タスク並列言語におけるノード間通信の実装方式の検討

平石 拓[1,a)]　村岡 大輔[2,4]　八杉 昌宏[3]

概要：我々は，ワークスティールによる自動負荷均衡化を実現するタスク並列言語 Tascell を開発している．Tascell は分散メモリ環境にも対応しているが，本発表ではこれを実現するノード間通信のいくつかの実装方式について紹介し，性能や実装コストなどの観点から議論する．最初に採用した TCP/IP による実装では，ノードの動的な追加や広域分散環境に対応できるという利点がある一方，通信性能が低い，一部のスパコン等の環境は TCP/IP によるノード間通信に対応していないという問題がある．そこで MPI ライブラリによる実装も行い，京コンピュータ等の高並列環境で良好な性能を得られることを確認している．なお，片方向通信や複数スレッドでの通信関数の呼出しに対応しない MPI 環境でも動作するような実装とするため，デッドロック回避のための工夫が必要となった．本発表では，本 MPI 実装の東京大学のスーパーコンピュータ Reedbush-U での性能を報告するとともに，片方向の MPI 通信や，大規模環境での効率良い通信を柔軟に記述するために近年開発されている通信ライブラリ ACP を用いた実装を含めた，今後の改良方針について議論する．

キーワード：タスク並列言語，ノード間通信，動的負荷分散

1. はじめに

今日の高性能計算は，複数のコアを持つ計算ノードを多数接続した大規模並列計算システムにより実現されることがほとんどである．探索木やグラフの探索に代表される不規則アプリケーションのこのような並列計算環境向けの実装では，各計算コアへの負荷が均一になるようなタスク割当を事前に行うことは困難なため，動的負荷分散が必須となる．このような処理を簡単に実装できるようにするため，並列実行可能な単位をタスクとして記述しておくと，それらタスクの計算コアへの割り当てによる負荷均衡化が処理系により自動的に行われる「タスク並列言語」がいくつか提案されている．

我々が提案している Tascell [7] もそのような言語のひとつである．Tascell ワーカは他ワーカからのタスク要求がなければ，タスク生成可能箇所の記憶は行うのみでタスクの生成を一切行わず逐次実行を行う．他のアイドルなワーカからのタスク要求があると，要求を受けたワーカは，できるだけ大きなタスクを生成するため，一時的バックトラックにより最古のタスク可能状態を復元してからタスク生成を行う．その後，バックトラック前の状態を再度復元し自身が実行中だったタスクを再開する．この機構により論理スレッドの生成／

[1] 京都大学学術情報メディアセンター
[2] 九州工業大学大学院情報工学府
[3] 九州工業大学大学院情報工学研究院
[4] 現在，株式会社ジーニー
[a)] tasuku@media.kyoto-u.ac.jp

維持コストを完全に無くすことができるのに加え，ワーカの作業領域の再利用性および局所性を向上させ，高い性能を実現できる．

他のタスク並列言語としては Cilk [6]（およびその発展版の Cilk Plus [8]）や X10 [5] などが有名だが，これらの言語に対して，Tascell は上記の一時的バックトラック機構による性能面での優位性に加え，計算ノードを跨いだワーカ間でのワークスティールによる自動負荷均衡化にも対応しているという優位性をもつ．

しかし，このノード間負荷均衡を実現している Tascell の通信層の実装についてはいくつか改善の余地がある．

初期の Tascell の実装 [7] では，ノード間通信は Tascell サーバと呼ぶ中継サーバを介する TCP/IP 通信により実現している．この実装は計算ノードの追加や広域分散環境での並列計算にも容易に対応できる [12] という利点がある一方，Tascell サーバが通信のボトルネックになるという自明な欠点をもつ．さらに，近年のスーパーコンピュータでは計算ノード間の通信手段として TCP/IP を使えないことも多く，そのような環境では Tascell を動かすことができないという問題もある．

そこで我々は以前，スーパーコンピュータ環境でのノード間通信手段の主流となっている MPI を用いた Tascell の通信層の実装を行った．またこれと同時に，性能ボトルネックとなる中継サーバを用いない実装とした．なお，ノード間通信を Tascell プログラマ自身が直接記述することは（でき）ないため，このような実装の違いは Tascell プログラムを書く側は気にする必要がないことに注意されたい．

