Vol.2017-HPC-158 No.34
2017/3/10

並列ベンチマークのための同期複数タスク実行フレームワークの設計

岩井 厚樹[1,a)]　建部 修見[2]　田中 昌宏[2]

概要：本論文では並列ワークフローエンジンを活用した同期複数タスク実行フレームワークの設計について述べる．データインテンシブサイエンスを実現する大規模計算環境において，これまで並列ベンチマークを行う際は MPI を用いた単一タスクのマイクロベンチマークや，サイエンスワークフローアプリケーションを用いてベンチマークをとるのが一般的であった．しかし，大規模計算環境では多数のタスクが同時に実行され，同時に多数タスクを実行するベンチマークが求められている．スーパーコンピュータにおいても実際は多数のタスクが同時に実行されており，それらの状況におけるベンチマークが求められている．だが，大規模計算環境において，多数のノードで複数のベンチマークタスクを同時に実行するのは容易ではない．また，同期処理を行う場合はさらに工夫が必要となる．本研究では同時に多数のタスクを実行するワークロード向けの並列ベンチマークを可能にする同期複数タスク実行フレームワークの設計を行う．これにより，多数のタスクが同時に複数ノードで実行される環境における，計算機およびタスクにとって最適な並列ベンチマークの実現を目指す．

1. はじめに

素粒子物理学や天文学，生命科学などの分野におけるデータ量はセンサーや加速器などの実験装置の向上，スーパーコンピュータによるシミュレーションなどにより，そこで扱うデータ量は増加の一途を辿っている．例えば，気象の分野においては，30 秒間に 200GB のデータが入出力される [1]．また，X 線自由電子レーザーに関する科学計算では実験データが最大で毎秒 5.8GB 生成される [2]．それらの大規模データを解析することで結果を得る手法をデータインテンシブサイエンスと呼ぶ．

解析やシミュレーションを実行する前に，スーパーコンピュータの性能測定やアプリケーションの実行見積もりをするために，IOR[3] や mdtest[4] のようなマイクロベンチマーク，Montage[5] や CyberShake[6], [7] といったサイエンスワークフローアプリケーションを用いて，複数ノードのベンチマークをとることが一般的である．これまでは先のようなベンチマークが主流であったが，データインテンシブサイエンスにおいては，複数のノードで多数のタスクが同時に実行されている．そのため同時に多数タスクを実行するベンチマークが求められている．また，スーパコンピュータにおいても実際は多数のタスクが同時に実行されており，それらの状況におけるベンチマークが求められている．以上の要求を満たすには，多数のタスクを複数ノードで並列実行できるベンチマークが必要である．本研究では，多数のタスクを同時に実行して，並列ベンチマークを行うための同期複数タスク実行フレームワークを設計する．

同期複数タスク実行フレームワークを実現するにあたり，並列ワークフローエンジン Pwrake[8] を用いた．Pwrake は Rake[9] というビルドツールを拡張した並列分散実行可能なワークフローエンジンである．ワークロードを記述する Rakefile は Ruby 構文で記述できるため，Rake と同様のビルドツールである Unix make の Makefile と比べて柔軟性高く記述することができ，タスクを定義しやすいという利点がある．また，Pwrake は Gfarm ファイルシステム [10] と協調することでデータの局所性を考慮したスケジューリング [11] が可能であり，データインテンシブサイエンスにおいて多く利用されるワークフローの実行，データ解析に適している．しかし，Pwrake はベンチマークツールとして使用することを想定して開発されてはいない．並列ベンチマークにおいて重要なことの一つに，各ベンチマークタスクの開始時刻を同期して実行することがある．そこで本稿では，提案するフレームワークにおけるベンチマークタスク実行の同期性について評価を行う．

以下 2 章では，関連するベンチマークおよび並行実行可能な SSH について述べる．3 章では，フレームワークの設

[1] 筑波大学情報学群情報科学類
[2] 筑波大学計算科学研究センター
[a)] iwai@hpcs.cs.tsukuba.ac.jp

計および実装に際して必要となる Pwrake について解説する．4 章では，同期複数タスク実行フレームワークの設計および実装について述べる．5 章では，実装した同期複数タスク実行フレームワークの評価を行った結果を示す．6 章では，まとめと今後の課題について述べる．

2. 関連研究

2.1 既存の並列実行ベンチマークプログラムの調査

これまでスーパーコンピュータのような多数のノードを持つ並列ファイルシステムのベンチマークは，IOR や mdtest のような MPI を使ったベンチマークが一般的であった．

