pollingと軽量スレッドによる高速な待ち合わせ処理

田邨 優人[1,a)]　中島 耕太[1,b)]　山本 昌生[1,c)]　前田 宗則[1,d)]

概要：ビッグデータ処理の高速化を目指し，全てのデータをメインメモリ上で処理するインメモリコンピューティングがビジネスアプリケーションにおいて台頭している．高多重となるビジネスアプリケーションをインメモリで実行する際には，従来より用いられてきた割り込みやシグナルよりも高速なスレッド間・プロセス間での待ち合わせ処理が必要となる．割り込みやシグナルよりも高速な待ち合わせ手法にpollingがあるが，pollingを用いるとスレッドがCPUコアを占有してしまうため，待ち合わせを必要とするスレッドの数がCPUコア数を超えると適切に待ち合わせを行えない．その問題を回避するために，複数のスレッドのpollingを1つのスレッドに代行させるpolling代行という仕組みを用いたとしても，その仕組みの中でシグナルを用いるためpollingによる高速性を失ってしまう．そこで本稿では，polling代行の際のシグナル処理を軽量スレッド切り替えに置き換えることで，pollingの高速性を維持しながら高多重なスレッドの待ち合わせに対応する手法を提案する．スレッド間のデータ送受信の片道レイテンシにより提案手法を評価したところ，従来のシグナルによる手法よりも約5.21倍高速であることを確認した．

Abstract: In-memory computing, by which all data are processed in main memory for high performance processing of big data, have become popular for business computing. In-memory execution of business applications have many processes and thus the conventional method of waiting for the completion of inter-process communication has to be reexamined. The polling method delivers higher performance than the conventional interruption signal method. However, since the number of threads that can use the polling method at the same time is limited by the number of CPU cores, polling is not appropriate when the number of waiting threads exceeds the number of CPU cores. And a method where a polling agent thread polls multiple threads needs to use interruption and thus its performance would be degraded. In this paper, we propose a novel waiting method based on polling and lightweight thread that aims to achieve the same high performance of the polling method regardless waiting for the completion of many threads. We evaluate the performance of the proposed inter-thread communication and show that the new waiting method is 5.21 times faster than the conventional method of POSIX real time signal.

1. はじめに

近年，大規模データ処理の高速化を目指し，全てのデータをメインメモリ上で処理するインメモリコンピューティングがデータベースを中心とするビジネスアプリケーションで台頭している．多くのビジネスアプリケーションは取り扱うデータをデータベースに格納し，システム全体を構成している．従来のデータベースシステムは大半のデータを外部記憶装置に配置しており，外部記憶装置へのデータI/Oがしばしば性能のボトルネックとなっている．これに対し，SAP社製インメモリデータベースSAP HANA [1]に代表されるインメモリDBが近年広がりを見せている．インメモリコンピューティングでは，データI/O処理がメインメモリへのロードストアに置き換わる．つまり，I/O処理のレイテンシ（1ms〜100ms）がメモリのレイテンシ（50ns〜100ns）に置き換えられるため，データ処理の大幅な高速化が実現できる．

このインメモリコンピューティングによる性能をビジネスアプリケーションで十分に発揮するには，割り込みやシグナルよりも高速なプログラム間での待ち合わせ手法が求められる．昨今のビジネスアプリケーションは高多重度なマルチプロセス・マルチスレッドであることが一般的で，その中では多くのプロセスやスレッドが割り込みやシグナルによる待ち合わせ処理を用いて連携し合っている．割り込みやシグナルのレイテンシ（約1μs）はI/O処理のレイテンシに対しては十分に小さいため，従来の外部記憶装置を用いたデータ処理では問題視されることはなかった．しかし，I/O処理のレイテンシがメモリのレイテンシに置き換わるインメモリコンピューティングにおいては，割り込

[1] 株式会社富士通研究所
Fujitsu Laboratories Ltd, Kawasaki 211-8588, Japan
[a)] tamura.yuto@ip.fujitsu.com
[b)] nakashima.kouta@jp.fujitsu.com
[c)] masao.yamamoto@jp.fujitsu.com
[d)] maeda.munenori@jp.fujitsu.com