MPI による実装を行う際に気を付ける必要があるのが，MPI 環境のスレッドサポートレベルの問題である．たとえば，スーパーコンピュータ「京」における MPI ライブラリは，スレッドサポートレベルは `MPI_THREAD_SERIALIZED`，すなわち，同時に 1 つのスレッドしか MPI 関数を呼び出すことが許されてない環境での使用が推奨されており，複数スレッドからの MPI 関数呼出しをサポート（`MPI_THREAD_MULTIPLE`）する設定での利用は推奨されていない．その他のシステムでも，性能面等の問題から `MPI_THREAD_MULTIPLE` の利用は推奨されていないことが多い．そこで我々は，そのような環境でも動作する実装を行った．この実装は，実際に「京」上で動作し，7,168 ワーカを用いた実験で，19-女王全解探索問題において 4,615 倍の性能向上を得られることを確認している [10]．

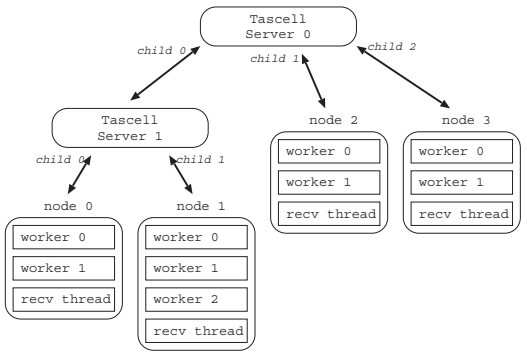

図 1 Tascell における計算ノード間の接続（TCP/IP 実装）

1 つのスレッドから同時に MPI 関数を呼べないという上記の制約下で，デッドロックを起こさずかつ効率的に通信レイヤの実装を行うためにはいくつかの工夫が必要となる．本稿では，この実装手法の紹介を行うとともに，本実装の東京大学情報基盤センターのスーパーコンピュータ Reedbush 上での性能評価の結果を報告する．また，Tascell の通信レイヤの実装に関する今後の展望についても議論する．

2. Tascell の概要

本節では，Tascell フレームワークについて，その初期の実装である TCP/IP ベースの実装の説明を交じえながら説明する．MPI 実装においても全体の動作自体は，ノード間通信が中継サーバ（Tascell サーバ）を介さずに行われることを除いてはほとんど同じである．

Tascell フレームワークは，Tascell サーバと Tascell 言語のコンパイラから構成される．コンパイルされた実行形式のプログラムが，1 つ以上の計算ノードで実行される．**図 1** に示すように，各計算ノードは共有メモリ環境内に 1 つ以上のワーカ

（通常は各ワーカがひとつの計算コアに割り当てられる）を持つ．また各ノードは TCP/IP で Tascell サーバに接続される．

負荷均衡化のため，仕事を持たない（アイドルな）ワーカは仕事を持つワーカにタスク要求を出す．ここで，要求先を特定のワーカとする場合と，ワーカを特定せずに要求を出す場合（any 要求）がある[*1]．ノード内に対する any 要求が発生した場合は，Tascell のランタイムが適切な戦略で負荷が高いと判断されるワーカを選択し，要求を転送する．any 要求がノード外，すなわち接続されている Tascell サーバに送られた場合は，Tascell サーバが負荷が高いと判断される計算ノードを選択し要求を転送する（送り先となったノード内で，ランタイムにより適切な要求先ワーカが選択される）．なお，このようなメッセージ転送は処理系により自動的に行われ，Tascell プログラマがメッセージを直接扱う必要はない（扱うことができない）．

タスク要求の返信として送られるタスクや，さらにその返信として送られるタスクの実行結果は，タスクオブジェクトとして転送される．タスクの授受がノード内で行われていた場合は，このタスクオブジェクトの授受はポインタのやりとりのみで高速に行われる．ノードを跨ぐタスク授受の場合は，タスクオブジェクトはシリアライズされたうえで Tascell サーバを介して転送される．

図 1 に示されるように，Tascell サーバを別の Tascell サーバに接続しツリーを構成することもできる．これを利用することで，複数の計算クラスタを WAN を介して接続するような広域分散環境でのプログラム実行も可能となる [12].