IOR は分散ファイルシステムのバンド幅を測定するベンチマークである．MPI を用いることで，複数のノードからの並列アクセスバンド幅を測定できる．オプションを指定することで，POSIX，MPIIO，HDF5，NCMPI の各種インタフェースに対応可能である．

mdtest はメタデータ操作を測定するベンチマークプログラムである．複数ノードのディレクトリおよびファイルの create，stat，remove 操作のスループットを MPI で収集し，出力する．

しかし，いずれのベンチマークも性能について測定できる項目は限られており，新たな測定項目を導入するには，独自に拡張するか，新たに MPI で測定プログラムを記述する必要があった．本論文で設計するフレームワークを用いれば，ユーザはシングルスレッドで動作するマイクロベンチマークプログラムをフレームワークに登録するだけで，アプリケーションに適した独自のマイクロベンチマークを多数のノードで並列実行でき，性能を容易に測定することができる．

2.2 並行実行可能な SSH の調査

複数ノードでのプログラムの実行は dsh[12] や pssh[13]，ClusterSSH[14]，GXP[15] のような複数ノードに一括にログインし，コマンドを制御可能な並行 SSH を用いることで可能である．

dsh および pssh はコマンドラインもしくは指定のファイルにログインノードと実行コマンドを記述することで並行 SSH を実現する．ClusterSSH は複数の xterm が起動し，それらを単一のコンソールから制御することができる．そのため，別々の xterm を利用して複数ノードで別々のプログラムを実行することが可能である．GXP は多数のノードで瞬時にプログラムを並行実行することを目標に開発された分散クラスタ環境向けのシェルであり，600 ms で 300 ノードにコマンドを送信できるという特徴を持つ．

以上の並行実行可能な SSH を用いて，スーパーコンピュータのような数千ノードある環境で複数回同期処理を実行しようとした場合，幾つか問題がある．dsh および pssh はログイン先での処理が完了するたびに接続を切るため，その都度数千ノードに接続しなければならない．ClusterSSH の場合，一度起動すれば ClusterSSH を終了するまで各ノードと接続を保ったままだが，全ての接続先のシェルを xterm で表示するため，あまりにもノード数が多いと接続元の計算機に負担がかかる．GXP は，拡張機能の GXP make[16] により，複数回同期処理が可能であるが，GXP make にはタスクを決められたノードで実行する機能はない．

Pwrake で複数回同期処理を実行しようとした場合，一度処理が始まると終了するまでワーカーとキープアライブな状態となるため，逐一ノードに接続しなくて済む．また，複数回同期処理の実現には Rake の事前タスクを応用すれば容易に実現できる．そして，Pwrake には Gfarm ファイルシステムの局所性を考慮したタスク配置が可能であり，そうした環境下での性能測定が可能である．

```
task :task_name => [:prerequisite_task_name] do
  actions
end
```

図 1 Rakefile の記述方法

3. 予備知識

3.1 Rake

Rake[9] とは Unix make のようなビルドツールである．ワークロードを記述する Rakefile は Ruby 言語で記述可能であり，なおかつ Ruby のライブラリも使用可能なため，Unix make の Makefile よりも柔軟性高く記述できるという利点がある．

Rakefile の記述方法を**図 1** に示す．*task_name* は，タスクの名前を表す．*prerequisite_task_name* は事前タスクを表す．事前タスクとは，あるタスクを実行する前に実行するタスクのことであり，複数個指定できる．*actions* にはタスクの実行内容を記述する．

3.2 Pwrake

Pwrake[8] とは，Rake をベースに並列分散実行の機能を拡張したワークフローエンジンである．そのため，Rake との互換性があり，Rakefile も同様の規則で記述できる．Pwrake はデータインテンシブなワークフローを目的として，Gfarm ファイルシステムを利用し，局所性を高めるスケジューリングにより，高い I/O 性能を発揮する処理を可能にする．

図 2 に Pwrake の動作概要を示す．Pwrake は起動すると，実行ノードとその使用コア数を記したホストファイルを読み，総コア数分のワーカースレッドを作成する．次に Rakefile を読み，Task クラスのインスタンスを作成し，Task Queue クラスとして実装したタスクキューに挿入す

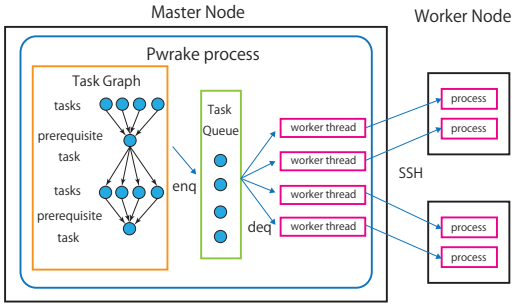

図 2　Pwrake の動作概要図 ([11] より作成)