みやシグナルのレイテンシはメモリレイテンシよりも 10 倍以上大きいため問題となる．例えば，インメモリ DB を持つサーバへクライアントがアクセスする分散システムにおいては，数十 ns 程の DB 処理のために，最低でも数 µs 程かかるプロセス間通信を行なわなければならない場合が生じてしまう．そのため，インメモリコンピューティングの性能をビジネスアプリで十分に発揮させるためには割り込みやシグナルよりも高速な待ち合わせ手法が必要となる．

本稿では，インメモリでビジネスアプリケーションを実行する際に求められる高速な待ち合わせ手法として，polling 代行と軽量スレッドを組み合わせた待ち合わせ手法の提案を行う．高速な待ち合わせ手法である polling をベースにし，polling が苦手とする高多重度なスレッドの待ち合わせを polling 代行という仕組みにより実現する．また，polling 代行の仕組みにより，待機スレッドを再開させる処理が新たに必要となってしまうが，この処理にかかる時間を軽量スレッド化により削減することで polling 本来の高速性を維持する．この提案手法を用いることでビジネスアプリケーションの実行においてもインメモリコンピューティングの高速性を十分に引き出すことが可能となる．

以降の章では，2 章にて従来手法の課題を明確にし，それに対する提案を 3 章にて行う．そして 4 章では提案手法の有効性を確認するための評価実験の内容とその結果を示す．その後 5 章にて関連研究を述べてから 6 章で本論文の結論を述べる．

2. 従来の待ち合わせ手法の課題

2.1 待ち合わせ処理

高多重なマルチプロセス・マルチスレッドのプログラムにおいては，多くのプロセス間，スレッド間でデータ送受信や同期が行われる．その際には，受信すべきデータの到着を待つ，同期するべきスレッドの処理が終わるまで待つといった，特定のイベントを待たなければいけない状況が生じる．このような場合，処理の実行を一時中断して待機状態に遷移し，イベントの発生によって待機状態から復帰して処理を再開する．このような一連の処理を本稿では待ち合わせ処理と呼ぶ．

従来，汎用 OS が提供する割り込みやシグナルが待ち合わせ処理に一般的に使用されてきた．なぜならば，これらの手法は多くのプロセスやスレッドが動く状況でも適切に待ち合わせが管理され，かつ性能のボトルネックであるディスクの I/O 処理のレイテンシと比較して十分に高速なものだからである．ただ，インメモリコンピューティングにおいては I/O 処理は全てメインメモリへのロードストアに置き換えられる．メインメモリへのロードストアにかかるレイテンシが 50ns〜100ns であるのに対して，割り込みやシグナルによる待ち合わせのレイテンシは約 1µs と 10 倍以上大きい．そのため，インメモリコンピューティングの性能を十分に引き出すためには，従来よりもレイテンシが短い待ち合わせ手法が必要となる．

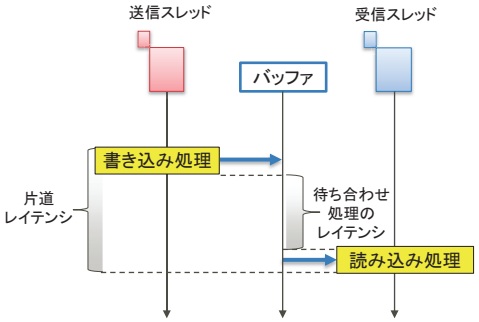

図 1　データ送受信における片道レイテンシ

待ち合わせ手法には割り込みやシグナル以外に，よりレイテンシが短い polling がある．以降では，これらの異なる手法の性能を，スレッド間でのデータ送受信の片道レイテンシにより比較しながら議論する．

2.2 待ち合わせ処理の性能測定方法

異なる待ち合わせ手法の性能を比較するために行った，スレッド間でのデータ送受信の片道レイテンシの計測方法について述べる．ここでの片道レイテンシはスレッド間での共有バッファを介した一度のデータ送受信にかかる時間である．具体的には図 1 に示すように，送信スレッドがバッファへデータを書き込む時間と，待ち合わせのレイテンシと，受信スレッドがバッファからデータを読み込む時間の合計である．

