修士論文

分散スレッドスケジューラと協調する分散共有メモリライブラリの設計と実装

Design and Implementation of a Distributed Shared Memory Library Cooperating with Distributed Thread Scheduler

平成29年2月3日提出

指導教員　田浦 健次朗 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻
48-156410　遠藤 亘

高性能な大規模並列分散計算機は，現代の科学技術分野の進歩を支える社会基盤の一つであり，その計算速度向上のために多大な労力が費やされてきた．現在でも計算ノード数や CPU コア数は増大を続けており，ハードウェアの進歩は続いている．その一方で，そのような並列分散計算機を効率的に活用するためのソフトウェアランタイムシステムは古くから研究されているが，未だに決定版といえるモデルと実装が存在せず，ハードウェアの更新毎にアプリケーションプログラマが手動でチューニングすることでようやく高性能なプログラムが実現できる，という状況は数十年前から変わっていない．

分散型計算機に共有メモリモデルを実現するシステムは，分散共有メモリ (Distributed Shared Memory, DSM) と呼ばれる．DSM は，分散システム上で大域アドレッシングとコヒーレントキャッシュを提供することで，あたかも集中型システム上でプログラムが動作しているかのように振る舞う．DSM 上でのプログラミングモデルは共有メモリと完全に同一であり，既存のマルチスレッドアプリケーションをそのまま動作させることができる利点を持つ．一方で，2000 年代以降はコヒーレントキャッシュのスケーラブルな実装が困難であると認識されるようになり，コヒーレントキャッシュを持たない Partitioned Global Address Space (PGAS) のようなモデルが注目されるようになった．しかし，これまで理論的にコヒーレントキャッシュがスケーラブルでないと証明されたことはなく，近年のマルチコアプロセッサ研究で計算量削減のための新たなアイディアも提案されている．共有メモリモデルがもたらす高生産性を諦めなければならないほどの重大な問題は，理論面からも実装面からも存在しないのではないか，という観点が，本研究の着眼点の一つである．

並列計算機の実行モデルとしては，動的負荷分散を自動で行えるようなスレッド処理系が有用である．ワークスティーリングのように，理論的にも計算量の面でスケーラブルであることが知られていて，共有メモリシステム上で実用化されているスケジューリング手法も存在する．高速なスレッド処理系の実装の鍵となるのが，スレッドの実行状態を並行に中断・再開し，あたかも OS が管理するスレッドであるかのように振る舞うユーザレベルスレッド (User-Level Threads, ULT) の技術である．

また，分散メモリシステムで動作するランタイムに欠かせないのが，低水準な通信ハードウェアを制御するランタイムシステムである．本稿ではこれを低水準通信ライブラリ (Low-Level Communication Library) と呼んでいる．ハードウェアの通信性能を活かし切るには，ソフトウェアオーバーヘッド削減やマルチスレッド対応などの多くの課題が存在する．本研究では低水準通信システムも開発し，設計と実装について述べる．

本研究では，DSM と ULT という 2 つの実装技術を前提とし，生産的かつ高性能なランタイムシステムの設計について検討し，主に低水準通信やデータ再配置といった要素技術を実装・評価した結果について報告する．

目次

第 1 章

序論

1.1 背景

1.1.1 分散メモリシステムのプログラミングモデルに関する研究背景

現代の科学技術の発展には莫大な量の計算からなるシミュレーションが不可欠であり，それを実現する基盤となっているのがスーパーコンピュータとも呼ばれる大規模並列分散計算機である．大規模並列分散計算機上のシステムやアプリケーションに関する学問分野は **高性能計算 (High Performance Computing, HPC)** と呼ばれ，その研究成果は科学技術の持続的な発展を支え続けている．

並列分散計算機のハードウェアは，数十年間一貫して性能向上を続けてきた．逐次プロセッサの周波数上昇による CPU 性能向上は期待できなくなったが，その後もマルチコア化・メニーコア化といった手法で，1960 年代から現在に至るまでムーアの法則と呼ばれる指数的性能向上を続けてきた．並列分散計算機は大量の計算ノードを接続した集合体で，それぞれはインターコネクトと呼ばれる高速なネットワークによって接続されており，そのネットワーク性能も飛躍的な向上を続けてきた．このようにハードウェアが高性能化・大規模化してきた一方で，ソフトウェアもそれを活用できるよう改良が続けられている．主にシステムソフトウェアに着目すると，MPI [1] による分散システムの通信規格の標準化，OpenMP [2] によるディレクティブによるループ並列化，コンパイラによる自動 SIMD 化技術，ワークスティーリングによるスケジューリングなどが実用化され，これら以外にも数多くのプログラミングシステムや言語といったソフトウェア要素が提案・開発されてきた．

並列計算機に対するそのようなハード・ソフト両面の改良が行われてきた一方で，数十年前から根本的には解決していない問題が，**分散メモリ (Distributed memory)** と呼ばれるハードウェアのメモリ形態と，それを前提としたプログラミングモデルの性能と生産性の問題である．分散メモリ型のメモリアーキテクチャとは，各計算ノードが独立したメモリを持ち，他ノード上へのメモリアクセスあるいは通信が自ノード上のメモリアクセスと比べて大幅に低速なものを指す．CPU・メモリ間の転送速度はアプリケーション性能に直結する（フォン・ノイマン・ボトルネック）ため，高性能化のためには CPU の近くにキャッシュメモリ (cache memory) を置くことが必要 [3] であり，その延長として CPU を増やしていくと考えれば自然に分散メモリ型メモリアーキテクチャに近づいていく．特にノード間インターコネクトは大規模なネットワークなので，ノード内の CPU・メモリ間の通信より低速になるのは技術的にやむを得ないという事情がある．そのようなハードウェア事情をほぼそのままプログラミングモデルとして扱えるようにしたのが，**メッセージパッシング (Message-passing)**（2.3.1 項）などの分散メモリ用の通信インターフェイスである．

分散メモリと対立した概念が，**共有メモリ (Shared memory)**（2.3.5 項）のモデルである．共有メモリとは，全ての計算ノードが単一のアドレス空間を読み書きできるというメモリモデルを意味する．ここで注意すべきなのは，共有メモリシステムだからといって単一のメモリしか保持することを許されないわけではない，ということである．実際，現在標準的に使用されているマルチコアプロセッサ（あるいは Chip Multiprocessor (CMP) とも）では，下位層のメモリ（L3 やメインメモリ）に関しては共有されているが，L1/L2 やレジスタといった上位層のキャッシュメモリに関しては分散していて，それらを同期するためのコヒーレンスプロトコル（2.5 節）によってあたかも共有されているかのように見せている，という実態がある．実際にはキャッシュメモリが分散しているにも関わらず，マルチコアプロセッサのために分散メモリ型のプログラミングモデルを導入せずに済んでいる理由は，ノード内のコア数が高々数

十程度なので，コヒーレンスプロトコルを十分に高速に実装することが容易だからである．

共有メモリモデルは，メッセージパッシングといった分散メモリ向けの各種プログラミングモデル（2.3 節）と比べて制約が緩いので，アプリケーション生産性は比較的高い．そのため，分散メモリ型アーキテクチャを保ちながら，共有メモリモデルを実現できれば性能と生産性を両立できるはずであるという発想が数十年前から存在し，そのような形態のシステムを **分散共有メモリ (Distributed Shared Memory, DSM)** [4] [5] と呼ぶ．

DSM は 1990 年代に盛んに研究された後，2000 年代に入ってからほとんど研究されなくなったという歴史的経緯がある．その背景には，コヒーレンスプロトコルのスケーラビリティ向上が困難であるということが次第に明確になってきたことが挙げられる．DSM が注目されなくなった後，メッセージパッシングはそのまま使われ続けていて，DSM の代替としては PGAS（2.3.4 項）というモデルが盛んに研究されるようになった．PGAS は共有メモリが持つ「グローバルアドレス性」を一部取り入れてはいるが，コヒーレンスプロトコルを実装せずにプログラマに手動でキャッシュを行わせるという決定的な違いがある．

PGAS の研究が活発であることから，共有メモリ “的” なプログラミングモデルが分散メモリシステムでも有用であるという考え方は多くの HPC 研究者の共通認識となった一方で，DSM のような真の共有メモリモデルについてはスケーラブルな実装が不可能である，と信じられている．確かに，そのような真の共有メモリシステムにおいて数百コア以上でスケールするような例は知られていない．しかし，少なくとも筆者が知る限りにおいて，共有メモリモデルがスケーラブルではないということを理論的に証明した研究は存在しない．そのような DSM 批判に対する懐疑論が，本研究の出発点となった．システム研究者の多くが共有メモリの特徴であるグローバルアドレスやコヒーレントキャッシュといった機能を有用だと考えているにも関わらず，DSM に関しては「過去に成功例がない」という理由だけで批判的であるという現状に対し，DSM を批判するのであればせめて正確な理解を得ようという試みである．

メモリのコンシステンシやコヒーレンスの問題解決を事実上放棄することは，DSM の代わりに PGAS の研究が主流になってから一般的になっているが，分散システムにおけるコンシステンシは本質的問題の一つであり，システムが担当しなければ必ずアプリケーションプログラマの負担となって手動でコヒーレンス管理を行うことが必要となる [6] ．小規模なマルチコアプロセッサではシステムが全面的にコヒーレントキャッシュを管理するのに対し，より並列度が上がると突然システムが全く支援をしなくなる，という不連続な生産性のギャップは，コヒーレンスプロトコルの実装の困難さという曖昧な理由だけで放置されたままになっている．

DSM のスケーラビリティには多くの課題があるのは事実であり，2 章ではそれらの各問題点について一つ一つ述べていく．しかし，それら全ては何らかの解決策が過去に考案されていて，いずれについても現実的に解決不能な問題であると示されたことはない（2.7 節）というのが筆者の現在の見解である．そのような考察から，本研究ではスケーラブルな DSM の開発は可能なのではないか，という仮説を立てている．仮に DSM がスケーラブルでない確証が得られたとしても，それ自体システム研究として有益な知見であり，その問題を回避するようなプログラミングモデルの修正につなげていくことができる．

プログラミングモデルそのものの問題点と，それを実現するための既知の実装手法の問題点を混同することは，既存研究でも頻繁に見受けられる．例えば，2.5.1 項で述べるページベース DSM では，ページフォールトのオーバーヘッドを避けられない問題があるが，それを根拠に「あらゆる DSM は低速である」と結論付けることはできない．ページベース DSM というのは実装の一方式でしかなく，コンパイラベース DSM といった他の手法も知られている．仮にハードウェアにアドレス検査専用の機構を加えられるとすれば，ソフトウェアオーバーヘッドが完全に消滅する可能性すらある．従って，この例では DSM のモデルそのものが低速である，と結論付ける根拠にはならない．

