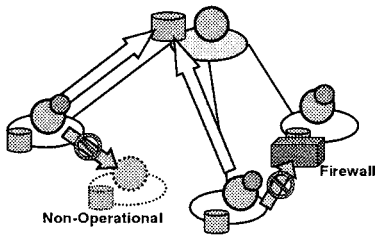

図 4　ワーカーがファイルの転送に失敗した場合，マスターから取得する
Fig. 4　When a worker fails to obtain a file

コルで，マスターを通さずにファイルの取得が可能である．

4.7　ユーザーとのインターフェイス

無事にジョブの記述と実行が開始出来ても，実行時にあらゆるエラーが発生する可能性がある．これは，ユーザーの記述エラーもあれば，システムのエラーもある．仮にジョブの実行中にエラーが発生した場合，ワーカーノードはそのエラーコード，エラーメッセージ，またスタックトレースをマスターに送信し，マスタープログラムのコンソールからエラーの究明が出来る．一つの端末にエラーを集積することで，ユーザーのデバッギングの手間を削減出来る．

5. 評　　価

今回提案した，ファイル転送を考慮した処理並列化フレームワークを実環境を用いて評価した．評価には英文の意味的・構文的解析する HPSG パーサー Enju[5] を用いた．入力には，MEDLINE という医学系論文のアブストラクトのデータベースから得られた大量のアブストラクトを用いた．大量にある MEDLINE の文章の意味的解析は，統合的な医学系意味的サーチエンジンなどの応用が考えられている．[12] しかし解析する文献は膨大にある他，文章の意味的解析には大きな計算量がかかるため，大規模な計算資源で並列的に処理出来ることが望まれている．

今回の評価では，一つのジョブで MEDLINE の論文のアブストラクトを Enju でパースし (Enju Job)，その出力ファイルを入力とした子ジョブで出力ファイルを別のノードに再配置する (Store Job) という実験を行った．

評価では，このフレームワークが大量の資源を用いた時のスケーラビィリティ，また自由に計算資源を追加，削除した場合に正常に計算が進行するかと柔軟性を評価する．

5.1　問題の規模・設定

今回の評価では，MEDLINE のアブストラクトを 10 論文ずつ 1 つのジョブの入力ファイルにまとめた．合計で約 1 万 2000 個のジョブを投入した．一つの入力ファイルの大きさは，

平均61.0 [KB]
最大899.8[KB]
最小2.3[KB]

各ジョブに要する時間は実験の結果，以下のようになった．正確な時間は実行するマシンにより，バラつきが出るものの，

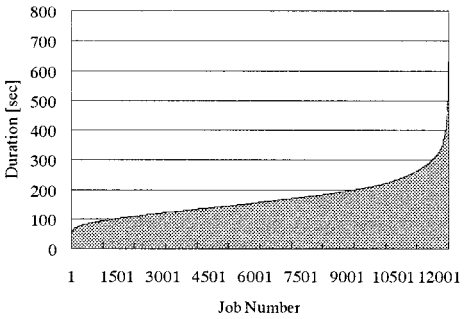

図 5　各 Enju ジョブ実行時間の分布
Fig. 5　Distribution of the computation time of the Enju Jobs

表 2　実験環境

コア数	クラスタ名
90 cores	hongo cluster
363 cores	hongo, chiba, suzuk, okubo clusters
513 cores	hongo, chiba, suzuk, kototoi, imade, okubo clusters

計算の規模を把握するには十分だと考えられる．ジョブ実行時間の分布を図 5 に示す．ジョブ実行時間の簡単な統計は以下の通りである．

平均168[sec]
最大1205[sec]
最小43[sec]

すべての実験で，マスタープログラムを走らせるノードは hongo クラスタにあるノードとした．また，初期のアブストラクトデータはすべて hongo にある 1 つのノードにおかれ，ファイルの取得プロトコルとしては，TCP プロトコルのみを用いた．

5.2　環　　境

本実験では日本各地に計算資源を持つ，InTrigger プラットフォームで実験を行った．特に下に示す，imade クラスタは NAT 環境で動作している．また，kototoi クラスタはグローバル IP を持っているが，firewall により，外部からの接続は受け付けない構成になっている．

5.3　スケーラビリティ

この実験では，計算に用いる計算機規模を変化させながら，ジョブの終了に要する時間を測定した．表 2 に示す構成で実験を行った．計算に使ったコア数に対する実行のスピードアップを図 6 に示す．スピードアップを計算するにあたって，ファイル転送を含まない全てのジョブの実行時間の累積を比較の対象とした．