2.2.1 スレッド間でのデータ送受信方法

図 2 に，実験で行ったスレッド間でのデータ送受信の概要図を示す．送受信を行う全てのスレッドはそれぞれ固有のバッファ領域を持ち，そのバッファを介してデータを送受信する．各スレッドは自バッファに別のスレッドからのデータが到着したことを知ると，バッファからデータを受信し，そのデータを隣りのスレッドのバッファ領域に対して送信するということを繰り返す．この時のスレッド間での送受信の順序制御を待ち合わせ処理によって行う．また本実験では，送受信するデータサイズを 8 バイトに設定した．これによって待ち合わせのレイテンシが片道レイテンシの大半を占めるため，片道レイテンシの比較により各待ち合わせ手法の性能比較ができる．さらに，データ送受信を行うスレッドの総数を任意に設定して計測を行い，多重度についての性能比較も行う．

2.2.2 片道レイテンシの計測方法

ある 1 つのスレッドがデータを送信してから再びデータを送信するまでの時間を計測し，これを総スレッド数で割った値を片道レイテンシの平均値として評価に使用した．各スレッドは一度データを送信すると，他の全てのスレッ

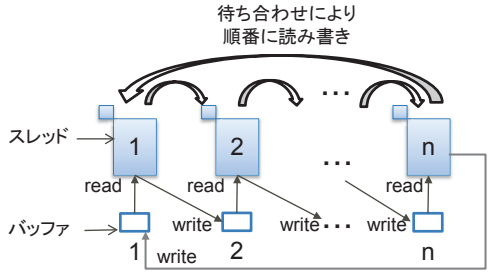

図 2　n スレッド間でのデータ送受信

表 1　評価用計算機の仕様

CPU	Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz
OS	Red Hat Enterprise Linux Server release 6.4
カーネル	2.6.32-358.el6.x86_64

ドで一度ずつデータ送受信が行われるまで順番を待つことになる．つまり，ある 1 つのスレッドが一度データを送信してから再びデータを送信するまでの時間が，各スレッドでの片道レイテンシの総和ということになる．そのため，上記の算出法で片道レイテンシの平均値を求めることができる．

また，片道レイテンシを測定した計算機環境については**表 1** に示す．

2.3　シグナルと polling の性能比較

シグナルによる待ち合わせと比較してレイテンシが短い手法に polling がある．シグナルの場合はシステムコール処理や idle 状態からスレッドを復帰させるためのスケジューリング処理が行われ，それらの処理時間の合計がレイテンシとなる．polling の場合は，CPU コアを占有したままイベントの発生を示すメモリ領域を常に監視し，書き換えを検知すればそのまま元の処理を再開する．つまり，メインメモリへアクセスして読み書きする処理時間のみがレイテンシであるため，シグナルと比較してレイテンシが大幅に短い．実際に，4 コアのマシンにおいて 4 スレッド間でデータ送受信を行った時の片道レイテンシを計測したところ，polling で待ち合わせを行った方がシグナルよりも約 19 倍高速であった（**図 3** 参照）．しかし，polling はスレッドが CPU コアを占有してしまうため，待ち合わせを必要とするスレッドがコアの数を超えた場合に，待ち合わせできないスレッドがでてくる．そして，待ち合わせできないスレッドはラウンドロビンでスケジュールされるのを待つことになるため，イベントに対して即時的な対応ができず，大幅に遅くなる．**図 4** は 4 コアのマシンにおいて 5 スレッド間でデータ送受信を行った時の片道レイテンシを示しており，polling の方がシグナルよりも 2,557 倍遅くなることが分かる．