システム研究を行う立場としては，メッセージパッシングや PGAS などの DSM より抽象度の低いシステムには，現状で競合する既存研究が乱立していることも留意すべき点である．システム研究には長期の開発期間が必要であり，安定的な開発リソースがあるという保証がない中でそうした分野で競争するのは得策ではない．一方で，DSM は 1990 年代に盛んに研究された後，現在は研究分野として下火になっている．筆者は DSM が将来的に有望なシステム形態であると考えると同時に，競合相手が少ないために研究分野としてみても魅力的であるという視点を持っている．

システムが複雑であればあるほど，高並列環境でのスケーラブルなシステムの実現は困難になるので，DSM を実用的にすることは PGAS などと比べて困難である．一方で，実用面からいっても，あるシステムがその時点での大規模並列環境で完全にスケーラブルでなければならないというわけではない．実装のスケーラビリティを段階的に高めていく過程において，例えば数十〜数百ノードでスケーラブルであると示されれば，その時点でのアプリケーション

開発に寄与できる可能性はある．プログラマがアプリケーションを再開発する時間を加味すれば，特殊なプログラミングモデルに基づいて高速に動作する処理系よりも，実装自体は低速だがプログラミングモデルを変更しない処理系の方が，利用者の全体の労力が少なくて済む，という状況は十分に考えられる．

1.1.2　スレッドスケジューリングとメモリモデルに関する研究背景

並列性表現のモデルとしては，プログラムの一部を「スレッド」あるいは「タスク」に切り出すマルチスレッドプログラミングが一般的である．スレッドというのは必ずしも OS スレッドを意味せず，**ユーザレベルスレッド (User-Level Threads, ULT)** *であることもある．並列性よりも並行性を意識する場合は「コルーチン」という概念でも呼ぶことがある．

コルーチンには自動変数を保持するメモリ領域が必要であり，その管理はコルーチンの実装において主要な問題である（3.5 節）．これを単純に解決しているのがやはり共有メモリであり，中断 (suspend) されたコルーチンの自動変数を共有メモリ上に配置することで透過的にプロセス間でやり取りすることが，広く知られた共有メモリ上で動作するユーザレベルスレッド処理系の動作原理である．そして，共有メモリ上に置かれた自動変数は，ポインタとして保持したり逆に参照されたりといったことが当然のように可能である．本研究では，これと同様の実装手法が DSM でも可能であるということに着目している．自動変数あるいはコールスタックを特別視せず，単なる DSM 上のメモリ領域として扱うことで，スケジューラのソフトウェア階層をシンプルに保てる．そして，そのためにもやはり DSM のスケーラビリティが重要である．

2.4.4 項で示すように，スレッドスケジューリングとメモリコンシステンシモデルを結びつけることで，直感的なコンシステンシと高速なコヒーレンスプロトコルを実装できるという考え方は既に提案されている [7] [8] [9]．そのような実装手法が主流となることは起きていないが，透過的なシステム高速化手法として有望なものの一つであり，提案システムでの実装方式の基礎として取り入れている．本研究の特色は，そうしたスケジューリングの観点を DSM のコヒーレンスプロトコルに持ち込むとともに，長きに渡る DSM 研究の経緯も踏まえて，それら 2 つの概念を有機的に連携させるという方針にある．

並列計算機の性能問題はメモリモデルだけではなく，CPU にどのように仕事を割り振るかを決めるスケジューリングにもある．並列計算機のスケジューリング手法として一般的なものは，**ワークスティーリング (work-stealing)**（3.3 節）である．最適なスケジューリング手法はメモリの配置問題と不可分であるので，ここでもやはり共有メモリシステムを切り離して考えることはできない．現時点では，単純にワーカーを乱択するワークスティーリングについてしか理論的研究が進んでいない状況にあるが，定量的アプローチに基づいてメモリモデルを踏まえたスケジューリング手法が今後は求められると筆者は考えている．そして，共有メモリに細工をしてスケジューリングに役立てるといった手法についても，改造されたコヒーレンスプロトコルの実験環境が必要で，その研究のためにもソフトウェア DSM は扱いやすい基盤になる．

1.1.3　高性能かつ高生産な並列分散プログラミングシステムの研究の要件

C 言語が初めて開発されてから既に 45 年以上経過しているが，その間にハードウェアのアーキテクチャは劇的に変化し，計算性能は指数的に増大し続けてきた．そのようなハードウェアの変遷の中でも C 言語が大幅な変更なしにそのままプログラミングに応用され続けてきた理由は，そのメモリモデルと実行モデルが確固たるもので，新規のハードウェアが登場してもそれに対応できるだけの柔軟性を持っていたからである．逆に言えば，C 言語のセマンティクスをわずかでも破壊するプログラミングモデルに対して，システム開発者は極めて慎重でなければならない．新しいパラダイムのためのアプリケーションの再開発は，そのコストに見合うだけの強力な動機がなければ進まないからである．そういった互換性の観点からいっても，「共有メモリ」と「スレッド」という 2 つの並列プログラミングのプリミティブは優れた抽象化であり，将来の並列分散計算機の進化の過程でも存続し続けるであろうと筆者は予想する．

システムを現実的な時間で実現可能であるかという尺度は，実務的なソフトウェア開発の現場だけでなく，システ

* ユーザレベルスレッドとほぼ同一概念を指す用語には user-mode threads や，あるいは OS によるスレッドより軽量であることを強調した**軽量スレッド (lightweight threads)** というものもある．ユーザレベルスレッドのことを指して**タスク並列処理系 (task parallel systems)** と呼ぶこともあるが，「タスク」という用語が濫用傾向にある（例として 2.3.2 項の AM など）ため，本稿ではその使用を避けている．

ム研究の分野においても重要である．C++ に代表される幾つかの言語はゼロオーバーヘッド抽象化が可能であり，このことはシステム開発者が常に低級なシステムプログラミングを強いられることを回避し，高抽象度で記述されたシステムとアプリケーションを少なくとも C 言語と同程度に高速に動作させることができる．システムの性能というのはシステムの一面に過ぎず，システム開発とその保守には莫大なコストが必要であるという側面を軽視してはならない．

以上の論点を整理すると，高性能かつ高生産な並列分散プログラミングシステムの研究と開発にとって，筆者は次のような要素が不可欠であると考える．

- 既存のプログラミングモデルに対して，理論的根拠の弱い制約を一切課さないこと．
  - 具体的には，「共有メモリ」や「スレッド」といった基本的な並列プログラミングの枠組みをそのまま実現できること．ある時点での実装上の問題と，モデルそのものの問題を明確に切り分けること．
- スケーラブルな実装が可能で，かつ各種オーバーヘッドが現実的なレベルに小さいこと．
  - 仮にスケーラブルであるとしても，逐次プログラムより遅ければ並列化する意味がない．オーバーヘッド削減にはハードウェアに近いレベルの理解が不可欠で，モデルだけで議論していては達成できない．
- 現実的な時間で実装し，かつ長期的に保守可能なシステム規模であること．
  - 研究と開発に充てられる時間には限界があり，特にメモリシステムのような複雑なシステムでは開発時間が最も深刻な問題の一つである．ソフトウェア開発一般について言えることだが，コードの再利用を徹底すること，可能な限り抽象度の高いコードを維持すること，ユニットテストによって単純なバグの数を最小化することは HPC 研究の現場においても必要である．

1.2　研究目的

大規模並列計算機において生産的なアプリケーション開発を実現するために，理論的にスケーラブルなプログラミングモデルに基づき，かつ現実的な期間で実装可能なランタイムシステムを設計し，それを実装することを目指す．

DSM と ULT，そしてそれらの相互運用について研究することには，次のような意義がある．

- 共有メモリモデルに基づいた並列分散プログラミングの可能性と限界を明確にし，将来的な並列分散計算機向けシステムの設計に貢献する．
- データ局所性に配慮したスケジューリングといった未解決の問題に対して，通信システムやメモリシステムがどのような実装方式を提示できるかを示す．

1.3　本稿の構成

本稿の構成は次の通りである．

2 章 並列計算機の通信とメモリモデルに関する関連研究
: 並列計算機の代表的な通信モデルとメモリモデルを概観するとともに，特に本研究で重視する共有メモリとその実装方法について述べる．

3 章 並列プログラムの並列性記述とスケジューリングに関する関連研究
: 並列プログラムにおける並列性記述について，特に本研究で重視するユーザレベルスレッドとその関連技術について述べる．

4 章 高性能な低水準通信ライブラリの設計と実装
: 分散メモリ型アーキテクチャ上で高速なシステムを構築するには，インターコネクトのハードウェア・ソフトウェアのインターフェイスや性能特性についての理解が欠かせない．本研究では実際に低水準通信ライブラリを設計・実装し，その性能を評価した．

5 章 データ再配置とアドレスキャッシュのコンシステンシ

PGAS 処理系に対するデータ再配置機能の導入と，RDMA 活用のためのアドレスキャッシュという 2 つの機能を両立するためには，アドレスキャッシュのコンシステンシの問題が立ちはだかる．この章では，それを解決するためのコヒーレンスプロトコルについて実装・評価した結果を報告する．

6 章 ユーザレベルスレッドと協調する分散共有メモリ処理系の設計

前章までを踏まえた上で，各ソフトウェア階層のアイディアを統合した分散共有メモリ処理系として実現するための設計について述べる．この章で述べる処理系は開発段階で未だ実現には至っていないが，既に発見された問題についての解決策について紹介する．

7 章 結論

本研究の結論である．

第 2 章

並列計算機の通信とメモリモデルに関する関連研究

並列計算とは，同時刻に並列動作するプロセス*の集合であるといえる．各プロセスは完全に独立動作しているわけではなく，ある決められたデータの集合を協調して処理しているため，その協調動作のために必ず何らかの通信が必要である．並列プログラムの性能問題は，演算器やメモリといったハードウェア資源の不足やその利用効率の低下による場合と，メモリや他プロセッサとの通信待ち時間が原因の場合の 2 種類に大別できる．前者の問題はハードウェア資源を増強することで解決可能なため，後者の問題，すなわち通信性能にまつわる問題の方が遥かに深刻である．

プロセッサ間の通信でやり取りされるデータはメモリとして管理されているので，メモリモデルと通信プロトコルの間には密接な関係が存在する．例えば，RMA（2.3.3 項）に用いる領域は事前に通信処理系に “レジストレーション” する必要があるというメモリモデルになっているが，これは通信処理系が内部で用いる RDMA が物理アドレスを固定しなければならないという事情に基づくもので，通信のオーバーヘッドを減らすためにメモリモデルを改変した妥協の一つであるといえる．実用的なメモリモデルを定義するためには，それを実現する処理系が効率的に実装できるということを確認することが不可欠である．