る．それぞれのワーカースレッドでは起動すると SSH で担当のワーカーノードに接続し，タスクキューから取り出されたタスクを接続先で実行する．

4. 設計と実装

4.1 フレームワークの設計

多種多様なアプリケーションに適した柔軟性の高いベンチマークワークロードを提供するために，ワークフローエンジンを用いた同期複数タスク実行フレームワークを設計する．フレームワークが提供する機能は以下の通りである．

- タスクをワーカーに分配し，実行する
- 複数のワーカーに割り振られたタスクの同期処理
- ノード内での複数タスクの識別

フレームワークはワークロードをタスクに分割し，それらをワーカーに分配する．ワーカーは分配されたタスクを実行したり，他のワーカーと同期をとったりすることで多種多様なワークロードに対応できる．同一ノードに複数のワーカーが割り当てられた場合，ベンチマークプログラムの出力ファイルが重複するような問題が発生する．これを回避するために，実行するプログラムにワーカーごとの識別子を与える．これらを実現するために，Pwrake を用いる．

Pwrake は複数台のノードを用いた並列分散実行が可能なため，ワーカーにタスクを割り振ることができる．また，Pwrake は Rake をベースにしており，Rake の事前タスクを応用することで同期処理の実現が可能である．そして，Rakefile は Ruby が使用できることから，複雑な処理の記述が可能である．さらに，Pwrake は天文学やバイオインフォマティクス [17] の分野で実際のアプリケーション実行に用いられており，実用性が高いなどの理由からフレームワークの実装に Pwrake を使用した．

4.2 フレームワークでのタスク間の同期手法

タスクの分配は並行実行可能な SSH で実現できるが，タスク間での同期処理は新たに設計しなければならない．**図 3** に設計したタスク間の同期処理の概要を示す．

図 3 を用いて同期処理の手法の説明をする．図中の全同期タスクは，同期を行うための空タスクとなっている．

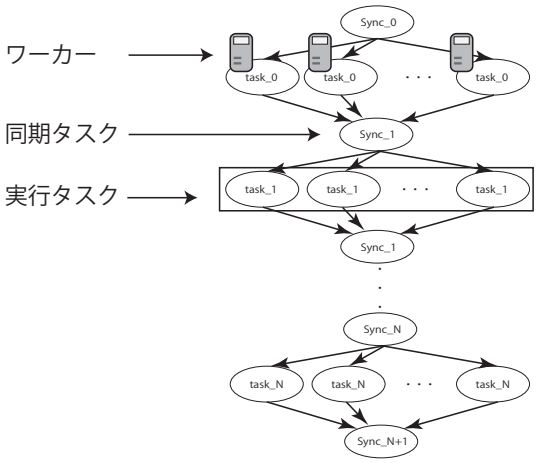

図 3　フレームワークでのタスク間の同期手法の図

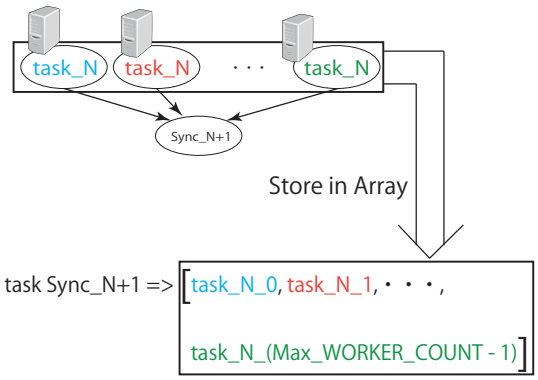

図 4　同期タスクの事前タスクの生成

同期タスク *Sync_0* 実行後，ワークフローエンジンによって全ワーカーに一斉に実行タスク *task_0* が割り振られる．各ワーカーで実行タスク *task_0* が完了すると，同期タスク *Sync_1* が実行される．同期タスク *Sync_1* は実行タスク *task_0* が全ワーカーで実行終了しない限り開始されないので，同期をとることができる．そして，同期タスク *Sync_1* 終了後に実行タスク *task_1* がワーカーに割り振られる．このように同期タスクを活用することで同期処理ができるようにしている．

4.3 実装手法

Pwrake は事前タスクが複数個指定されていた場合，それらをワーカーに割り振るという特徴がある．そのため，同期タスクの事前タスクを複数の実行タスクが格納された配列にすることで，各ワーカーに実行タスクを割り振れるようにした．**図 4** に，その概要図を示す．実行タスク *task_N* は同期タスク *Sync_N+1* の事前タスクとなる必要がある．これを実現するためにワーカーに割り振られるタスクを，識別子を付与して配列に格納するようにした．