図 6 より，全ての Enju ジョブが終了するまでの時間はほぼ線形にスケールしている．これは，入力ファイルの大きさが KB オーダーである上，一つのファイルに対する処理は比較的長いからだと言える．Enju の出力ファイルを別のノードにファイル転送するという Store ジョブを含めると，以前

表 1　各クラスタの仕様
Table 1　Specifications of each cluster

サイト名	設置機関	CPU	ノード数 (コア数)	メモリ	ディスク (ローカル)	ディスク (共有)	ネットワーク
chiba	国立情報学研究所	Pentium M 1.86GHz Core2 Duo 2.13GHz	70 (70) 58 (116)	1GB 4GB	300GB 500GB	9TB	グローバル
hongo	東京大学	Pentium M 1.86GHz Core2 Duo 2.13GHz	70 (70) 14 (28)	1GB 4GB	70GB 500GB	2TB	グローバル
kototoi	東京大学	Xeon 2.33GHz	22 (88)	8GB	2TB	9TB	グローバル (firewall)
imade	京都大学	Core2 Duo 2.13GHz	30 (60)	4GB	500GB	9TB	プライベート
okubo	早稲田大学	Core2 Duo 2.13GHz	14 (28)	4GB	500GB	9TB	グローバル
suzuk	東京工業大学	Core2 Duo 2.13GHz	36 (72)	4GB	500GB	9TB	グローバル

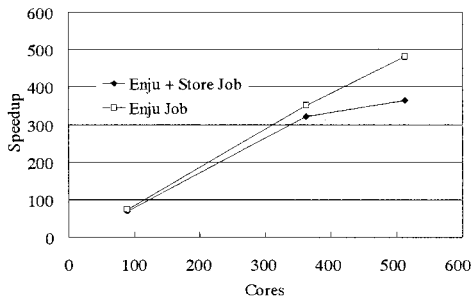

図 6　コア数に対する実行のスピードアップ
Fig. 6　Speedup with respect to the number of cores

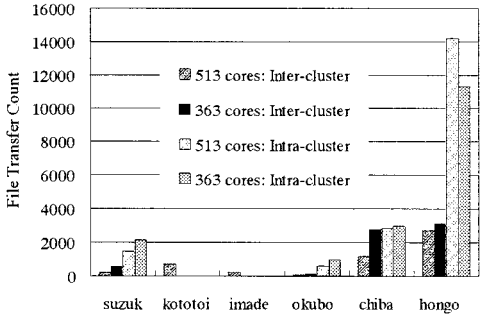

図 7　ファイルの送信先をクラスタ別に分類
Fig. 7　Classification of inter-.intra-cluster file transfers for each cluster

と同様にスケールするものの，スピードアップに大きな影響がある．これは Store ジョブはほぼファイル転送のみをするものであり，転送のオーバーヘッドを秩序に強調するものだからである．

また，実験環境の構成も影響している．513 コアの構成の場合，新たに kototoi，imade という firewall，NAT の環境のノードを含めた．これらのノードで Enju の処理が行われた場合，クラスタ外のノードはファイルの取得に失敗し，hongo クラスタにあるマスターノードにリクエストが来る．この影響で，マスターノードの負荷が一気に上がる．図 7 によると，513 コアの実験の場合，hongo クラスタ外へのファイルの送信が 363 コアの場合よりも大きく上昇している．これは hongo にあるマスターへのファイル要求が多く発生しているからである．

5.4　資源の追加・削除の実験

次に，実験中に動的に計算資源を投入したり，削除しながらジョブが最後まで終了することを確認した．資源の投入の時系列は以下のとおりである．

0 sec: hongo クラスタ投入 (89cores)
1000 sec: chiba, suzuk クラスタ投入 (246cores)
2400 sec: hongo クラスタ削除 (-89cores)
2700 sec: kototoi, imade, okubo クラスタ投入 (176 cores)
4200 sec: chiba, suzuk クラスタ削除 (-246cores)
4800 sec: kototoi, imade, okubo クラスタの大半と通信途絶える (-150cores)
7800 sec: chiba, suzuk クラスタ再投入 (246cores)

投入されたワーカ数を時系列で図 8 に示す．また，ジョブ完了結果を図 9 に示す．無事 13100sec 後に全てのジョブが完了したことが確認取れた．これにより，このフレームワークでは動的に資源の追加・削除が出来ることが確認された．仮にファイルを保持するノードが脱退しても，そのファイルはワーカーから取得され，処理は継続した．しかし，入力ファイルを保持したノードが大量に脱退した場合，マスターからファイルの取得が殺到するため，マスターに非常に大きな負荷が発生する．