2.1 並列計算機の通信性能モデル

並列計算機のモデルとして最も原始的なものは，**Parallel Random Access Machine (PRAM)** [10] である．PRAM モデルでは，全てのプロセッサが同期的に進行し，メモリアクセスのコストは全てのメモリ領域に対して均一である，という仮定を行う．実際の計算機では，メモリ階層が多段となっていて，L1 キャッシュメモリとノード外メモリではレイテンシが数千倍のオーダーで違うため，通信の観点からいって PRAM モデルは現実的な仮定ではない．しかし，モデルの単純さから現在でも使われることがある．

PRAM よりも現実のシステムに近いモデルとしてその後提案されたのが，**LogP モデル** [11] である．LogP モデルを改良して，スループットを考慮に入れたモデルが **LogGP モデル** [12] であり，本稿ではこれをベースに議論を行う．LogGP モデルにおいて定義されるのは，次のような指標である．

“L” = Latency （レイテンシ，遅延）
: 2 プロセッサ間で小さいメッセージをやり取りする時にかかる時間．
  - レイテンシを片道/往復のどちらで算出するかは文脈によるので注意が必要である．ここでは片道のレイテンシを使用する．

“o” = overhead （オーバーヘッド）
: プロセッサが通信を発行するためにかかる時間．
  - プロセッサが通信システムに通信開始を指示する際は，プロセッサ自身が何らかの処理を行って通信システムに委譲することが必要であり，o はその委譲のためにかかる時間を指す．

“g” = gap （ギャップ）

* ここでいうプロセスとは，実際の OS の “プロセス” そのものであることもあるが，一般には異なる概念である．

メッセージを大量に送信した時に，1 メッセージ当たりにかかる時間．

- g の逆数はメッセージレートと呼ばれる．

“G” = Gap per byte （バイト当たりギャップ）

十分大きいサイズのメッセージを送信した際の，バイトあたりにかかる時間．

- G の逆数であるバンド幅あるいはスループットの方がよく用いられる．

“P” = number of Processors （プロセッサ数）

計算機全体でのプロセッサの数．

オーバーヘッド o とギャップ g は，似ているが区別する必要がある．オーバーヘッド o は CPU が通信を実行するためにかかる時間であり，ギャップ g （あるいは逆数であるメッセージレート）はネットワーク全体でのメッセージ処理速度を意味している．仮に，メッセージの最大投入速度がネットワーク上の様々な機器（NIC，スイッチなど）によって律速されている場合は，$o < g$ となる．一方で，CPU よりもネットワーク機器の方がメッセージを高速に処理できる場合もある．例えば，InfiniBand においてオフローディングを行ってメッセージ集約を行った実験結果（4.7.2 項）では，オーバーヘッド $o = 0.35 \times 10^{-6}$[sec]，ギャップ $g = 1/(6.2 \times 10^{6}$[1/sec]) $= 0.16 \times 10^{-6}$[sec] であり，$o > g$ となっていることがわかる．

バイトあたりギャップ G，あるいはその逆数であるバンド幅 $B = 1/G$ を導入する理由は，長大なメッセージの送信時には専用の機構（例: Direct Memory Access (DMA) 転送）を用いて高速化することが一般的だからである．ある大量のデータを送るという場合に限っても，小さいメッセージを大量に送る場合は g，大きいメッセージを送る場合は G が律速要因となる．

LogGP モデルに基づくと，いくつかの典型的な指標を見積もることができる [13] .

- 片道のメッセージを送る際にかかる時間は $L + 2o$.
- 往復のメッセージを送って返ってくるまでにかかる時間は $2L + 4o$.

メッセージサイズ s を変数に取った時，片道の合計通信時間は次のように見積もることができる．

$$t(s) = L + 2o + Gs \tag{2.1}$$

Martin らの論文 [13] では LogGP の各指標に対するアプリケーション性能の依存性を実験的に計測している．その結果では，オーバーヘッドとギャップを増加させると性能が大幅に低下したのに対し，レイテンシとバイト当たりギャップ（バンド幅）の値はそれほど大きく影響しなかった，と報告している．レイテンシによる性能低下はノンブロッキング通信を使うなどのアプリケーションの工夫によって回避可能であるが，そのような工夫を全く行わなかったアプリケーションだけが性能低下したと指摘している．

2.2　通信ハードウェアのセマンティクス

通信について議論する際には，実際に使われるハードウェアの通信インターフェイスを踏まえた上で，それに沿った通信のセマンティクスを定義する必要がある．「ハードウェアに近いモデル」であると主張するには，実際の多くのハードウェアに搭載されているという根拠が必要である．

現在，多くのスーパーコンピュータで採用されているインターコネクトが InfiniBand [14] であり，その標準的なインターフェイスとして InfiniBand Verbs (IBV) [15] という API が用いられている．以降の説明では IBV をベースに述べる．IBV における主要なオペレーションは次の 4 つである．

SEND オペレーション

送信側が `ibv_post_send()`，受信側が `ibv_post_recv()` という関数をそれぞれ呼び出す．送受信それぞれのプロセスは自分のバッファのアドレスだけを与えるので，相手側のプロセスのアドレスについては知らなくてよい．

RDMA WRITE オペレーション

あるプロセスが別プロセス上のメモリに一方的にデータを書き込む．書き込まれた側のプロセスは自分のプロ

セスにデータが書き込まれたことを検知できない．

RDMA WRITE with Immediate オペレーション

RDMA WRITE とほぼ同様だが，書き込まれた側のプロセスが書き込みを検知できるという違いがある．

RDMA READ オペレーション

あるプロセスが別プロセス上のメモリから一方的にデータを読み取る．読み込まれた側のプロセスは RDMA WRITE 同様に読み込みを検知できない．

SEND オペレーション以外は `ibv_post_send()` という関数のみを用いる．この他にリモートアトミックの機能もあるが，本章と直接関係しないため省略する．

RDMA の送受信側双方のデータ領域は，`ibv_reg_mr()` という関数を呼び出して**メモリレジストレーション**する必要がある．RDMA を実行する場合は，送受信側双方で事前に自プロセスのメモリ領域をレジストレーションする．RDMA を実行するプロセスは，送受信側双方のメモリ領域のアドレスと，レジストレーションの際に取得できる ID（IBV の場合は `rkey` と `lkey`）をセットにして，送信要求関数（`ibv_post_send()`）に引数として渡す必要がある．

2.3　並列計算機の通信インターフェイス [16]

本節では，並列計算機の通信インターフェイスとして代表的なものを取り上げ，通信インターフェイスが通信ハードウェアの機能からどれだけ乖離しているか，という定性的な議論を試みる．通信インターフェイスだけではシステムの性能は決定できず，ハードウェア特性や実装手法も含めることでモデル化することで定量的な議論を行うことができる．

2.3.1　メッセージパッシング

メッセージパッシング (Message-passing) とは，「あるプロセスが別のプロセスを指定して送信を明示的に実行し，指定された側のプロセスが受信を明示的に実行すると，それらのプロセス間でデータが送受信される」というモデルである．Message Passing Interface (MPI) [1] は，その名の通りメッセージパッシングを基本とした通信インターフェイスであり，`MPI_Send` と `MPI_Recv` の対で表されるメッセージパッシング関数を提供する，MPI の用語では，メッセージパッシング関数群を “Point-to-Point 通信” と呼んでいる．

MPI におけるコミュニケータやデータ型といった概念を省略すると，次のような API として表せる．MPI の場合は，複数のメッセージを送受信する場合は “タグ (tag)” によって識別することができる．

```
void send(const void* buf, size_t size, int rank, int tag);
void recv(      void* buf, size_t size, int rank, int tag);
```

メッセージパッシングの特徴は次のように挙げられる．

1. 送受信双方のプロセスが明示的に通信を実行している．（逆に，片方のプロセスだけでは通信が進行しない．）
   - 受信側は，「送信側が何かデータを送った」ということから，データそのものだけでなく，送信側プロセスと同期も取れているということを意味する．後述する PGAS のような例では，誰かが読み書きを行ったかどうかは，明示的に同期するか変数をポーリングするかしない限り分からない．
   - 3. のアドレスの秘匿性とも関連している．送信側からみて，最終的に受信側のどこにデータが送信されるか実は分からないが，「受信側のアドレスは知る必要がない」というセマンティクスからいつでもデータは送信できるということになる．
2. 送受信双方のプロセスが，お互いのプロセス ID（とバッファサイズ）を明示的に指定している．
   - 後述する PGAS や共有メモリのように，「書き込んだ値が次に誰に読まれるか分からない」というモデルと比較すると，お互いのプロセス ID を明示することで「誰とでも通信できる」可能性を排除していることが分かる．
3. 送受信双方のプロセスは，お互いの受信先・送信元のアドレスについて知らない．
   - アドレスが分からないという点は，後述する RMA とは対照的である．

メッセージパッシングはしばしばブロッキング通信のみを前提として議論されることがあるが，その場合は送受信の順番によってデッドロックの問題が起きうる．この問題はノンブロッキングなメッセージパッシングでは起きないため，メッセージパッシングの問題というよりブロッキング通信の問題であるといった方が正確である．MPI でのデッドロックには，`MPI_Sendrecv` を使うなどいくつか回避策が知られている．

MPI において広く用いられていることから，メッセージパッシングは通信ハードウェアのモデルに近いと一般に捉えられているが，必ずしもハードウェアに直接マッピングされるわけではない．例えば，IBV にはメッセージパッシングに相当する `SEND` オペレーションが存在するが，このオペレーションは必ずしも MPI の Point-to-Point 通信と一致しない．主要な要因は次の通りである．

1. 受信側のバッファは有限長しかない．
   - この事実は，インターフェイスとして任意サイズのメッセージを送信可能であると定義した際に問題になる．データサイズが小さい場合は Eager プロトコルと呼ばれる send/recv を使う方法でよいが，データサイズが大きすぎる（**Eager limit** と呼ばれる値を超えている）場合には，MPI ランタイムが内部で Rendezvous プロトコルに切り替えることで対応している．Rendezvous プロトコルでは，ゼロコピーを活用するため RDMA としてメッセージパッシングが実装される．
   - MPI には送信モード (send modes) があり，これによってバッファリングやプロセス間同期をある程度管理できる．例えば，`MPI_Rsend` は受信側が必ず `MPI_Recv` を事前に発行していることを示すので，Eager プロトコルになると期待できるが，実際にそのように実装するかは処理系の自由である．
   - LogGP モデルに基づいて Rendezvous プロトコルに切り替える意味を考えると，Eager プロトコルで通信速度を律速するギャップ g ではなく，バイト当たりギャップ G によって通信が律速されるように切り替えているといえる．
   - Eager limit を通信性能モデルに組み込んだ LogGPS モデル [17] [18] や，その派生である LogGOPS モデル [19] も提案されている．
2. `MPI_ANY_SOURCE` や `MPI_ANY_TAG` に相当する機能がない．
   - `MPI_ANY_SOURCE` は，全てのプロセス (MPI の用語でランク) からの送信を受け付けるという意味だが，例えば IBV ではそのままでは記述できない．`MPI_ANY_SOURCE` や `MPI_ANY_TAG` の性能については，MPI の標準化委員会でも議論されている [20] .