この問題に対して，**図 5** に示すような，複数スレッド

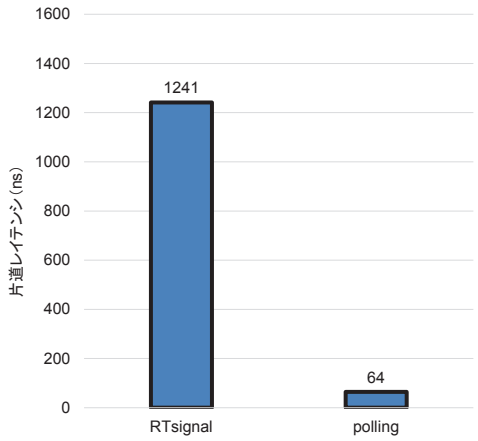

図 3　片道レイテンシの比較（4 コア・4 スレッド）

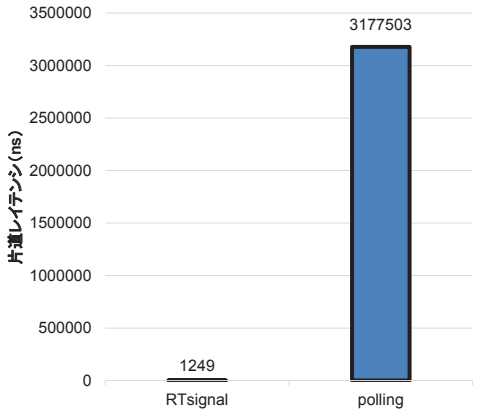

図 4　片道レイテンシの比較（4 コア・5 スレッド）

の polling を代行する 1 つのスレッド（以降，polling 代行スレッドと呼ぶ）を用いた手法も考えられる．この手法ではそれぞれのスレッドは polling 代行スレッドに対して polling を依頼し，待機状態に遷移する．そして，polling 代行スレッドは各スレッドに関してのイベントを監視し，イベント発生を検知すると当該スレッドを再開させる．この手法では polling するスレッドは 1 つだけなので，コア数を超えるスレッドが待ち合わせを行うことが可能である．一方，polling 代行スレッドが当該スレッドを再開させる際には，シグナルを用いるので 1μs ほどの遅延が生じる．このため，**図 6** に示すように，シグナルによる待ち合わせと同程度以上のレイテンシとなってしまう．

2.4　課題

以上で述べてきたことをまとめると，待ち合わせを必要とするスレッドがコア数よりも多い状況で待ち合わせ処理を高速化することが必要である．従来より利用されてきた

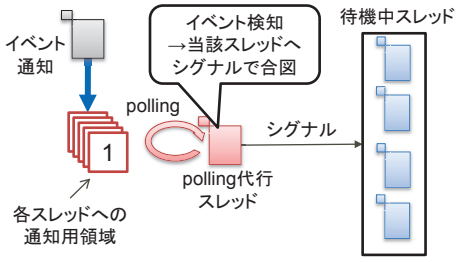

図 5　polling 代行スレッドを用いた待ち合わせ

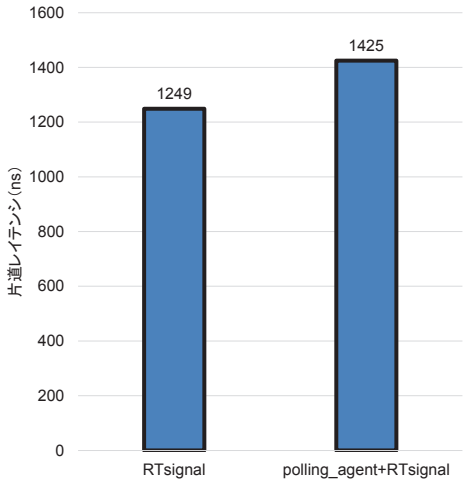

図 6　片道レイテンシの比較（4 コア・5 スレッド）

シグナルのレイテンシは 1μs 以上であり，インメモリコンピューティングではそれ以上に高速な待ち合わせ手法が求められる．また，polling はレイテンシが短く高速であるが，スレッドが CPU コアを占有してしまうため，コア数を超えるスレッドは待ち合わせできない．これを解決するために複数スレッドの polling をまとめて代行する polling 代行スレッドを用いる手法も考えられる．しかし，待機状態のスレッドを再開させるためのシグナル処理に時間がかかり，polling 本来の高速性を妨げてしまうという課題がある．