3. MPI による通信層の実装

この節では，MPI ベースの Tascell の通信層の実装の詳細を説明する．1 節で延べたように，この実装はサーバレス，すなわち**図 2** のように各計算ノードが MPI 通信により直接他ノードとメッセージを送受信するものとした．

さらにこの実装では，MPI に MPI_THREAD_FUNNELED のスレッドサポートレベル（メインスレッドしか MPI 関数呼出しができない）までしか要求しないこととし，また片方向通信は使わず両方向通信のみを用いることとした．したがって，従来の TCP/IP の実装で用いられていた send/recv のシステムコールを単純に MPI_Send/MPI_Recv に置き換えるということはできない．

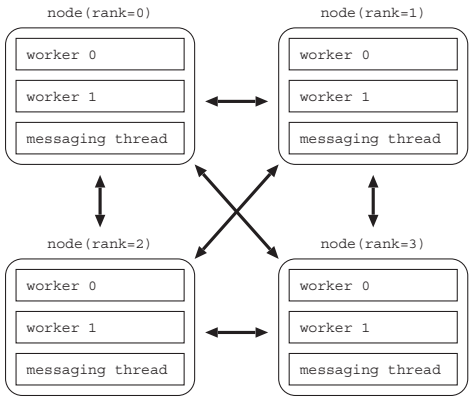

図 2 Tascell における計算ノード間の接続（MPI 実装）

図 2 に示されるように，各計算ノードは 1 つ以上のワーカに加え，1 つのメッセージ送受信スレッドを持つ．TCP/IP ベースの実装と同じく，単一ノード内でのワーカ間のメッセージ通信は共有メモリを介して行われる．外部ノードのワーカにメッセージを送る際は，送信ワーカ自身が MPI 関数を呼ぶのではなく，この送受信スレッドに送信を依頼する．メッセージ処理スレッドはまた，MPI 関数による外部ノードからのメッセージの受信および受け取ったメッセージに対する応答処理も受け持つ．

以下本節では，通信層の実装の詳細を説明する．

3.1 ノード指定方法

TCP/IP 実装では，外部ノードにメッセージを送る際の送信先は，送信元から送信先の経路を示す相対アドレスによって指定していた．一方 MPI 実装では，送信先ノードは MPI プロセスのランク番号によって直接指定する．

外部ノードに対するタスクの「any 要求」については，TCP/IP 実装ではその要求をどのノード

[*1] この 2 種類の使い分けの詳細については本稿では触れないので，文献 [7] などを参照されたい．

に転送するかを Tascell サーバが判断していたが，MPI 実装ではランダムに送信先を選択することとした．

3.2 デッドロック回避

一般に，ノード間通信を行うプログラムは，ブロッキング通信が別のブロッキング処理の完了を待ち合わせる必要が生じるとデッドロックを起こす可能性がある [9]．Tascell の実装においても，不注意にメッセージ通信を実装してしまう（たとえば，単純にメッセージ送受信スレッドにブロッキング通信の送信・受信関数を交互に呼び出させるような実装）とデッドロックを引き起こす．加えて，ワークスティールによる動的負荷分散を実現する Tascell では，各ワーカがいつでも任意の他ワーカに対してタスク要求などのメッセージを送る可能性があるため，そのような待ち合わせ関係を静的に解決するのは困難である．

TCP/IP 実装ではこのようなデッドロックは，Tascell サーバに各接続ノードに対してそれぞれ 1 つずつのメッセージ送信および受信スレッドを持たせることによって，全てのブロッキング通信が非同期に完了するようにすることで回避していた．しかし，同様の手法を MPI 実装で用いようとすると，通信相手となりうる各ノードに対して 2 つの送受信スレッドを立ち上げる必要があり，`MPI_THREAD_MULTIPLE` レベルのサポートが必要となるため本実装で採用することはできない．

単一の送受信スレッドのみでデッドロックを回避するため，本実装では送受信スレッドにノンブロッキングの MPI 送信関数でメッセージ送信を行わせ，ブロッキング MPI 受信関数の呼出し前に `MPI_IProbe` 関数でノンブロッキングで受信メッセージがあるかを確認させるという手法を採った．以下，詳細を説明する．

本実装における計算ノードは，1 つ以上のワーカスレッドと 1 つのメッセージ送受信スレッドを持つ．この送受信スレッドは，外部ノードとの通信処理を実行する責任を負う．各ワーカスレッドは，送信依頼キューにエントリを追加することで送受信スレッドにメッセージ送信を依頼することができる．送受信スレッドは以下の処理を繰り返し実行する．