要約すると，メッセージパッシングは「受信バッファサイズ以下の大きさのデータ」を「決められた 2 プロセス間で send/recv しあう」という条件付きで初めて，ハードウェアのモデルに近い通信が行える，ということができる．

2.3.2　Active Messages (AM)

Active Messages (AM) [21] とは，「あるプロセスが別プロセスを指定してメッセージを送ると，そのメッセージに基づいて送信先プロセス上で処理が実行される」というモデルである．Active Messages の特徴は次のように挙げられる．

1. 受信先のプロセスは通信の確立に介在しないが，メッセージを受け取り次第，CPU を一時的にメッセージ処理に充てる．
   - 受信先のプロセスは送信元から呼び出される形になるので，送信元のプロセス ID やアドレスを事前に与える必要はない．
2. 送信元のプロセスはリモートのバッファアドレスを知らない．
   - GASNet [22] における AM のインターフェイスはリモートのアドレスを要求せず，その他の多くの実装も同様である．そのようなインターフェイスは，「システムによるバッファ管理が自動的に行われる」ことを暗に仮定している．
   - UCX [23] においては，「ゼロコピーの AM」のインターフェイスが存在し，その場合はローカルバッファのレジストレーションについて強制されるが，リモートバッファについてはやはり言及はない．

AM は，IBV における RDMA WRITE with Immediate オペレーションに相当するが，メッセージパッシングの場合と同様に必ずしも AM と一対一しない．一度に送信する AM のメッセージサイズが一定以上になると，受信側バッファを超過してしまうので，任意サイズの AM を実現するには MPI の Rendezvous プロトコルのような機構が同様に必要になる．AM を実装している GASNet [22] における 3 種類のメッセージタイプ (short, medium, long) も，MPI の場合と同様の問題解決のために導入されている．

システム開発において，受信先の CPU の介在が必要である一方で受信先のプログラムに明示的な通信確立を記述したくないという場合は存在し，そのような場合に AM は有用である．後に述べる RMA によって記述できない処理は多く存在するため，それらの代替として AM は使われる．

AM のことを指して「タスクベース」モデルであると表現することもある [24] [23]．リモートプロセスにタスクを押し付けているという考え方自体は誤りではないが，AM はスレッドよりも遥かに制約が厳しく，その主たる用途もシステム開発である点が異なっている．本稿で用いる「タスク」の用語も，AM を意味しているわけではない．

リモートプロセス上で関数を呼び出す仕組みは，一般に **Remote Procedure Call (RPC)** と呼ばれる．RPC と AM はモデルとして似ているが，RPC は両方向でメッセージングを指し，AM は片方向のメッセージングであるという違いがある [3]．IBV に関してみると，AM の方がよりハードウェアに近いモデルであるということができるが，返答が不要な状況というのは限定的であるため，RPC の方がより利便性が高い．

AM の実装例としては，GASNet [22]，UCX [23]，AM++ [25]，PAMI [26] などがある．

2.3.3　Remote Memory Access (RMA)

Remote Memory Access (RMA) とは，「(プロセス ID, アドレス) の組によって他プロセス上の特定のメモリ領域を表し，その領域を一方的に読み書きできる」というモデルである．MPI の用語では，**片方向通信 (One-sided communication)** とほぼ同義になっている．RMA の特徴は次のように挙げられる．

1. リモート側のプロセスの CPU は，明示的に通信に関与しない．
   - 逆に言えば，リモート側のプロセスと処理を同期させることが基本的にできない．
   - メモリモデルとして陽に現れないだけで，RMA であってもリモート側プロセスにおいてシステムが内部で CPU を使うことはあり得る [27]．RMA を完全に RDMA で実装したときのみ，真にリモート CPU を使用しないということができる．
2. 通信の前にリモートバッファは必ずメモリレジストレーションされていなければならない．
   - 他プロセスのメモリ領域を読み書きすることができるといっても，そのプロセス上の全てのメモリ領域に自由にアクセスできるわけではなく，事前に通信処理系にレジストレーションを行ってその際に取得した ID などを共有しておく必要がある．

RMA は通常 RDMA にそのままマップされるので，RDMA の抽象化であるとみなせる．IBV においては，RDMA WRITE と RDMA READ が RMA に相当するオペレーションである．

RMA の発展形として，**Notified Access** [28] というインターフェイスも提案されている．Notified Access はリモートプロセスに対して通知を送る機能を備えた RMA で，IBV の RDMA WRITE with Immediate に相当する．通知を送るという性質から，Notified Access と AM は似た概念であり，元となったインターフェイスがメッセージングか RMA かという点が主要な差異である．

RDMA をベースとした RMA の実装例としては，MPI-2 [1]，GASNet [22]，ARMCI [29] などがある．他にも，UCX [23]，libfabric [30]，OSPRI [31] など多くの研究例がある．

2.3.4　Partitioned Global Address Space (PGAS)

Partitioned Global Address Space (PGAS) とは，図 2.1 のように，「各プロセス上のメモリがグローバル領域とローカル領域に分割されており，グローバル領域に関してだけ全てのプロセッサが他プロセッサのメモリ領域に対して一方的に読み書きできる」というモデルである．グローバル領域アクセスのシンタックスは，Unified Parallel C

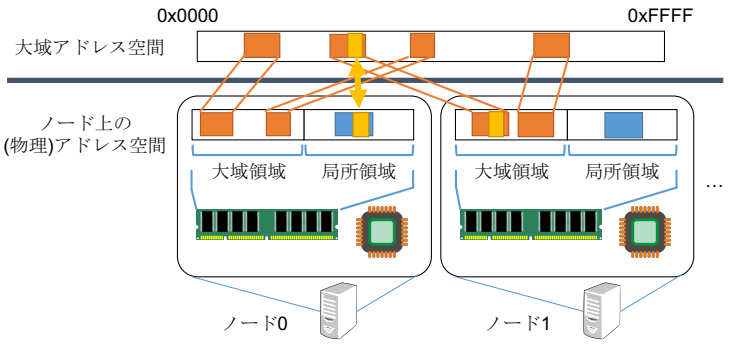


図 2.1: PGAS のメモリモデル

表 2.1: 代表的な PGAS 処理系の比較

システム名	グローバルビュー	ライブラリ	ベース言語
UPC [32]	Y	N	C
UPC++ [34]	Y	Y	C++
Global Arrays [35]	Y	Y	C
Grappa [36]	Y	Y	C++
Co-array Fortran [37]	N	N	Fortran
OpenSHMEM [38]	N	Y	C
X10 [39]	Y	N	Java
Chapel [40]	Y	N	独自

[32] のように通常の代入操作として記述される場合もあるが，明示的に **get/put** という関数によってアクセスできるようにするモデルの方が多い．get とはグローバル領域からローカル領域への転送，put とはその逆を意味する．明示的な get/put の有無というシンタックスの違いはあまり本質的ではなく，グローバル領域とローカル領域が明確に区別されているメモリモデルであるということが重要である．

PGAS のモデルをグローバルビュー PGAS とローカルビュー PGAS として 2 つに分類することもしばしば行われる．まず，ローカルビュー PGAS とは，「(プロセス ID, インデックス) の組によって，他のプロセスのグローバル領域にアクセスできる」というモデルである．それに対し，グローバルビュー PGAS は，「グローバルなポインタ」や「グローバルな配列とそのインデックス」といった形で，プロセスを明示的に指定することなくデータ転送ができる．

PGAS 処理系は RMA の上位層として実装することが一般的である．RMA の実行にあたっては，グローバルアドレスからリモートアドレスを事前に決定する必要があり，これは PGAS の主要なオーバーヘッドであるといえる．ローカルビュー PGAS におけるアドレス変換は自明なため，オーバーヘッドは事実上存在しない．グローバルビュー PGAS においてはこれは自明でなく，何らかの変換手法を実装する必要がある．5 章で述べる動的再配置機構は，このアドレス変換の高速化も重視して設計されている．

表 2.1 に，代表的な PGAS 処理系の比較を示す．生産性の観点からはローカルビューよりグローバルビューの方が望ましく，実用性の観点からは独自言語よりもライブラリのほうが相互運用性が高い．そのような観点からは，グローバルビューかつライブラリである UPC++ などが有用であるといえるものの，定量的な処理系間の比較は同一プログラムかつ同一環境で実行する必要があってそれ自体容易なことではなく，本稿の範疇を超えるので割愛する．

通常の PGAS においては，共有メモリにみられるようなキャッシュ機構を導入することはなく，そのために後述するコンシステンシモデルについて議論されることもほとんどない．コンシステンシの問題はデータ複製（2.5.2 項）を認めなければ原理的に発生せず，通常の PGAS 処理系はグローバル領域の複製を行わないからである．PGAS モデルの範囲内で複製を行うことは可能であり，例として UPC の relaxed モデル [33] が挙げられるが，そのようなモデルが注目されることは現在起きていない．

2.3.5　共有メモリ

共有メモリ (Shared memory) とは，「全プロセスに共通のアドレス空間があり，あるプロセスがそのメモリ領域のどこかに書き込むと，特定の条件下で別のプロセスがそれを読むことができる」という通信モデルである．「共有メモリが通信モデルである」という事実はしばしば無視されるが，プロセッサ間でデータをやり取りするためにごく自然に用いられており，紛れもなく通信手法の一種である．共有メモリの特徴は次のように挙げられる．

- **全てのメモリ領域は全てのプロセッサからアクセス可能である．**
  - 全てのプロセスは，プログラム実行中のある時点で存在する全てのメモリ領域にアクセスできる．
  - PGAS はグローバルとローカルの区別を行うという特徴があり，この点において共有メモリと決定的に異なっている．PGAS においてローカル領域は他のプロセッサから読み書きされることはないが，共有メモリにはそのような概念がなく，ユーザプログラムから可視なメモリ領域があればそれは必ず全プロセスから読み書きできる．
- **通信は暗黙のうちに行われる．**
  - PGAS も含めて，共有メモリ以外の多くのメモリモデルでは，システムがいつ実際の通信を発行するかをソースコード上で決定的に判断できる．共有メモリにおいては，原則的にメモリへの読み書きのみがインターフェイスであり，実際にいつ他のプロセッサとの通信が起きるかはシステムの実装次第である．
  - 後に述べるコンシステンシモデルの定義次第では，バリアの追加挿入などによって通信を事実上強制することができるため，その際には明示的に通信しているのとほとんど同様になる．当然ながらその場合はシステムによる Optimization の自由度が低下する．
- 通信にあたってレジストレーションのような操作を必要としない．

共有メモリ処理系の高速化にあたっては，通信を暗黙のうちに行えることを利用して，次の 2 つの機能を実装することが多い [4] .