3. 提案手法

我々は，前章で示した課題に対して，polling 代行の仕組みと軽量スレッドとを組み合わせた待ち合わせ手法を提案する．提案手法では以下の 2 点がポイントとなる．

- 軽量スレッドによる polling 代行機構
- 複数の polling 代行スレッド生成

提案する待ち合わせ処理機構の全体構成図を**図 7** に示す．以下では提案する機構についての説明を行う．

3.1 軽量スレッドによる polling 代行機構

マルチスレッドの管理をカーネルレベルではなく，ユーザレベルのライブラリで軽量に行うことでソフトウェアの

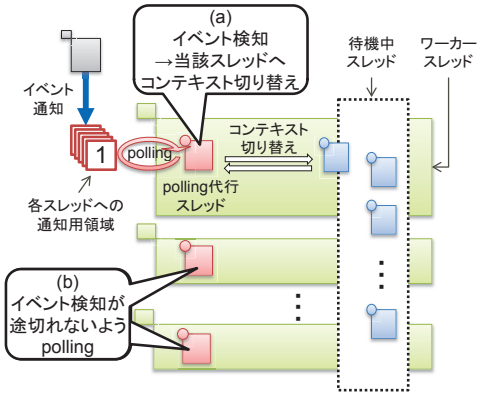

図 7　提案手法の全体構成図

性能向上を目指す手法がある．本稿ではその際に管理されるスレッドをカーネルレベルで管理されるスレッドと対比して軽量スレッドと呼ぶ．軽量スレッドの処理系では，各 CPU コアと 1 対 1 で結びつくカーネルスレッド（以降，ワーカースレッドと呼ぶ）上で軽量スレッドが実行される．各ワーカースレッドはユーザレベルのライブラリを介して，実行する軽量スレッドのコンテキストを切り替えることができる．この時のコンテキスト切り替えが，従来の OS が管理するスレッドのコンテキスト切り替えよりも高速であるため，高速かつ高多重度な軽量スレッド実行を実現できる．

提案手法においては，待ち合わせを必要とするスレッド及び polling 代行スレッドを軽量スレッド化する．polling 代行スレッドは，イベントの発生を検知すると，当該スレッドを再開させなければならない．この時，従来はシグナルを用いて再開させていたが，提案手法では軽量スレッド化された当該スレッドへの高速なコンテキスト切り替えにより再開させる（図 7 の a 参照）．これによって当該スレッドを再開させるのにかかる時間を大幅に削減でき，2.4 節で述べた課題を解決できると考える．

3.2 複数の polling 代行スレッド生成

コンテキストの切り替えにより，当該スレッドが高速に再開される一方で polling 代行スレッドによるイベント検知は停止してしまう．そこで，あらかじめ別の polling 代行スレッドを生成して実行しておくことでイベント検知を継続する（図 7 の b 参照）．この時，複数の polling 代行スレッドが同時に動くことになり，その数は要求される処理性能や省電力性能に応じて適切に設定されるべきである．提案手法では，polling 代行スレッドは処理系で動くワーカースレッドの数だけ生成し，各ワーカーで polling の必要性を判定しながら polling を行うようにする．これによって，同時に polling するスレッド数を柔軟に調節できる．また，polling 代行スレッドとワーカースレッドとを 1 対 1 で結びつくように管理することでワーカースレッド

上での polling 代行スレッドについてのスケジューリングコストを小さくする．

3.3 ワーカースレッドの処理フロー

提案手法において，ワーカースレッドがどのような手順でスレッドの待ち合わせ処理を行うかについて述べる．ワーカースレッド上で実行していたスレッドがイベントの発生を待つことになると，ワーカースレッドはそのスレッドが待機状態に入ることを記録し，スレッドのコンテキストを保存してからワーカースレッド固有の polling 代行スレッドへとコンテキストを切り替える．polling 代行スレッドを実行する際には，まず他のワーカースレッドの polling の状況を確認し，さらなる polling が必要かを判定する．polling が必要であれば，各スレッドごとのイベント通知用領域に対して polling を開始する．そして，あるスレッドの通知用領域が書き換えられているのを検知すると，待機中の当該スレッドへコンテキストを切り替えることで当該スレッドを再開する．またこの時，polling を中断したことを別のワーカースレッドが確認できるように通知しておく．