(1) `MPI_Iprobe`（ノンブロッキング）の呼び出しにより外部ノードからのメッセージが届いているかどうかを確認する．もし届いていればそのメッセージを `MPI_Recv`（ブロッキング）により受信し，受信したメッセージの処理を行う（処理中に新たなメッセージ送信が必要になった場合，送信依頼キューへのエントリ追加のみを行い，その場では送信処理は行わない）．

(2) 一定時間 t_{slp} だけスリープする．

(3) 前回行った `MPI_Isend` によるメッセージ送信処理が完了しているかを `MPI_Test`（ノンブロッキング）により確認する．完了していれば，さらに送信依頼キューにエントリがあるかを確認し，もしあればキューからエントリをひとつ取り出し，その依頼メッセージを `MPI_Isend`（ノンブロッキング）により送信する．

図 3 に，メッセージ送受信スレッドが実行する手続きの擬似コードを示す．

この実装では，メッセージの送信をノンブロッキング関数で行い，かつ外部からのメッセージ受信を対応する送信メッセージが存在することを（ノンブロッキング関数で）確認してから実行するためデッドロックを回避できる．

t_{slp} はメッセージ確認のポーリング間隔を指定するもので，メッセージ送受信の遅延と送受信スレッドによる CPU 資源の消費量のトレードオフを考慮して決定する．

この実装の自明な問題点は，送受信メッセージ確認のためにビジーウェイトを行っていることである．しかし，`MPI_THREAD_MULTIPLE` のサポートレベルを要求せずにこのようなビジーウェイトを使わない実装を行うことは不可能である．このビジーウェイトは，`MPI_THREAD_SERIALIZED` レベル（同時に 1 つであればメインスレッド以外でも MPI 関数呼出しが行える）を要求しても回避できない．

その理由を以下で説明する．

我々の実装と同様に，送信依頼キューを利用するある実装を考える．メッセージ送受信スレッドは MPI_Waitany 関数を使うことでビジーウェイトを回避しつつメッセージの送信および受信の完了を同時に待つことは可能である．しかし，このスレッドは外部からのメッセージの到着（MPI_Recv または別の MPI_Waitany 呼出しなどを使うことになる）やワーカスレッドからの送信依頼キューへのエントリ追加（pthread_cond_wait などを使うことになる）をさらに同時に待つことはできない．

送信依頼キューを用いない実装では，各ワーカスレッド自身が MPI 関数を呼出すことでメッセージ送信を行う必要がある．この場合，メッセージスレッドは少なくとも外部ノードからのメッセージ受信のみを担当すればよい．しかし，このような実装は MPI_THREAD_SERIALIZED の制約によりデッドロックを引き起こす．メッセージスレッドが外部ノードからのメッセージ到着を待っている間，ワーカスレッドはメッセージ送信を行えないためである．

```
1  for (;;)
2  {
3    // Check whether there are any incoming
4    // messages without blocking.
5    MPI_Iprobe(...);
6    if ( any incoming messages? )
7    {
8      MPI_Recv(...); // Receive a message.
9      Handles the received message without MPI calls.
10   }
11   sleep(t_slp);
12   if ( sending_message )
13   {
14     MPI_Test(...);
15     if ( Is the previous MPI_Isend completed? )
16     {
17       sending_message = false;
18       Deallocate the send buffer.
19     }
20   }
21   else
22   {
23     if ( any entries in the send request queue? )
24     {
25       Dequeue an entry.
26       // Send a message.
27       MPI_Isend(...);
28       sending_message = true;
29     }
30   }
31 }
```