同期タスクの事前タスクは実行タスクとし，実行タスクの事前タスクは同期タスクとすることで，図 3 のような依存関係を実現した．

実行タスクの実行には Rake の *sh* メソッドを用いた．

```
WORKER_COUNT = 8
TASK_FILE = "./command"
PARAMETER_ARRAY = ["4k", "8k", "16k"]
ARGUMENTS = "-i 10 -d /tmp/work"
```

図 5　フレームワークの設定ファイル例

表 1　評価環境

CPU	Intel(R) Xeon(R) E5620 CPU 2.40 GHz x2
コア数	8 (4cores/socket)
メモリ	24GB
OS	CentOS 6.8
ストレージ	Fusion-io ioDrive 160GB
ネットワーク	IPoIB
Ruby	ver. 2.3.1
Pwrake	ver. 2.1.2

4.4　環境変数の定義

フレームワークの実行の際には，以下の環境変数を通じてパラメータを指定する．

- *WORKER_COUNT*
- *TASK_FILE*
- *PARAMETER_ARRAY*
- *ARGUMENTS*

WORKER_COUNT は．Pwrake がタスクを割り振る対象のワーカー数を表す．

TASK_FILE にはワーカーで実行するプログラムの相対パスもしくは絶対パスを与える．Pwrake はワーカーに Rakefile のみを送信するため，実行プログラムを各ノードに配置するか，NFS もしくは Gfarm で共有しておく必要がある．

PARAMETER_ARRAY にはワークロード内での変数の一覧を与える．例えば，ブロックサイズを変えながらバンド幅を測定する時は *["4k", "8k", "16k",...]* のように与えることができる．

ARGUMENTS には実行プログラムの引数を与える．

図 5 に以上の環境変数を記述した設定ファイルの例を示す．設定ファイルに記述した環境変数は Rakefile 内で呼び出され，Pwrake によって処理される．

5.　評価

5.1　評価環境

表 1 に評価環境を記す．評価に使用したノードはすべて同一の NTP サーバを参照するようにした．ノードは 8 台使用し，Gfarm を用いてファイルシステムを構築した．メタデータノードは 1 台，計算ノードは 7 台とした．

5.2　ノード間のシステム時間の評価

フレームワークの評価の準備として，ノード間のシステム時間差が評価に影響するかどうか調べるため，ノード間のシステム時間差および往復遅延時間 (RTT) の評価を行った．システム時間差は 2 つのノード間でのシステム時間の差を表し，RTT はシステム時間差の限界値とする．システム時間差が RTT よりも長い場合，測定対象ノードは測定基準ノードと比べてシステム時刻が大幅にずれていると言える．

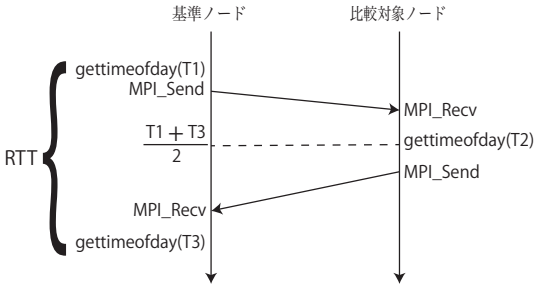

図 6　システム時間差の評価手法

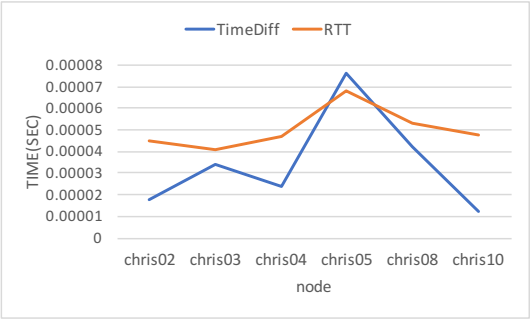

図 7　ノード間時間差評価および RTT の評価結果

評価方法は図 6 に示す．システム時間差は 1 つのノードを基準とし，対象とするノードとの時間差を Ping-Pong 方式で評価した．図において，*T1* は基準ノードで *MPI_Send* をする直前の時間，*T2* は比較対象ノードで *MPI_Send* をする直前の時間，*T3* は基準ノードで *MPI_Recv* をした直後の時間である．*T1*，*T2*，*T3* を用いた計算式 $\left(\frac{T3+T1}{2} - T2\right)$ をノード間の時間差とした．RTT は *MPI_Send* を実行し，*MPI_Recv* が完了するまでの時間 $(T3 - T1)$ とした．