- データの**移動，再配置 (relocation, migration)**
  - アドレス空間上のデータを実際に保持しているプロセッサや，そのプロセッサ内での実際のデータのアドレス，あるいはその両方を変更する．
- データの**複製 (replication)**
  - 複数のプロセスに同じデータを複製する．

共有メモリはあくまでモデルであり，どのように実装するかはシステム設計者に委ねられている．例えば，原理的にはキャッシュが一切存在しない共有メモリ処理系が存在しうる．一般的に，共有メモリ上で動作するプログラムをチューニングする際はキャッシュメモリの存在を前提とするが，厳密には共有メモリであるというだけでは前提が不足している．

あるプロセスがあるアドレスに書き込んだデータを他プロセスが読める，という挙動自体は一見自然であるが，キャッシュメモリが分散している状況においては自明なことではなく，システムが裏で通信を行うことで初めて成立している．プロセス間でのデータの “見え方” を強制するのが，**メモリコンシステンシモデル (memory consistency models)**（2.4 節）である．また，メモリコンシステンシモデルを満たすためのプロセッサ間通信の取り決めを**コヒーレンスプロトコル (coherence protocols)** と呼ぶ．

Distributed Shared Memory (DSM) [4] [5] とは，「物理的なメモリとしては分散しているが，共有メモリモデルを提供するシステム」のことを意味する．DSM はあくまで共有メモリシステムの実装の一形態を指す用語であり，プログラミングモデルとしては通常の共有メモリと等価である．また，この定義だけでは「何が DSM で，何が DSM ではないか」という区別は必ずしも明確ではない．例えば，現在のハードウェアはプロセッサごとに異なる L1/L2 キャッシュメモリを持ち，それらの状態を同期するためにコヒーレンスプロトコルに基づいて通信を行っているので，実際にはメモリが分散している共有メモリ処理系であるともいえる．そのような形態とは異なり，一般的に「DSM」といった場合に想定されるのは，「最下層のメモリ階層も含めて分散しており，その同期に使用する通信階層も共有メモ

表 2.2: 通信インターフェイスの比較

通信インターフェイス名	IBV 上での対応するオペレーション	通信発行側の CPU のみが介在	リモートバッファの明示的レジストレーション	ローカルバッファの明示的レジストレーション
メッセージパッシング	SEND	N	N	N(Y)
Active Messages	RDMA WRITE with Immediate	N	N	N(Y)
RMA	RDMA WRITE/READ	Y	Y	Y
Notified Access	RDMA WRITE with Immediate	N	Y	Y
PGAS	RDMA WRITE/READ	Y	Y	N
共有メモリ	なし	Y	N	N

リ処理系の一部として制御される処理系」であり，本稿の以降ではこの定義に従う．

共有メモリは，ハードウェアとしてもソフトウェアとしても実装可能である．DSM もハードウェア DSM とソフトウェア DSM があり，ハードウェア DSM の代表例は DASH [41]，ソフトウェア DSM には TreadMarks [42] などが挙げられる．DSM については主に 2.5 節で詳細に述べる．

2.3.6　通信インターフェイスの比較

表 2.2 に，各種通信インターフェイスの比較を示す．抽象度とアプリケーション生産性は共有メモリが最も高く，続いて PGAS，それ以外という順に低下していく．高抽象度のモデルの方がシステムによる Optimization の自由度は上がるが，システムの実装はより困難になる．高効率なシステムが実装可能であると示されなければ，プログラマは性能向上のために抽象度の低いインターフェイスを使うことを強いられることになる．

2.4　共有メモリのコンシステンシモデル

メモリコンシステンシモデルとは，共有メモリ処理系において各プロセッサ上でのメモリから読み出せる値をプログラムの各時点で強制するルールである．メモリコンシステンシモデルがあって初めて，プログラマはプロセッサ間でデータを確実にやり取りできる保証を得ることができる．一方，共有メモリシステムにとってのメモリコンシステンシモデルは，プログラムの正常動作のために守る必要がある義務を示している．逆に言えば，あるメモリコンシステンシモデルに従っているとして，そのルールに従ってさえいれば，各プロセッサ上のプログラムには別々の値が読み出せてもよい，という最適化の余地が存在する．

コンシステンシモデルにまつわる問題は，共有メモリシステムのみならず，分散ファイルシステムや分散データベースなどキャッシュ機構を内包するあらゆるシステムに共通しており，分散システムの普遍的問題の一つである．メモリコンシステンシモデルについての議論の多くは，そのような異なったソフトウェア階層のキャッシュ機構にも応用が可能である．

コンシステンシモデルの強制力は，厳しい (strict) ものと，弱い (weak) あるいは 緩和された (relaxed) [43] ものが存在する．strict なモデルは「メモリへの読み書きのみがインターフェイス」であり，relaxed なモデルは「メモリの読み書きに加えてフェンス命令を持つ」というのが主な違いである．

2.4.1　共有メモリの定義

本稿での共有メモリについての定義は，Batty らによる C++ のメモリモデルについての論文 [44] に則って行う．以前の C や C++ の言語標準にはマルチスレッド下のメモリモデルが一切含まれておらず，それらは全て処理系依存で曖昧な定義しかなかった．C++11 の標準化の過程で，まず Boehm ら [45] がメモリモデルを文章で定義した．その後，Batty らがそれらのセマンティクスを数学的に定式化し，さらにモデルの正当性について証明ツールで証明している．現在の C++ の言語標準に組み込まれているのは，この Batty らが証明したスレッディングモデルである．

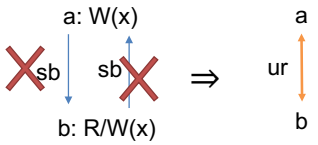


図 2.2: unsequenced race が起きる状況

まず，プログラムのセマンティクスを **Operational semantics**（操作的意味論）と **Existential witness** の 2 つに分けて考える．Operational semantics は言語とプログラムによって定義づけられるセマンティクスで，Existential witness は各実行ごとに異なった観測結果のことである．あるプログラムについて Operational semantics は一意であるが，Existential witness は実行毎に変化する．

命令 (instruction) は計算の最小単位で，メモリへの読み書きのどちらかまたは両方を含んでいる．命令の種類として，メモリからの読み出しを $\mathrm{R}(x)$，書き込みを $\mathrm{W}(x)$ のように記述することとする．ここで，x とはメモリ上のオブジェクトであり，具体的には「バイト」や「ワード」といった小さい単位である．また，Existential witness として読み出された値が x_1 のとき $\mathrm{R}(x)x_1$，書き込んだ値が x_1 のとき $\mathrm{W}(x)x_1$ というように記述する．

C++ の逐次プログラムにおいて，ステートメントの評価順序は制御構造の記述順通りであり，特定の演算子の評価順序も定められている．言語によって定められたこれらの評価順序のことを **sequenced-before**（略称 sb）と呼ぶ．sequenced-before は Operational semantics によって決定される．言語は大量の演算子や制御構造を含んでおり，それらの評価順序は必ずしも自明ではないが，一般に逐次処理の順序定義において曖昧性が発生することは少ない．

逐次プログラムにおいても言語によって評価順序が定められていない場合が存在し，場合によっては同一オブジェクトに対する複数の並行なメモリアクセスによる競合 (race) が起きる．逐次プログラムにおける競合を，**unsequenced race**（略称 ur）と呼ぶ．unsequenced race を含むプログラムは未定義動作 (undefined behavior) を引き起こす．以下に示す例では，==演算子の評価順序が決められていないことに起因して unsequenced race が起きる．

```
y = (x == (x=3));
```

unsequenced races の定義は，図 2.2 のように，同一オブジェクトに対する sequenced-before で順序付けられないアクセスのうち，1 つ以上が書き込みであることである．

$$\begin{aligned}\text{unsequenced\_races} = \{(a,\ b).\ &\\ &(a \neq b) \cap \text{same\_location}\ a\ b\ \cap\ (\text{is\_write}\ a\ \cup\ \text{is\_write}\ b) \cap\\ &\text{same\_thread}\ a\ b \cap\\ &\neg(a \xrightarrow{\text{sequenced-before}} b\ \cup\ b \xrightarrow{\text{sequenced-before}} a)\}\end{aligned} \tag{2.2}$$

シングルスレッドプログラムにおけるメモリの状態はプログラムの各実行ステップごとに一つしかなく，メモリの状態は大域的なクロックによって決まるとしてよいので，読み出し $R(x)$ は sequenced-before に基づいて「その時点での最新の値」を読む，と定義すればよい．マルチスレッドプログラムにおいても，仮に全てのメモリアクセスが後述する Sequential Consistency に基づくとすれば同様であるが，そのような仮定は性能上非効率であり，C++ の言語標準にも含まれていない．大域的なクロックではない形でメモリの読み出しを定義するためには，**happens-before 半順序**（略称 hb）[†] を導入する．happens-before 半順序は，sequenced-before を含んでいて，さらにマルチスレッドプログラムにおけるスレッド間の依存関係を含んだ **synchronizes-with** を含んだ関係である[‡]．

$$\xrightarrow{\text{happens-before}} \equiv \xrightarrow{\text{sequenced-before}} \cup \xrightarrow{\text{synchronizes-with}} \tag{2.3}$$

一般的なコンシステンシモデルでは，逐次プログラムはそのまま動作するように定義するので，sequenced-before で定義されるセマンティクスに加えて，マルチスレッドプログラムとして動作するための synchronizes-with 関係を追加で定義する．この synchronizes-with 関係の定義の方法が各コンシステンシモデルの違いとなっている．

[†] 半順序 (partial order) は反射律・推移律・反対称律が成り立つ関係 (relation) のことであるが，happens-before などの議論には反対称律は必要でなく，反射律・推移律のみからなる Preorder であるとしてもよい．

[‡] Batty らの元論文では，C++11 で導入されたメモリオーダーの一種である memory_order_consume を定義するために，より複雑な happens-before の定義を使用している．