図 3 メッセージ送受信スレッドが実行する手続きの擬似コード．

4. 性能評価

本 MPI 実装の性能評価を，東京大学情報基盤センターのスーパーコンピュータ Reedbush-U の最大 16 ノードを用いて実施した．

評価に用いたプログラムは以下の通りである．

Fib(n)　n 番目の Fibonacci 数を doubly recursive アルゴリズムで求める．

Nq(n)　n 女王問題のバックトラック全解探索を行う．

Pen(n)　n ピースの Pentomino パズル問題のバックトラック全解探索を行う（$n > 12$ のときは余分なピースを追加し，盤面を拡張）．

Comp(n)　2 つの n 要素の配列間の全要素ペア（(a_i, b_j) for all $0 \leq i, j < n$）について比較演算を実行する．

Grav(n)　$(2n+1)^3$ 個の同一質量の質点からある一点にかかる総引力を計算する．

LU(n)　$n \times n$ の行列の LU 分解を cache-oblivious な再帰アルゴリズムで行う．

表 1　評価環境

	Reedbush-U SGI Rackable C2112-4GP3
CPU	Intel Xeon E5-2695v4 2.1 GHz 18-core × 2 (36 cores in total per node)
Memory	256 GB (153.6 GB/sec)
Network	InfiniBand FDR 4x (56 Gbps) × 2 Topology: Full-bisection Fat Tree
OS	Red Hat Enterprise Linux 7
Compiler	GCC 4.8.5 with -O3
Nested functions	Trampoline-based implementation in GCC
MPI	Intel MPI Library Version 2017 Update 3

なお，どのプログラムにおいても，タスクのサイズが cutoff 以下になったときに並列化を打ち切る（逐次計算手続きに切り替える）ような最適化は行っていない．

評価環境の詳細を**表 1** に示す．実行に用いるノード数は，1–16 の範囲で変化させ，各ノードにコア数と同数の 36 ワーカを立ち上げた（ワーカスレッドに加え，メッセージ送受信スレッドも起動している）．また比較および参考のため，1 ノード実行でワーカ数を 1–36 の範囲で変化させたときの性能，および C 言語による逐次実装の性能も測定した．

ポーリング間隔 t_{slp} は 20 µs に設定した．

測定結果を**図 4**（実行時間）および図 4（C 言語実装に対する理想の性能向上を 1 としたときの並列化効率）に示す．

LU を除くほとんどの測定でノード数追加による性能向上が得られているものの，ノード数を増やすにつれて並列化効率は大きく下がっている．この低下は文献 [12] 等で報告されている TCP/IP 実装のものより大きくなっている．この原因としては，送受信スレッドによるメッセージ送受信が追いつかず遅延が大きくなってしまったことや，Tascell サーバがなくなり外部ノードへのタスク要求をランダムに出すようにした結果，タスクを持たない計算ノードへのタスク要求メッセージが増えてしまったことなどが考えられる．これらの問題はポーリング間隔 t_{slp} の調整や，タスク要求先ノードの選択戦略の改善等により解決できる可能性があり，今後検討していく必要がある．

なお，LU ではノード数を増やしたときの性能向上が全く得られていない．これは一回のスティールあたりのタスクの入力および結果のためのデータ通信量が大きいためであり，ワークスティールベースの動的負荷分散では性能向上を得ることは基本的に困難である．文献 [7] 等の TCP/IP 実装の評価では，ノード数を増やすことで逆に性能が低下してしまっており，それに比べれば MPI の通信性能の恩恵により良好な性能が得られているとは言える．

5. 関連研究

分散メモリ環境に対応したタスク並列言語である X10 [11] のノード間通信は MPI で実装されており，`MPI_THREAD_SERIALIZED` 対応および `MPI_THREAD_MULTIPLE` のサポートまで利用した実装の両バージョンが存在する．`MPI_THREAD_SERIALIZED` 対応版では，MPI 関数の呼出しに対して排他制御を行う．

Uni-threads [2, 3] は分散環境でノード間のワークスティールにも対応したタスク並列ライブラリであるが，ノード間通信は MPI ではなく，RDMA あるいは GASNet [4] を用いた実装となっている．

6. 今後の展望

6.1 MPI 片方向通信による実装

最近では，`MPI_THREAD_MULTIPLE` スレッドサポートレベルに加え，MPI-3 の片方向通信が利用可能になっているシステムも増えてきているため，これらのサポートを前提とした実装も有用であると考えられる．これらのサポートを利用すれば，各ワーカスレッドは片方向通信によりノンブロッキングでメッセージ送信を行い，メッセージスレッドは受信メッセージのみをビジーウェイトなしに監視・処理するという実装が可能になると期待できる．ただし，片方向通信自体が MPI ライブラリによるポーリングにより実装されている可能性はあり，その場合は必ずしも本稿で紹介した実装より良い性能が得られるとは限らないため，比較評価は必要になると考えられる．

6.2 ACP ライブラリの利用

現状の Tascell のプログラミングモデルでは，ノード間の通信はこれまでに述べた処理系内部でのタスクの送受信に関するものに限られ，Tascell プログラムでタスク授受以外のタイミングでのデータ通信を記述することは基本的にはできない．たとえば実用的なグラフ探索等のアプリケーションでは，探索空間削減のために必要なテーブル情報などを外部計算ノードを含むワーカ間で共有したいことがあり，そのためタスク実行途中での他ノードとのデータ通信を可能とするできるようなプログラミングモデルへの拡張は必要であると考えている．