評価結果を図 7 に示す．青い線は基準ノードと比較対象ノードとのシステム時間差を表し，オレンジの線は RTT を表す．図より，基準ノードとのシステム時刻差はおよそ 0.012~0.076 ミリ秒となっていることがわかる．このことから後述するフレームワークの評価結果には最大で 0.064 ミリ秒の誤差があると言える．

5.3　同時実行開始ラグ

実行タスクがワーカーに割り振られ実行開始した時の，最初に実行開始したタスクの開始時間と最後に実行開始したタスクの開始時間の差 (図 8 中の Time Lag) を評価した．一度のワークロードでワーカーに 10 回実行タスクを割り振り，それを 10 回実行し，評価結果を得た．最大使用ワーカー数は 56 とした．タスクの実行プログラムは予め使用するノードに配置した．

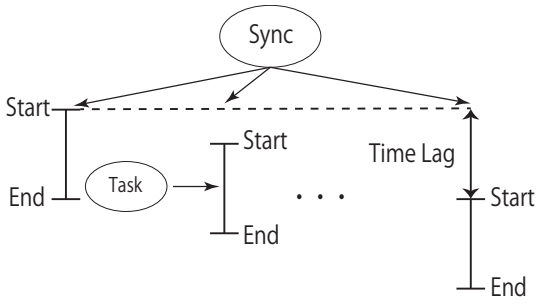

図 8　同時実行評価の概要図

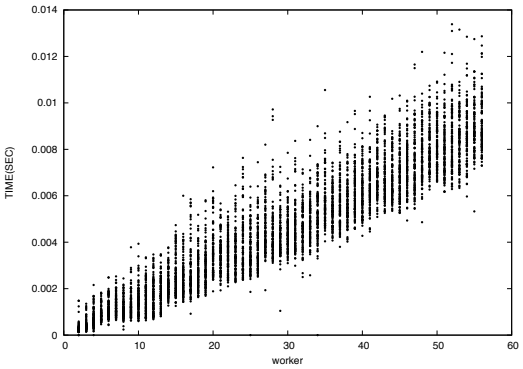

図 9　ワーカーごとの同時実行開始ラグの全データ

5.3.1　ワーカー数を変化させた時の同時実行開始ラグの変化

図 9 は，評価結果の全データをプロットした図である．図 10 は，全データから平均値および最小値を算出したグラフである．

ワーカー内でのタスクの実行は OS のプロセス管理の影響を受けるため多少の変動はあるものの，ワーカー数が増えると同時実行開始ラグが大きくなることがわかる．

5.2 より，フレームワークの評価誤差は 0.064 ミリ秒である．評価誤差と図 10 のワーカー数 56 における平均値 9 ミリ秒を比べると，評価誤差が平均値に比べ非常に小さく，評価誤差は同時実行開始ラグの評価に影響しないと言える．

本評価環境ではワーカー数 56 でおよそ同時実行開始ラグが 9 ミリ秒という結果が得られた．図 10 を見ると，平均値はリニアに増加している．これより，ワーカーが 1 つ増えるごとに同時実行開始ラグはおよそ 0.16 ミリ秒増えると言える．

5.3.2　ワークロード内での同時実行開始ラグの比較

図 11 は一度のワークロードでワーカーへのタスクの割り振りを 10 回実行した時の，それぞれの同時実行開始時間ラグの平均である．Pwrake は一度のワークロードを実行している間は各ワーカーとネットワークを繋いだままなので，最初の実行だけワーカーへの接続のオーバーヘッドがあるかと思われたが特になかった．いずれの実行でも同時実行開始ラグに差はなくフレームワークの安定性を確認できた．