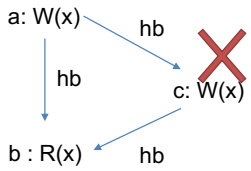


図 2.3: visible side effect の定義

読み出される値を書き込んだ命令のことを **visible side effect** と呼び，読み書きの命令間の関係を visible-side-effect 関係として以下のように定義する[§]．

$$\begin{aligned} & a \xrightarrow{\text{visible-side-effect}} b \equiv \\ & \quad a = W(x) \cap b = R(x) \cap a \xrightarrow{\text{happens-before}} b \cap \\ & \quad \neg(\exists c.\ c = W(x) \cap c \neq a \cap a \xrightarrow{\text{happens-before}} c \xrightarrow{\text{happens-before}} b) \end{aligned} \tag{2.4}$$

この定義の意味は，図 2.3 のようになる．書き込み a と読み出し b の間に happens-before で関係づけられた別の書き込み c が存在していた場合，b は c の書いた値を読むことになるので，その場合は a は b の visible side effect ではない，ということである．

ある読み出し b に対して，その visible side effect である書き込み a は複数存在しうる．実際に b が読み出した値が a の書いた値である時，**reads-from** の関係があると定義する．visible side effect は reads from の必要条件である．

$$a \xrightarrow{\text{reads-from}} b \Rightarrow a \xrightarrow{\text{visible-side-effect}} b \tag{2.5}$$

実際に読み出す値は実行毎に異なるため，reads from は Existential witness の一部である．

データレース (data race) は異なるスレッド同士においてのみ定義される．data race の定義は unsequenced race の定義とほとんど同じであるが，sequenced-before の代わりに happens-before を用いることと，異なるスレッドであるという違いがある．

$$\begin{aligned} \text{data\_races} = \{(a, b).\ & \\ & (a \neq b) \cap \text{same\_location}\ a\ b \cap (\text{is\_write}\ a \cup \text{is\_write}\ b) \cap \\ & \neg\text{same\_thread}\ a\ b \cap \\ & \neg(a \xrightarrow{\text{happens-before}} b \cup b \xrightarrow{\text{happens-before}} a)\} \end{aligned} \tag{2.6}$$

2.4.2　Sequential Consistency

逐次一貫性 (Sequential Consistency) [46] は，並列計算機の実用的なメモリコンシステンシモデルのうち，最も厳しい (strict) ものである．メモリシステムが以下の 2 つの性質を満たす時，そのシステムは sequential consistent であると定義される．

1. 全てのプロセッサによる全てのメモリアクセスが，ある逐次的な順序によって実行されたかのようにみえる．
2. 各プロセッサによる命令列の実行順序は，プログラムによって指定された順序と同じである．

これらの条件は，次のような条件と等価であることが示されている [43]．これを図示したのが図 2.4 である．

1. 全てのプロセッサによる全てのメモリアクセスは，ある全順序によって実行される（大域的なクロックの保証）．
2. 各プロセッサにおける全てのメモリアクセスは，プログラムで指定された順番で実行される．

Sequential Consistency を満たすとすると，あらゆるメモリ領域に対して書き込んだ後それを読み出すと synchronizes-with の辺が追加されるということになる．

$$a \xrightarrow{\text{reads-from}} b \Rightarrow a \xrightarrow{\text{synchronizes-with}} b \tag{2.7}$$

[§] この例ではアトミック命令のような読み書き両方に含まれる命令 (Read-modify-write) を無視している．

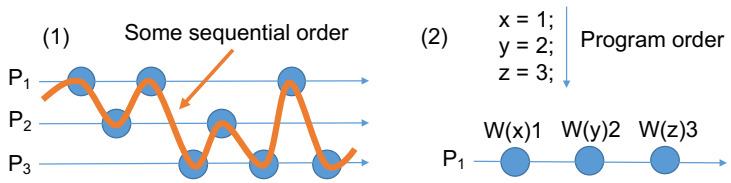


図 2.4: Sequential Consistency の定義

各メモリアクセスが全て happens-before 関係を作り出すので，システムにとっては厳しい制約となっている．

プログラマから見た Sequential Consistency は，ある大域的な全順序に従ってメモリアクセスが実行されていくため，メモリ全体の状態が逐次的に変更されていくと考えればよく，直感的なモデルであるといえる．一方で，メモリアクセスの全順序を保証するには，全ての書き込みが全プロセッサで同じ順序で実行されるようにする必要があり，必然的に書き込み順序の集中管理が必要となる．また，プログラムで指定された順序を守るために，メモリリーダリングやアウトオブオーダー実行といった各種の高速化技法も制限されることになる．そのため，Sequential Consistency を満たすことは，並列プログラムのスケーラビリティを損ねる原因となる．

ハードウェアの共有メモリシステムにおいては，Sequential Consistency よりも僅かに緩いコンシステンシモデル採用されることが一般的である．例えば，x86 のマルチコアプロセッサでは Total Store Order (TSO) [47] というメモリモデルが採用されている．そのようなコンシステンシモデルは Sequential Consistency を厳密には守らず，いくつかパターンでのリオーダリングを許可するが，依然として厳しいコンシステンシモデルであることに変わりはない．

2.4.3　Release Consistency

Release Consistency [48] は緩和型コンシステンシの一種で，排他ロックにコンシステンシを紐付けたものである．Release Consistency では，次のようなメモリ操作があることを前提としている．

- メモリの読み書き: R(x)a, W(x)a
- acq(L): L のロックを取得する (acquire)
- rel(L): L のロックを解放する (release)

一般的なミューテックスと同様，ある時点でロックを取得できるノードは 1 つだけという制約が課される．そのため，ロックを管理するノードを別に用意する必要があり，これは関連する論文において**マネージャ**と呼ばれている．コヒーレンスプロトコルの設計において，このマネージャに付加的な情報を持たせることがしばしば行われる．

次の条件が満たされるとき，そのメモリシステムは release consistent であると定義される [49] .

1. あるプロセッサにおいて実行した read か write が他のプロセッサでも実行された時，全ての先行する acquire もその（他の）プロセッサ上で実行されていなければならない．
2. あるプロセッサにおいて実行した release が他のプロセッサでも実行された時，全ての先行する read/write もその（他の）プロセッサ上で実行されていなければならない．
3. acquire と release が sequential consistent である．

1,2 の条件は，クリティカルセクションの概念から自然に導かれる．あるプロセッサ P_1 が acquire を実行してクリティカルセクションを始めようとした際に，その後に実行する予定の read/write を acquire より先に他のプロセッサ P_2 上で実行してしまうことを考えると，P_2 からは P_1 がクリティカルセクションを開始する前にメモリアクセスを開始したかのように見えることになる．この状況は排他制御が必要なデータに排他制御なしでアクセスしているのと同様であり不適切だが，1 の条件からこのような状況は回避される．また，あるプロセッサ P_1 が release を実行してクリティカルセクションを終える際に，その前に実行していたはずの read/write が他のプロセッサ P_2 上では実行されていないとすると，クリティカルセクション内で実行したメモリアクセスが実際には外側で実行されていることになり，やはり不適切である．このような状況は，2 の条件から回避される．3 の条件も，通常想定されるクリティカルセクションの考え方とやはり同様である．

Release Consistency の概念は，Batty らによる定義にもミューテックスという形で含まれている．式中の $a \xrightarrow{\text{sc}} b$ は，ロックの取得順が sequential consistent であることを意味する．

$$a \xrightarrow{\text{synchronizes-with}} b \equiv (a = \text{rel}(L) \cap b = \text{acq}(L) \cap a \xrightarrow{\text{sc}} b) \cup \cdots \tag{2.8}$$

2.4.4　DAG Consistency

DAG Consistency [7] [8] [9] は緩和型コンシステンシモデルの一種で，計算 DAG (Computational DAG) 上でのスレッド間の依存性に基づいて定義される．具体的には，スレッドを新しく作る (fork) 場合や，スレッドを待ち合わせる (join) 場合といった状況において，happens-before の辺を追加する．Batty ら [44] は，**additional-synchronizes-with** という種類の辺としてそうした依存関係を追加している．スレッドの fork/join といった操作はプログラムによって決定的なので，Operational semantics に含まれる．

$$a \xrightarrow{\text{synchronizes-with}} b \equiv a \xrightarrow{\text{addititional-synchronizes-with}} b \cup \cdots \tag{2.9}$$

Blumofe らが定義した DAG Consistency [8] について以下に述べる．まず，計算 DAG とは，頂点の集合 V が命令の集合，辺の集合 E が命令間の依存関係の集合であるような有向非循環グラフ $G = (V, E)$ である．G 上で頂点 i から頂点 j に長さ 0 以上の経路がある時，$i \prec j$ と表す．

計算 DAG は，happens-before 半順序によって表される DAG と同値である．このことは，命令間の依存関係 E が Operational semantics と Existential witness の両方によって決定されることを意味する．

DAG Consistency を満たすということは，共有メモリ上のあらゆるオブジェクト x について，次のような条件を満たすある写像 $f_x : V \mapsto V$ が存在することである [8] .

1. $\forall i \in V$ について $f_x(i) = W(x)$
2. $i = W(x) \Rightarrow f_x(i) = i$
3. $j = R(x)a \Rightarrow f_x(j) = \exists i \cap i = W(x)a$
4. $\forall i \in V$ について $i \not\prec f_x(i)$
5. $\forall (i, j, k).\ i \prec j \prec k$ について $f_x(j) \neq i \Rightarrow f_x(k) \neq i$

写像 f_x とは，ある命令 i が x を読んだ時に読み込まれる値を書き込んだ命令を関連付けている．Blumofe らはこの写像が “タグ付け” のようなものであるとして説明しており，この説明に則って各条件について解説する．

1. $f_x(i)$ が必ず何かの書き込みの命令によるタグ付けであることを意味する．
2. 自身が x に対する書き込みであるような命令は，そのタグを更新するという意味である．
3. 命令 i が読みの際には，$f_x(i)$ でタグ付けられた書き込みが書いた値を読むということである．
4. 「未来の書き込みの値は読めない」という意味である．
   - 読み i と命令 j の関係が $i \not\prec j$ であるということは，(1) $i = j$ か (2) $j \prec i$ か (3) 順序付けられない (incomparable) のいずれかであるということである．(1) は i が書き込みの時，(2) は過去の値を読む時，(3) は計算 DAG によって順序付けられていない値を読む場合 (data race) である．
5. 対偶を取ると，$f_x(k) = i \Rightarrow f_x(j) = i$ となる．k が i 以前にタグ付けられた値を読んだとすると，その間にある j も同じタグとなる．つまり，計算 DAG 上の順序付けによってタグが強制されている．
   - 4. で既に過去の値を読むことは強制されているが，5. は最新の値を読むことを強制する．例えば，$i(= W(x)a) \prec j(= W(x)b) \prec k(= R(x)c)$ のように順序付けられている時，k が i でタグ付けられた値を読むとすると，5. から $f_x(j) = i$ であるが，2. から $f_x(j) = j$ であるにも関わらず $i \neq j$ であるからこれは矛盾である．つまり，上のような場合 $c = b$ となることが保証される．
   - 共有メモリ一般として述べた表現は異なっているが，5. の条件は本質的に visible side effect の概念と等価である．

additional-synchronizes-with 関係が追加されるのは，スレッドの依存関係がある時のみであり，それら以外には

happens-before の関係が追加されないので，プロセッサ間でデータを同期する必要がなくなる．後に述べるワークスティーリングスケジューラの実装上は，リモートプロセッサ上のスレッドと依存関係があるときだけコヒーレンス操作を発動させるだけでよいという特長を持つ（6.2.2 項）．