5.5　マスターへの負荷の軽減

必要なファイルが取得できない場合，マスターへのファイルリクエストが集中してしまう．ただし本フレームワークでは，HTTP サーバーからファイルを取得する方法もサポートしている．入力ファイルを出力するノードは，そのファイルを外部からアクセスできる HTTP サーバーにアップロードすることで，仮にそのノードが脱退したり，接続を受け付けられない環境であっても，ファイルの提供が出来る．こうすることでファイルリクエストの分散を図ることが出来，効果的にマスターへの負荷を低下出来る．

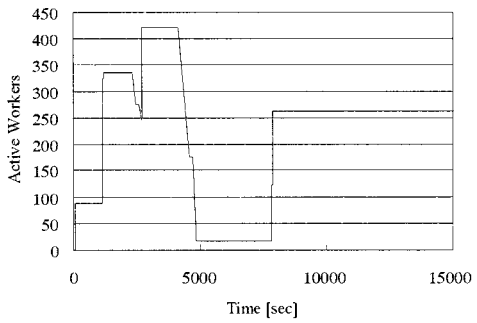

図 8　ワーカー数の変化の時系列
Fig. 8　Change of the number of workers over time

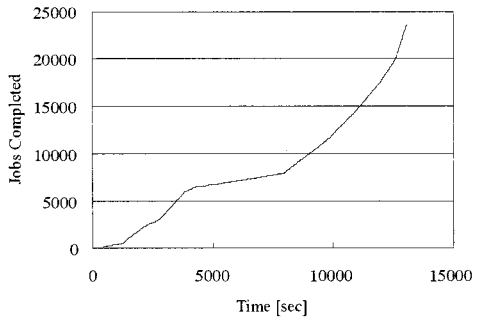

図 9　実行中に資源を追加・削除した場合の処理の進行
Fig. 9　Job progress when nodes are dynamically added and removed

6. まとめ・今後の課題

計算資源の追加・増減を考慮した，大規模な計算環境での並列分散アプリケーションに有効な分散オブジェクト指向ライブラリを提案した．このライブラリ拡張はPythonという高級言語で行い，設定ファイルなどの記述を一切求めないことで，ユーザーに高い生産性を与えることが出来る．ユーザーからネットワーク通信にまつわるIPアドレス，ポートなどの煩雑な部分は極力隠蔽しながら，動的に資源を投入・削除するすべを与えている．

またこのライブラリを用い，自動ファイル転送も行う計算処理を並列化するフレームワークを実装した．このフレームワークでも，ユーザーの記述を少量に留めた上で，ノードが追加・脱退する大規模な計算環境でスケールすることを示した．

今後，ノードが脱退することを受けて，そのノードに存在するオブジェクトをマイグレーションで避難させるプロトコルを構築したい．これはノードが参加・脱退する環境での計算をユーザーにより透過的に見せ，アプリケーションにより大きな柔軟性を与えると考えられる．

7. 謝　辞

本実験において，Enjuパーサーの提供とそれに携わる多大のアドバイスを東京大学情報理工学系研究科，辻井研究室の皆様から頂きました．この場を借りて深く感謝致します．本研究は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新IT基盤研究プラットフォームの構築」の助成を得て行われた．

参 考 文 献

1) R.Butler, W.Gropp, and E.Lusk. A scalable process-management environment for parallel programs. *Lecture Notes in Computer Science*, 1908:168–??, 2000.
2) J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, 2004.
3) dRuby. http://www.druby.org/ilikeruby/druby.html/.
4) M.T. Egner, M.Lorch, and E.Biddle. Uima grid: Distributed large-scale text analysis. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.
5) ENJU. http://www-tsujii.is.s.u-tokyo.ac.jp/enju/.
6) D. J. Goodman. Introduction and evaluation of martlet: a scientific workflow language for abstracted parallelisation. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 983–992, New York, NY, USA, 2007. ACM Press.
7) Hadoop. http://lucene.apache.org/hadoop/.
8) F. Huet, D. Caromel, and H. E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA, Nov. 2004.
9) Intrigger. https://www.logos.ic.i.u-tokyo.ac.jp/intrigger/.
10) JavaParty. http://svn.ipd.uni-karlsruhe.de/trac/javaparty/.
11) KaRMI. http://www.ipd.uka.de/JavaParty/KaRMI/.
12) MEDIE. http://www-tsujii.is.s.u-tokyo.ac.jp/medie/.
13) C. Perkins. Ad-hoc On-demand Distance Vector Routing, 1997.
14) M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997.
15) Pyro. http://pyro.sourceforge.net/.
16) K. Taura. Gxp: An interactive shell for the grid environment. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 59–67, Washington, DC, USA, 2004. IEEE Computer Society.
17) TSUBAME. http://www.gsic.titech.ac.jp/ccwww/t-gc/.