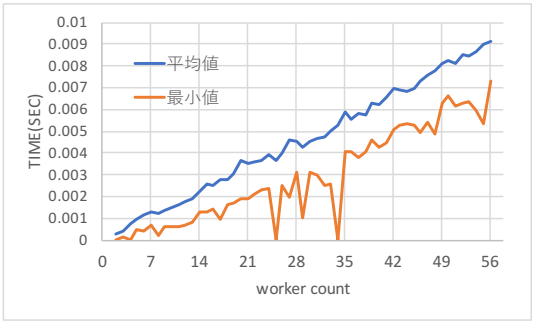

図 10　ワーカーごとの同時実行開始ラグの平均と最小値

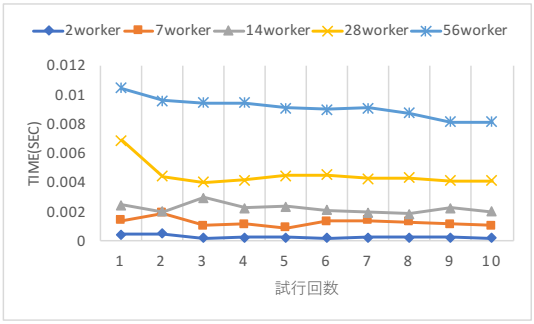

図 11　ワークロード内の同時実行開始ラグの比較

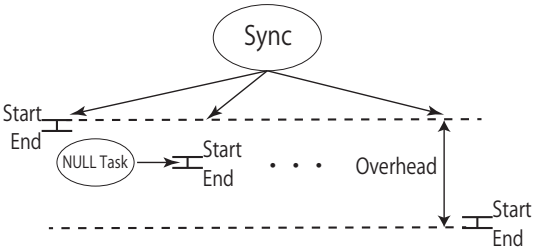

図 12　同期オーバーヘッドの評価概要図

5.4　同期オーバーヘッド

実行時間を 0 秒と仮定した何もしない NULL Task を用いて，同期オーバーヘッドを測定した．図 12 に評価の概要を示す．最初に始まったタスクの開始時間と最後に終わったタスクの終了時間の差分を同期オーバーヘッドとした．最大使用ワーカー数は 56 とした．タスクの実行プログラムは予め使用するノードに配置した．

5.4.1　ワーカー数を増やした時の同期オーバーヘッドの変化

図 13 は，評価結果の全データをプロットした図である．図 14 は全データから平均値および最小値を算出したグラフである．

同期オーバーヘッドも OS のプロセス管理による多少の影響はあるものの，ワーカー数が増えるにつれ同期オーバーヘッドが大きくなることがわかる．

今回の評価では，ワーカー数 56 での平均同期オーバーヘッドはおよそ 17 ミリ秒であった．この結果より，ワー

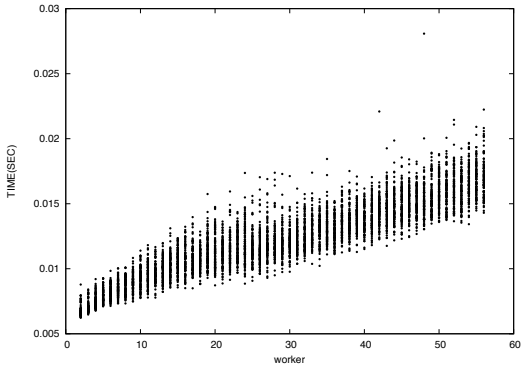

図 13　ワーカーごとの同期オーバーヘットの全データ

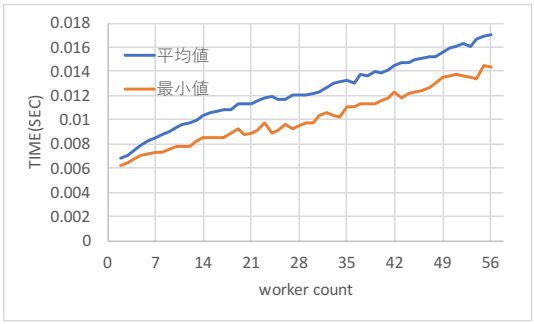

図 14　ワーカーごとの同期オーバーヘッドの平均と最小値

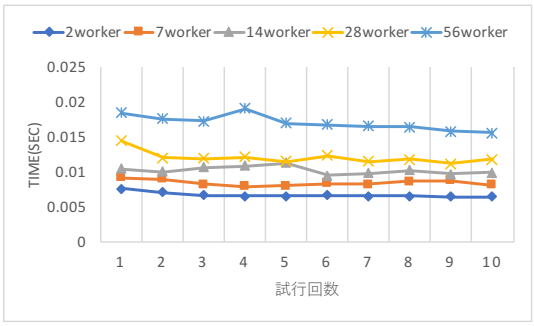

図 15　ワークロード内の同期オーバーヘッドの比較

カー数が 56 以下において，実行時間が 1.7 秒以上のプログラムであれば，プログラムの実行時間中 99%は同一の時間帯に複数ワーカーで同時にタスクを実行できているということになる．

5.4.2　同一ワークロード内での同期オーバーヘッドの評価

図 15 は一度のワークロードでワーカーへのタスクの割り振りを 10 回実行した時の，それぞれの同期オーバーヘッドの平均である．

いずれのワーカー数での実行でも同期オーバヘッドに大きな差がないことがわかる．ゆえに，一度のワークロードで複数回処理を実行し，同一の時間帯に複数のノードが一斉に処理をする必要があるベンチマークを設計する場合，いずれの実行においても同期オーバーヘッドの変動を考慮する必要なくベンチマークを取ることができる．