DAG Consistency とはスレッドの生成・破棄を組み込んだコンシステンシモデルであり，C++ 言語標準で定義されるコンシステンシモデルの一部であるとみなせる．すなわち，DAG Consistency に基づいてコヒーレンスプロトコルの Optimization を行うことは，C++ の言語上許可されるということである．スレッドに基づいたコンシステンシモデルは，プログラマにとっても直感的であり，C++ 言語標準のメモリモデルにも組み込まれている妥当なモデルである．

DAG Consistency は主に Blumofe らによって研究されたが，Peng ら [50] [51] は DAG Consistency に Release Consistency を組み合わせたコンシステンシモデルについて厳密化を行っている．Release Consistency と DAG Consisistency のいずれも，現在では C++ のメモリモデルに含まれている[¶]．

2.5　共有メモリ処理系のコヒーレンスプロトコル

本節では，共有メモリ処理系のコヒーレンスプロトコルについて，特に DSM 処理系の例を中心に取り上げる．共有メモリというモデル自体は同一であっても，数十コア程度にスケールすればよいマルチコアプロセッサと，数千ノードでのスケールが要求される DSM 処理系ではコヒーレンスプロトコルの設計方針は異なっている．

2.5.1　コヒーレンス粒度とアクセス制御

共有メモリ上のあるオブジェクト（2.4.1 項）に対し複数のプロセスが読み書きを行う場合，コヒーレンスプロトコルによってオブジェクトの値は適切なタイミングで同期される．そのためにはオブジェクトがいつ読み書きされたかを検出する必要があり，プロセッサによって適切な値が読まれるようアクセス制御を行う必要もある．そのアクセス制御の単位のことを，一般に **コヒーレンス粒度 (coherence granularity)** と呼ぶ．コヒーレンス粒度は “オブジェクト” の大きさと一致する必要はなく，アクセス制御すべきオブジェクトを適切に管理できさえすればよい．アクセス制御を行う主な手法には，ページベース，コンパイラベース，オブジェクトベースの 3 通りがあり，それらについて述べていく．

まず，**コンパイラベース DSM (Compiler-based DSM)** とは，特殊なコンパイラによってプログラム中にコヒーレンス操作を埋め込む手法である．研究例として，Cheong ら [52] によるものなどがある．コンパイラベース DSM は自動並列化 (Automatic parallelization) に依存しているが，現実的なプログラムでコンパイラによる自動並列化は困難であるために，コンパイラベース DSM はあまり注目されなかった．

オブジェクトベース DSM (Object-based DSM) とは，プログラム上での意味のあるまとまりとして “オブジェクト” ごとにコヒーレンスを管理する手法である．ここで述べている “オブジェクト” とは，先ほどまで述べてきた「ワード」などにマップされるオブジェクトとは異なる点に注意する必要がある．オブジェクトベース DSM の例としては，Entry Consistency を用いた Midway [53] が挙げられる．オブジェクトベース DSM は追加のアノテーションを要求するため，言語の改変が必要となるという問題がある．

ページベース DSM (Page-based DSM) とは，MMU のメモリ保護機能を流用してキャッシュの存在判定に利用する手法である．ページベース DSM として初めて実装されたのは Ivy [54] である．ページベース DSM は OS の仮想メモリ機構を利用することから **Software Virtual Memory (SVM)** とも呼ばれる [5] [55] が，本稿では全てページベース DSM で統一する．

ベージベース DSM では，DSM における “メモリブロック” と，OS における “ページ” が一対一に対応している．ローカルにページがキャッシュされている状態ならば，そのノード上の仮想ページも実際に物理ページを割り当てられている．逆に，ページがローカルにキャッシュされていない状態では，物理ページが割り当てられていないので OS がページミスを起こす．この際に，ベージベース DSM はシグナルハンドラを利用してページミスを検知し，実際にデータを持ったリモートノードと通信して自ノードにページをキャッシュする．

ページベース DSM の利点を挙げる [5] .

[¶] より完全なメモリモデルとしてはアトミック命令などを組み込む必要がある．

1. キャッシュ存在時のソフトウェアオーバーヘッドが無い．
   - 他手法ではこのオーバーヘッドが避けられないため，最大の利点であるといえる．
2. コンパイラや言語を変更することなく，C 言語のライブラリとして実装可能である．
   - ライブラリであるということは，既存の大半のプログラムを再コンパイルなしに利用可能であることを意味する．

一方で，ページベース DSM には以下のような欠点もある．

1. ページサイズ未満の大きさでページの読み書きを検出できない．
   - これが最大の問題であるといえる．
2. ページフォールトハンドラを介するオーバーヘッドが不可避である．
3. ページ保護機構にアクセスするシステムコールによるオーバーヘッドが不可避である．
   - DSM 上でのキャッシュミスを起こすことになるので，リモートアクセスによるレイテンシと相対的に比較すべき問題である．

性能上のトレードオフとしては，キャッシュミス時はコンパイラベース，キャッシュヒット時はページベースの方が高速である．DSM を設計する上では，プリフェッチなどを組み合わせてキャッシュヒットが多くなることを目指すので，ページベースの手法のほうが妥当であるといえる．しかし，シグナルハンドラやシステムコールによるオーバーヘッドが余りにも大きい場合は，そのような考察が当てはまらない．本稿では行っていないが，定量的な評価が求められる点であるといえる．

ページベース DSM はコンパイラに全く手を加える必要が無いので，開発が比較的容易であることも重要な利点であり，以降では開発コストが現実的であるという面からページベース DSM を前提として議論を行う．コンパイラベースの手法も，現在では LLVM [56] のような比較的取り組みやすい環境が整備されているが，依然としてページベースより開発コストは大きい．

ページベース DSM は，コヒーレンス粒度が OS のページサイズ (e.g. 4KiB) に固定されており，それにまつわるオーバーヘッドが大きいと考えられている [57]．これに対しては次のように反論することができる．

- 「ページサイズより遥かに小さいデータ (数バイト)」を扱う場合，ページベース DSM では *diff*（2.5.2 項）を用いてデータを管理することが一般的である．diff によって生じるソフトウェアオーバーヘッドが大きいことは事実であるが，P/S 分類（2.6.3 項）のような手法を使うことで，ほとんど diff を使わずにコヒーレンスを実装することは可能である．ハードウェアの共有メモリではページサイズより小さいキャッシュラインサイズ (e.g. 64KiB) でコヒーレンスプロトコルが実装されているが，ソフトウェアによって同様の粒度でコヒーレンスを保つ効率的手法は現在のところ知られておらず，ページベースで管理する手法が現状の最善策である．
- 「ページサイズより大きいデータ (数 MiB〜)」を扱う場合，データアクセス毎にページサイズ分だけ通信するとすると，通信システムのバンド幅を使い切ることができない場合がある．例えば，Tofu インターコネクトで最大バンド幅で通信するには，最低 16KiB 程度のメッセージサイズ（4.7.1 項）が必要である．一方で，ページサイズが 4KiB であることは，4KiB 未満でページのアクセス保護を変更できないというだけであり，通信単位は 4KiB 以上であれば実装上の問題は起きない．プログラムがデータ局所性を活かして記述されていれば，周辺のページもアクセスされる可能性が高いと見積もれるので，連続した複数ページをまとめて通信すればバンド幅を使い切ること自体は困難ではない．しかし，当然ながら，プログラムがデータ局所性を活かせていない場合，例えばランダムアクセスを繰り返す場合は，ページサイズより大きい単位でデータを転送しても性能は向上しない．

2.5.2　共有メモリシステム上でのキャッシュ複製数の制限

2.3.5 項で述べたように，共有メモリシステムの特徴としてメモリブロックの再配置と複製を行うことが挙げられる．実際にどの程度複製を許可するかで，コヒーレンスプロトコルを大きく 3 種類に分類することができる [4] .

Single Reader / Single Writer (SRSW)
: いかなる状態でもデータを一切複製しない．複製を認めない場合でも，再配置は可能である．

Multiple Reader / Single Writer (MRSW)
: あるキャッシュブロックを同時に複数のプロセスが読めるが，書き込み中プロセスは 1 つだけしか許可しない．

Multiple Reader / Multiple Writer (MRMW)
: あるキャッシュブロックに対して，読み込みも書き込みも複数のプロセスに許可する．

コヒーレンス確保のためにシステム側で共有プロセス数（2.6 節）を制限することは，アプリケーションの並列性を低下させ，スケーラビリティ低下の原因になる．SRSW 型の場合，複数プロセスが同じキャッシュブロックにアクセスを行うだけでそれらが実質的にミューテックスと同様に逐次化される．MRSW 型の場合も依然として書き込みが逐次化されるので，実質的にキャッシュアクセスがリーダーライターロックのように働く．そのため，スケーラビリティが重視される DSM 処理系においては，MRMW 型プロトコルが盛んに研究されてきた．

MRMW 型プロトコルにおいては，同一オブジェクトが複数プロセス書き込み可能になりうるので，それらが実際に同一オブジェクトに書き込みを行った際にどのプロセスの書き込みが最終的な結果になるか不確定になる．2.4.1 項で述べたように，複数スレッドによる happens-before で順序付けられない書き込みはデータレースであり，決定的な実行結果を得ることがそもそもできない．そのため，データレースが起きないとした場合，一見すると MRSW 型と MRMW 型は同一の振る舞いをするようにみえる．仮に MRSW 型で書き込み中プロセスを制限したとしても，複数の書き込みプロセスが存在しなければ逐次化部分は増大しない．