4. 提案手法の性能評価

我々は前章で polling 代行による待ち合わせを高速化する手法の提案を行った．本章では，2.4 節で述べた課題を達成するかについて評価するべく，提案手法と従来の polling 代行による待ち合わせとの性能比較を行い，さらにシグナルによる待ち合わせとの性能比較も行う．

4.1 性能評価環境

本論文で性能評価に用いた計算機の仕様は表 1 と同じである．この計算機上でスレッド間でのデータ送受信を行うマルチスレッドプログラムを実行し，片道レイテンシを計測することで評価を行った．

4.2 実験方法

4.2.1 シグナルによる待ち合わせ手法

本実験では，スレッド間でのシグナルに，POSIX リアルタイムシグナルを用いた．送信側スレッドはバッファへのデータ書き込み後，pthread_kill により受信側スレッドに対してシグナルを送る．受信側スレッドは事前に sigwait を行っておき，送信側スレッドからのシグナルを受けて待機状態から復帰し，バッファからデータを読み出す（図 8 参照）．また，各スレッドがどの CPU コアに実行されるのかについては，OS が標準で持つスケジューラにより決定される．

4.2.2 polling とシグナルによる待ち合わせ手法

polling とシグナルによる待ち合わせ処理の概要図を図 9 に示す．送信側スレッドはバッファへのデータ書き込み後，polling 代行スレッドが監視を行っている領域を書き換えることで通知を行う．この通知を polling で検知した polling 代行スレッドは受信スレッドへシグナルで通知を行い，受信スレッドを待機状態から復帰させる．この際のシグナルには POSIX リアルタイムシグナルを用いる．また，polling 代行スレッドに対しては専用の CPU コアを 1 つ割り当て，残りの CPU コアにより他のスレッドの実行を行う．この時，polling 代行スレッド以外の各スレッドがどの CPU コアに実行されるのかについては，OS が標準で持つスケジューラにより決定される．

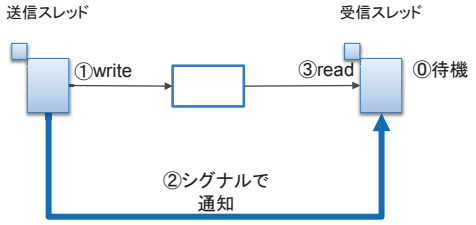

図 8 シグナルによる待ち合わせ

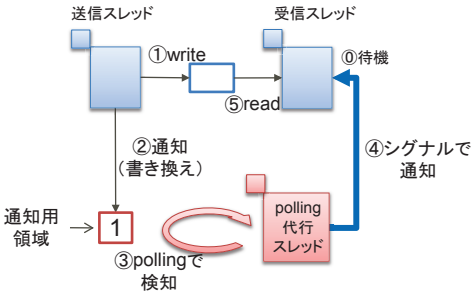

図 9 polling とシグナルによる待ち合わせ

4.2.3 polling と軽量スレッドによる待ち合わせ手法

polling と軽量スレッドによる待ち合わせ処理の概要図を図 10 に示す．送信側スレッドからの通知を polling で検知すると，ワーカースレッドは polling 代行スレッドのコンテキストを退避，受信スレッドのコンテキストを復元してバッファからデータを読み出す．ワーカースレッドは CPU コアの数だけ生成した．また，同時に polling を行うワーカースレッドの数については，常に 1 つになるようロックによる排他制御を行った．ロックが開放されるまでスピンで待機することで，polling 処理の排他制御にかかるオーバヘッドを最小にした．

4.3 実験結果

4 コアのマシンにおいて 5 スレッド間でデータ送受信を行った際の片道レイテンシの結果を図 11 に示す．実験結果を見ると，従来の polling 代行とシグナルを組み合わせる手法よりも，polling 代行と軽量スレッドを組み合わせる提案手法の方が約 5.95 倍高速であり，2.4 節で述べた課題を達成していることが確認できる．さらに，提案手法とシ