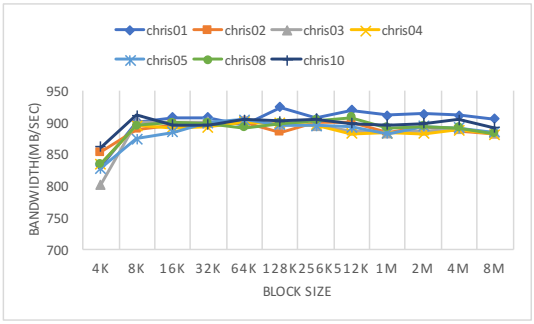

図 16　書き込みバンド幅

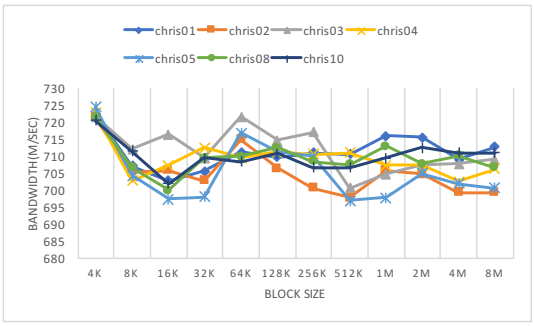

図 17　読み込みバンド幅

5.5　応用事例：バンド幅測定

設計したフレームワークを用いて，複数ノードのバンド幅を測定した．バンド幅の測定にはファイルへの読み書きを通してバンド幅を測定できる fio ベンチマーク [18] を使用した．fio はファイルへのアクセスパターンや I/O サイズを指定できる．評価ではファイルへのアクセスパターンをシーケンシャルリード，シーケンシャルライトとし，評価環境のメモリと同程度のサイズである 24GB をブロックサイズを変化させながら読み書きすることで評価した．

図 16 および図 17 に評価結果を示す．縦軸はバンド幅，横軸はブロックサイズを表す．単一ノードで動作するマイクロベンチマークをフレームワークの設定ファイルに登録するだけで，各ノードのバンド幅を測定できた．

5.6　応用事例：メタデータノードの性能測定

フレームワークを用いてメタデータノードの性能測定を行った．各ノードでディレクトリおよびファイルの create，stat，remove を 10000 回実行し，その経過時間でメタデータ操作回数を除算しスループットを求めるプログラムを C で実装した．試行回数は 10 回とし，その平均を求めた．メタデータ操作の実行ディレクトリを Gfarm のマウントポイントとし，ioDrive における Gfarm のメタデータ操作性能を測定した．

図 18 に評価結果を示す．いずれのノードにおいても，最も性能値が良い stat 操作がおよそ 700[IOPS] となっていることがわかる．1 秒間に 700 回の操作が実行できるため，10000 回の操作でおよそ 14 秒かかる．分散ファイルシ

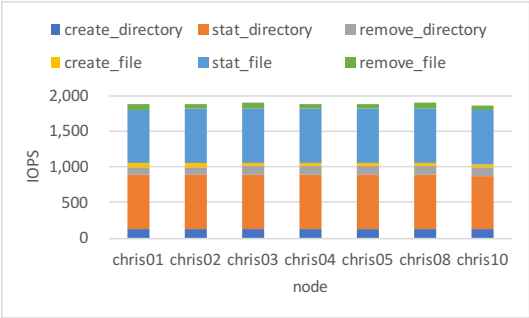

図 18　メタデータ性能

ステムのメタデータノードのメタデータ操作を評価する場合，複数ノードが同一の時間帯にメタデータ操作を行う必要がある．評価環境において 5.4.1 より，1.7 秒以上の処理であれば 99%同一の時間帯に処理を実行できるので，この結果は分散ファイルシステムのメタデータノードの性能測定に妥当な結果と言える．

6. おわりに

本論文では並列ベンチマークを可能にする同期実行可能なフレームワークを設計した．

データインテンシブサイエンスにおいては，多数のタスクが同時に実行されるため，同時に多数タスクを実行する並列ベンチマークが求められている．また，複数ノード上で独自のベンチマークプログラムを実行させるには，ワーカーへのタスクの分配や，複数タスク間の同期処理などの課題があった．これらの要求・課題を満たすために，複数のタスクをワーカーに分配し，同期処理が可能な同期複数タスク実行フレームワークの設計を行った．