しかし，上記のグラフ探索の例で必要となるような，ノード間で共有されるテーブルへの非同期な書

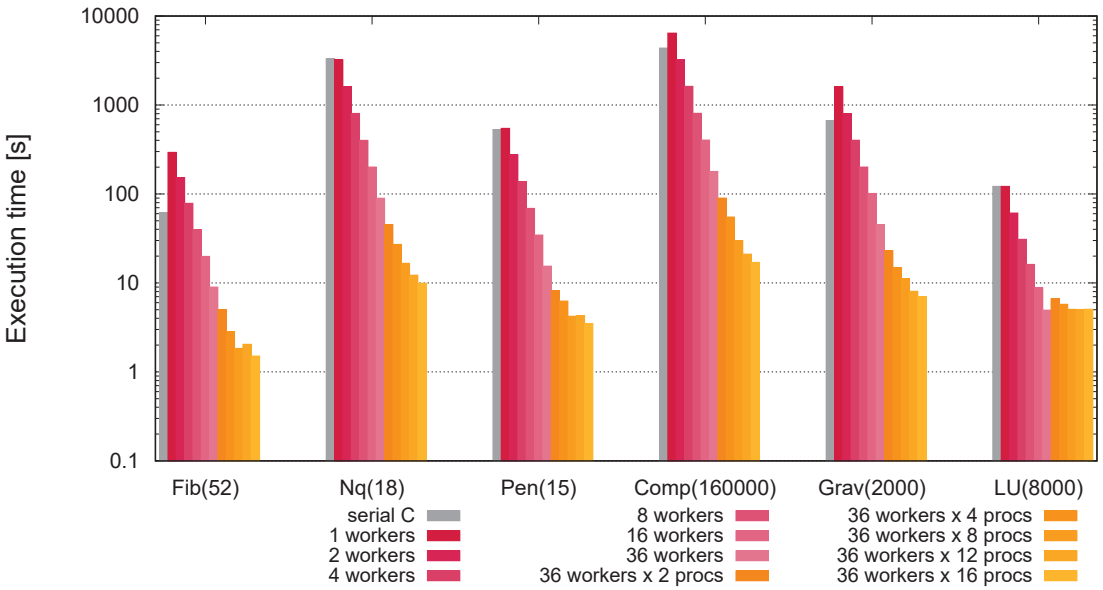

図 4　性能測定の結果（実行時間）

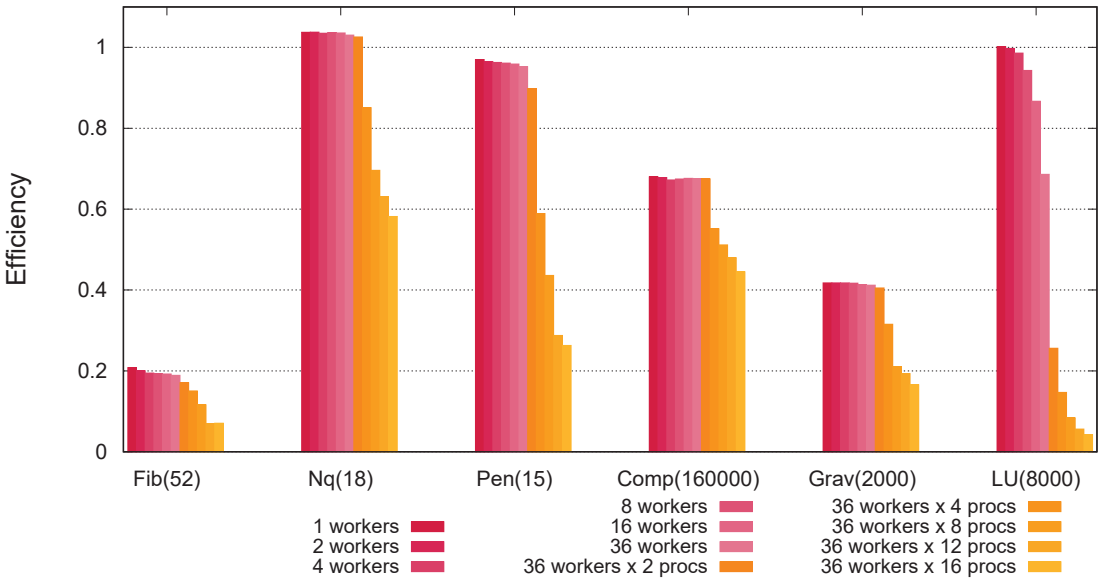

図 5　性能測定の結果（並列化効率）

き込みを MPI で実現しようとすると，実装が複雑になりすぎると考えられる．そこで我々は，MPI より柔軟な通信モデルに基づくライブラリである Advanced Communication Primitives（ACP）[1] ライブラリを利用することを検討している．このライブラリは，プロセス間で共有されるグローバル空間を提供し，その空間に対するアトミック操作を含む非同期なデータアクセスを簡便に記述できるようにしており，上記の目的に合致する．また，Infiniband，Ethernet，Tofu 等のインターコネクトの種類ごとの低レベルな通信インターフェースの違いを隠蔽しているため，可搬性も維持する

ことができる．

ワーカ間のメッセージ処理においても，タスクオブジェクトのデータの送信を必要時まで遅延させるなど，本ライブラリを利用した性能改善が実現できる可能性がある．

7. まとめ

本稿では，分散メモリ環境での動的負荷分散に対応したタスク並列言語 Tascell におけるノード間通信の実装について紹介した．また，MPI 版実装の Reedbush-U 環境での性能評価を示し，残された課題と今後の展望について述べた．

本研究では，実用的なグラフ探索などのアプリケーションの大規模並列環境での高性能実装が可能となるタスク並列処理系の実現を目指しており，そのために今後も通信機能の改善を中心に開発を進めていく予定である．

なお，Tascell 処理系は `https://bitbucket.org/tasuku/sc-tascell/` で公開している．

謝辞 本研究の一部は科学研究費基盤研究（B）「計算状態の精密操作に基づく高性能・高信頼システム技術」（26280023）および基盤研究（C）「グラフ探索アプリケーションの大規模並列環境での高性能化に向けた並列言語の開発」（17K00099）の助成を得て行った．

参考文献

[1] ACE Project: ACE (Advanced Communication for Exa) Project. `http://ace-project.kyushu-u.ac.jp/main/jp/01_overview/index.html`.

[2] Akiyama, S. and Taura, K.: Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing, *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '15*, pp. 15–26 (2015).

[3] Akiyama, S. and Taura, K.: Scalable Work Stealing of Native Threads on an x86-64 Infiniband Cluster, *Journal of Information Processing*, Vol. 24, No. 3, pp. 583–596 (online), DOI: 10.2197/ipsjjip.24.583 (2016).