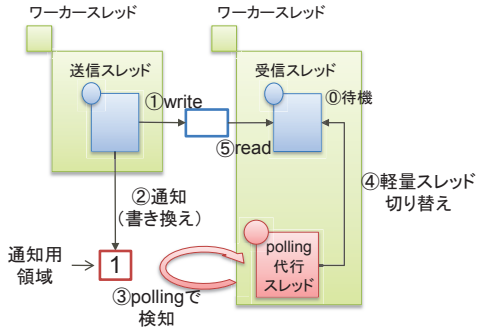

図 10　polling と軽量スレッドによる待ち合わせ

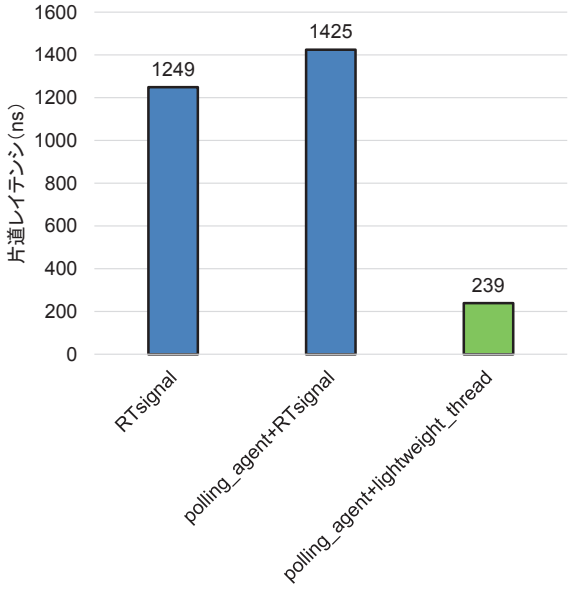

図 11　従来手法と提案手法の速度性能比較（4 コア・5 スレッド）

グナルとを比較すると提案手法が約 5.21 倍高速であることを確認した．

また，高多重度にスレッドが待ち合わせを行う時の性能を確認するべく，10，100，1000 スレッド間でデータ送受信を行った際の片道レイテンシについても測定を行った．その時の結果を図 **12** に示す．実験結果を見ると 1000 スレッドが待ちあわせを行う状況においても，提案手法が従来手法よりも約 5.24 倍高速であることが確認できる．

5. 関連研究

データ I/O 処理をマイクロ秒以下の低レイテンシで実現するデバイスを使用する状況においては，従来より割り込みで行っていた I/O 処理の完了管理を polling で行うことで性能が向上する [2–4]．本研究も割り込み・シグナルで行っていた待ち合わせ処理を polling で行うことで性能の向上を狙うものである．我々はそれに加えて，コア数以上のスレッドが待ち合わせを行う状況においても性能が向上する手法の提案を行った．

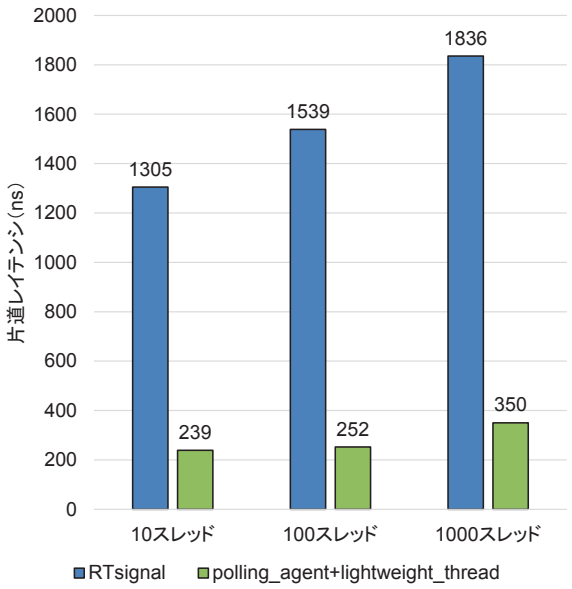

図 12　スケール時の従来手法と提案手法の速度性能比較（4 コア）

OS レベルのスレッド管理を使用せず，ユーザレベルでスレッド管理を行うことでマルチスレッドプログラムの高速化を目指す手法がある．この時に管理されるスレッドは軽量スレッドと呼ばれ，その処理系についてこれまでいくつか提案がされている [5–11]．軽量スレッドを用いることで，マルチスレッドプログラムの処理において以下の 2 つの利点が得られる．