これまで，複数のノードにログインし並行実行を可能とする SSH を用いることで，複数ノードでのタスクの実行は可能であった．しかし，それらは同期処理を考慮しておらず，新たに設計する必要があった．今回設計した同期手法は，空のタスクである同期タスクと各ワーカーに割り振られる実行プログラムが含まれる実行タスクに対して，互いに依存関係を持たせることで実現した．設計したフレームワークを使うことでユーザはシングルスレッドで動作する実行プログラムをフレームワークの設定ファイルに登録するだけで，容易に並列ベンチマークの開発および実行ができる．

設計・実装したフレームワークを評価したところ，同時実行開始ラグおよび同期オーバヘッドともにワーカー数を増やすにつれ大きくなることがわかった．同期オーバヘッドに関しては 56 ワーカーでおよそ 17 ミリ秒という結果が得られた．よって，実行時間が 1.7 秒以上のプログラムであれば，プログラムの実行時間中の 99%は複数ワーカーで同時に実行できているということが言える．

今後の課題として，数千ノードあるようなさらに大規模な環境で評価をする必要がある．数千ノードある環境において複数ワーカーで同時にタスクを実行開始しようとした場合，最後に始まるタスクが開始するまでに，最初に始まったタスクが終了してしまう可能性がある．この問題を回避する方法として，フレームワーク側で *sleep* コマンドのようなプログラムを用いて，割り振られるタスクの順番に応じて *sleep* 処理を付随させ，複数ワーカーで同時に実行開始できるようタスクの実行開始を調節する方法が考えられる．また，今回の設計では単一のプログラムしかフレームワークで実行させることはできないが，複数のプログラムを実行できるようフレームワークを拡張させる必要がある．

謝辞　本研究の一部は JST-CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」による．

参考文献

[1] Matsuoka, S., Sato, H., Tatebe, O., Koibuchi, M., Fujiwara, I., Suzuki, S., Kakuta, M., Ishida, T., Akiyama, Y., Suzumura, T. et al.: Extreme Big Data (EBD): Next generation big data infrastructure technologies towards yottabyte/year, *Supercomputing frontiers and innovations*, Vol. 1, No. 2, pp. 89–107 (2014).
[2] Sugimoto, T., Joti, Y., Ohata, T., Tanaka, R., Yamaga, M. and Hatsui, T.: Large-bandwidth data acquisition network for XFEL facility, SACLA, *Proceedings of ICALEPCS2011, Grenoble, France*, Vol. 626 (2011).
[3] IOR, https://github.com/LLNL/ior.
[4] mdtest, https://sourceforge.net/projects/mdtest/.
[5] Deelman, E., Singh, G., Livny, M., Berriman, B. and Good, J.: The cost of doing science on the cloud: the montage example, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, p. 50 (2008).
[6] Graves, R., Jordan, T. H., Callaghan, S., Deelman, E., Field, E., Juve, G., Kesselman, C., Maechling, P., Mehta, G., Milner, K. et al.: CyberShake: A physics-based seismic hazard model for southern California, *Pure and Applied Geophysics*, Vol. 168, No. 3-4, pp. 367–381 (2011).
[7] Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., Mayani, R., Chen, W., da Silva, R. F., Livny, M. et al.: Pegasus, a workflow management system for science automation, *Future Generation Computer Systems*, Vol. 46, pp. 17–35 (2015).
[8] Tanaka, M. and Tatebe, O.: Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ACM, pp. 356–359 (2010).
[9] Rake, http://rake.rubyforge.org/.
[10] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm grid file system, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (2010).
[11] 田中昌宏，建部修見ほか：ワークフローシステム Pwrake における I/O 性能を考慮したタスクスケジューリング，

研究報告ハイパフォーマンスコンピューティング (HPC), Vol. 2014, No. 3, pp. 1–10 (2014).
[12] dsh, https://sourceforge.net/projects/dsh/.
[13] pssh, https://linux.die.net/man/1/pssh.
[14] ClusterSSH, https://sourceforge.net/projects/clusterssh/.
[15] Taura, K.: GXP: An interactive shell for the grid environment, *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2004. Proceedings*, IEEE, pp. 59–67 (2004).
[16] Taura, K., Matsuzaki, T., Miwa, M., Kamoshida, Y., Yokoyama, D., Dun, N., Shibata, T., Jun, C. S., Jun' ichiTsujii : Design and implementation of GXP make— A workflow system based on make, *Future Generation Computer Systems*, Vol. 29, No. 2, pp. 662–672 (2013).
[17] Mishima, H., Sasaki, K., Tanaka, M., Tatebe, O. and Yoshiura, K.-i.: Agile parallel bioinformatics workflow management using Pwrake, *BMC Research Notes*, Vol. 4, No. 1, p. 331 (2011).
[18] Flexible I/O Tester, https://github.com/axboe/fio.