[4] Bonachea, D.: GASNet Specification, V1.8.1, Technical report, Berkeley, CA, USA (2017).

[5] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *SIGPLAN Not.*, Vol. 40, No. 10, pp. 519–538 (online), DOI: http://doi.acm.org/10.1145/1103845.1094852 (2005).

[6] Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI '98)*, Vol. 33, No. 5, pp. 212–223 (1998).

[7] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, pp. 55–64 (2009).

[8] Intel Corporation: A quick, easy and reliable way to improve threaded performance—Intel Cilk Plus. `https://software.intel.com/en-us/intel-cilk-plusx`.

[9] Luecke, G., Zou, Y., Coyle, J., Hoekstra, J. and Kraeva, M.: Deadlock Detection in MPI Programs (2002).

[10] Muraoka, D., Yasugi, M., Hiraishi, T. and Umatani, S.: Evaluation of an MPI-Based Implementation of the Tascell Task-Parallel Language on Massively Parallel Systems, *Proceedings of the 45th International Conference on Parallel Processing Workshops (ICPPW 2016) (Ninth International Workshop on Parallel Programming Models and Systems Software for High-End Computing P2S2 2016, held in conjunction with ICPP 2016)*, pp. 161–170 (2016).

[11] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: *X10 Language Specification Version 2.6* (2016). `http://x10.sourceforge.net/documentation/languagespec/x10-260.pdf`.

[12] 平石 拓，八杉昌宏，馬谷誠二：動的負荷分散フレームワーク Tascell の広域分散およびメニーコア環境における評価，先進的計算基盤システムシンポジウム（SACSIS2011），pp. 55–63 (2011).