- システムコールを必要としないためスレッド管理によるオーバヘッドが少ない
- アプリケーションの特性に応じた柔軟なスケジューリングを行える

本研究では，polling 代行の仕組みを軽量スレッドにより行うことを提案し，スレッド管理にかかるオーバヘッドが少ない利点により，待ちあわせ処理の高速化を達成した．

6. おわりに

インメモリでのビジネスアプリケーション実行においては，スレッド間・プロセス間での待ち合わせについて，従来の割り込みやシグナルよりも高速な手法が求められる．本稿では，高多重にスレッドが通信を行う状況での高速な待ち合わせ手法として，polling 代行と軽量スレッドを組み合わせた手法を提案した．高速な待ち合わせ手法である polling をベースにし，polling が苦手とする高多重度でのスレッド待ち合わせを polling 代行という仕組みにより実現した．また，polling 代行を行う際には，当該スレッドを再開させるのにかかるオーバヘッドが新たに発生するため，このオーバヘッドを軽量スレッド化により削減した．この方式を実装し，評価したところ，従来の待ち合わせ手法であるシグナルと比べて約 5.21 倍高速であることを確

認した．

本稿においては自作のマルチスレッドプログラムを用いて，提案手法の待ち合わせ時のオーバヘッドについて評価を行った．今後の課題として，高い実行性能を実現するための複数コアにおける軽量スレッドのスケジューリング手法についての検討，実アプリケーションへの適用と詳細な解析による有効性の検証がある．

参考文献

[1] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S. and Lehner, W.: SAP HANA Database: Data Management for Modern Business Applications, *SIGMOD Rec.*, Vol. 40, No. 4, pp. 45–51 (online), DOI: 10.1145/2094114.2094126 (2012).

[2] Yang, J., Minturn, D. B. and Hady, F.: When Poll is Better Than Interrupt, *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, USENIX Association, pp. 3–3 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2208461.2208464⟩ (2012).

[3] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J. and Swanson, S.: Providing Safe, User Space Access to Fast, Solid State Disks, *SIGARCH Comput. Archit. News*, Vol. 40, No. 1, pp. 387–400 (online), DOI: 10.1145/2189750.2151017 (2012).

[4] Wei, M., Bjørling, M., Bonnet, P. and Swanson, S.: I/O Speculation for the Microsecond Era, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USENIX Association, pp. 475–481 (online), available from ⟨https://www.usenix.org/conference/atc14/technical-sessions/presentation/wei⟩ (2014).

[5] 中島 潤, 田浦 健次朗: 高効率な I/O と軽量性を両立させるマルチスレッド処理系, 情報処理学会論文誌プログラミング（*PRO*）, Vol. 4, No. 1, pp. 13–26 (online), available from ⟨http://ci.nii.ac.jp/naid/110008616663/⟩ (2011).

[6] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *SIGPLAN Not.*, Vol. 30, No. 8, pp. 207–216 (online), DOI: 10.1145/209937.209958 (1995).

[7] BSC: Nanos++, (online), available from ⟨http://pm.bsc.es/projects/nanox⟩.

[8] Lea, D.: A Java Fork/Join Framework, *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, New York, NY, USA, ACM, pp. 36–43 (online), DOI: 10.1145/337449.337465 (2000).

[9] Pheatt, C.: Intel®; Threading Building Blocks, *J. Comput. Sci. Coll.*, Vol. 23, No. 4, pp. 298–298 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1352079.1352134⟩ (2008).

[10] Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *SIGPLAN Not.*, Vol. 34, No. 8, pp. 60–71 (online), DOI: 10.1145/329366.301110 (1999).

[11] Wheeler, K., Murphy, R. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8 (online), DOI: 10.1109/IPDPS.2008.4536359 (2008).