MRMW 型プロトコルが必要となる理由は，**“False sharing”** と呼ばれる現象に対応するためである．False sharing とは，ユーザプログラムとしては別々のオブジェクトであるにも関わらず，コヒーレンス粒度として同じメモリブロック上であるという状況を指し，実際には共有していないと思っていた変数同士がコヒーレンス粒度としては共有されている状況を意味する．MRSW 型プロトコルにおいてあるメモリブロックに False sharing が起きている場合，書き込みを行うプロセス同士でキャッシュが往来する “ピンポン現象 (cache ping-pong)” が発生する．このような状況が発生すると，書き込みレイテンシが増大するだけでなく，書き込み毎にスレッド間が逐次化されてスケーラビリティが低下する．一方で，MRMW 型の場合は同一メモリブロックに複数の書き込み中プロセスが存在できるために，False sharing が発生してもスケーラビリティは低下しない．

False sharing が起きているメモリブロックが複数のプロセッサによって書き換えられる場合，MRMW 型プロトコルではそれらの複数の書き込みを 1 つのメモリブロックに併合 (merge) する必要がある．ソフトウェア DSM 処理系で併合に使われるのが **diff** と呼ばれる手法で，更新前/後のページの差分を取ってそれを適用することで併合を行う．diff を初めて導入したのが，Munin [58] という初期のソフトウェア DSM 処理系である．

ページに書き込まれた場合に実際に diff を計算するには，書き込む前のページを保存しておく必要があり，その保存されたページを **twin** と呼ぶことが一般的である．ページベース DSM の場合は，書き込みが開始される前のページは書き込み禁止の状態にしておき，実際に初めて書き込みが行われた時に twin を作ればよい．

diff の適用については，RMA や AM で実装することができる．Noronha ら [59] は InfiniBand 上で RDMA WRITE with Immediate を用いて diff を実装しており，実質的に AM に基づいているといえる．RMA による実装は AM のようにリモート CPU を使用しない利点があるが，メッセージ数が多くなる欠点がある．

2.5.3　コヒーレンスプロトコルにおける Acquire 型と Release 型

緩和型コンシステンシモデルでは，キャッシュへの書き込みがいつ実際に読み込むプロセスに伝搬していくかをシステムが比較的自由に決定できる．MRMW 型プロトコルの場合は，diff による更新をいつどのプロセス上のメモリで実行するべきか，という議論になる．更新を早めに完了しておくプロトコルを **Eager 型**，実際に読み込む直前まで更新を遅延させるプロトコルを **Lazy 型**であるとしばしば呼称する．Release Consistency では，書き込み側が release を行った際に更新が起きるプロトコルを **Release 型**，読み込み側が acquire を行った際に diff を収集するプロトコルを **Acquire 型**と呼ぶ．Release 型は Eager 型であり，Acquire 型は Lazy 型であるといえる．

Release Consistency を保証する DSM としては，DASH [41] というハードウェア DSM が最初に提案された．DASH では，図 2.5 のように，メモリ書き込み時に即座に全共有プロセスに書き込みメッセージを送信するが，書き込んだ

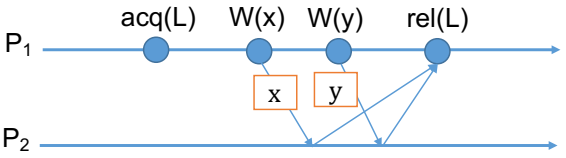


図 2.5: DASH におけるコヒーレンスのための通信例

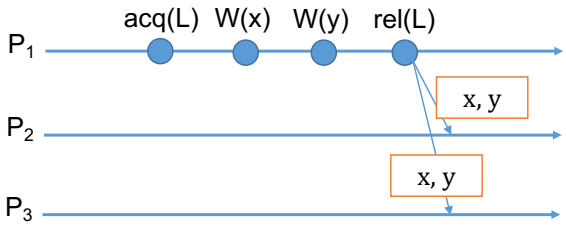


図 2.6: ERC におけるコヒーレンスのための通信例

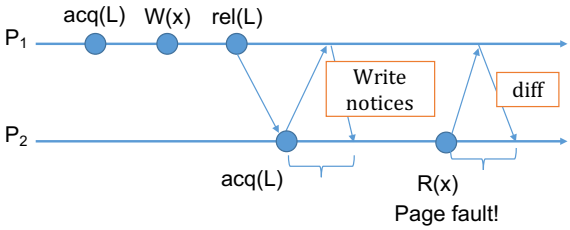


図 2.7: LRC におけるコヒーレンスのための通信例

プロセスはそれらの返答を待たずに次の命令に移る．Release Consistency に従うためには，release の前に実行したメモリアクセスは大域的に実行されている必要があるため，release が実行された時だけ発行済みの書き込みが全て完了するまで待つ．このような手法によって，書き込み毎のストールが削減されている．

ソフトウェア DSM である Munin [58] では Eager Release Consistency (ERC) というプロトコルを提案している．Release Consistency をソフトウェアで実装する際，DASH のようにメモリ書き込み毎の通信を行うことはオーバーヘッドが大きすぎるという問題があった．ERC では，図 2.6 のように，書き込みを即座にリモートノードに送信する必要が無いことを利用して，メッセージ送信を release 時まで遅延し，release 時にまとめて全共有プロセスに送信する．ERC においてキャッシュミスが発生した場合は，まずディレクトリマネージャにページを要求するメッセージが送信される．ディレクトリマネージャは，その時点で最後に release を行ったノードにメッセージを転送され，オーナーノードがキャッシュミスを起こしたノードに最新のデータを送り返す．

ERC では，データ書き込み毎に通信が発生せず，release 時にまとめてメッセージが送信されるため，メッセージ数を削減できる利点がある．また，DASH 同様に acquire 時には何も処理が必要ない．一方で，キャッシュを持っている全ノードに対してメッセージを送信する必要があるため，キャッシュしているだけで実際にはそのデータを必要としないノードにもメッセージを送信してしまい，不必要な通信が多発してしまうという問題があった．

Lazy Release Consistency (LRC) [48] [42] は，ERC の改良として登場したプロトコルである．LRC は，ERC で問題となっていた不要な通信を削減するためのプロトコルである．LRC の基本的なアイディアは，実際に各ノードがデータを必要とするのは acquire の発行以後であるため，acquire 時に初めて最新のデータを集めればよいのではないかというものである．LRC において，各ノードのキャッシュには，どのノードがそのページにいつ書き込んだかを記録したベクトルタイムスタンプ [60] という情報が付与される．acquire 時には，acquire を実行したプロセッサ p に記録されたベクトルタイムスタンプが，その時点で一番最後に release を実行したプロセッサ q に送信される．プロセッサ q はこれを用いて，q で実行済みだが p ではまだ実行されていない全ての変更を “Write Notices” として返答する．この Write Notices によって，プロセッサ p はキャッシュの更新に必要な diff をどのノードから集めればよいかが分かるので，それらのノードに diff を要求し，全ての diff を集め終わったら元の処理を再開する．release 時には，ERC とは逆に何もする必要がない．

LRC には Invalidate 型・Update 型（2.6 節）の 2 通りの実装手法が存在する．Invalidate 型では，acquire 時に Write Notices を受け取ったキャッシュを全て無効化する．キャッシュミスが発生すると，Write Notices を元に全ての必要な diff を他のプロセッサに問い合わせて，それらの diff を集めた後自ノードのキャッシュに書き込んでから処理を再開する．Update 型は，acquire 時に自ノードのキャッシュへの書き込みも含めて処理するプロトコルである．

図 2.8 に，ERC と LRC におけるメッセージ数の比較を示す．LRC は，ERC と比べると，不要なメッセージが減っ

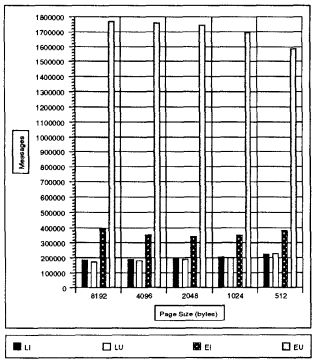


図 2.8: Cholesky ベンチマークにおける LRC と ERC のメッセージ数の比較 [42] (LI: Lazy Invalidate, LU: Lazy Update, EI: Eager Invalidate, EU: Eager Update)

たためにメッセージ数が減少していることが分かる．また，LRC においては diff だけを転送するため，データ転送量も削減される．

一方で，LRC には課題も多い [61]．まず，ソフトウェアで diff を処理することはオーバーヘッドが大きい．また，acquire を行って一旦キャッシュを無効化すると，複数のノードから diff を取得する必要がある．さらに，diff や Write Notices を記録するためのメモリ消費量が大きいという問題もある．diff や Write Notices を保持しているプロセッサは，他のノードがその情報を依然として必要としている可能性があるため，ローカルな情報だけでは破棄してよいか判断できない．そのため，これらを破棄するには定期的に大域的なガーベッジコレクションが必要となる [42] が，その処理もソフトウェアオーバーヘッドの要因となる．

2.5.4　ホームベース DSM とホームレス DSM

Home-based LRC (HLRC) [61] [62] は，LRC を改良した手法である．HLRC では，各ページごとに固定されたホームノードを設けて，ホームノードにほぼ最新のデータを保持させる．HLRC は，Automatic Update Release Consistency (AURC) [63] というプロトコルのアイディアを元にして考案されている．AURC は，ハードウェアの機能としてローカルメモリへの変更をリモートメモリにも反映させる機能を用いて，ホームへのライトスルーを行う手法である．AURC はライトスルーによって処理が単純化されており，多くの利点が存在する．AURC は，LRC における問題を次のように解決している．

- diff を求めるコストが不要．
- ホームノード上でのページフォールトが発生しない．
- ホームでないノードであっても，1 往復のメッセージでキャッシュを最新に更新できる．
- Write Notices を保存する必要がなく，メモリ消費量が LRC と比べて少ない．

しかし，AURC は特殊なハードウェア機構を要求するという問題があったため，ソフトウェアのみで実装可能な方法として HLRC が提案された．HLRC は，LRC 同様に acquire を実行した際に初めてデータを取得するプロトコルであるが，diff の適用方法が異なっている．acquire を実行したノードは，最後に release を発行したノードに対して自分のベクトルタイムスタンプを送信するが，これを受け取ったノードは acquire を行ったノードに diff を直接送信せず，一旦ホームに書き戻すという処理を行う．acquire を行ったノードは，ホーム上のページへの diff の適用が終わった時点で，ページ全体を丸ごと読み込む．

図 2.9 に，HLRC における通信の例を示す．acquire を行ったノードは，複数のノードから diff を集める必要が無く，ホームノードから 1 往復のメッセージで最新のデータを取得することができる．HLRC が AURC と異なる点として，HLRC は次の release まで書き込みを遅らせるという点がある．また，HLRC は複数の変更を diff として 1 つのメッセージで送信できるのに対して，AURC は複数のメッセージを使用するという点がある．HLRC は AURC よりも diff によるソフトウェアオーバーヘッドがある代わりに，メッセージの数を削減している．

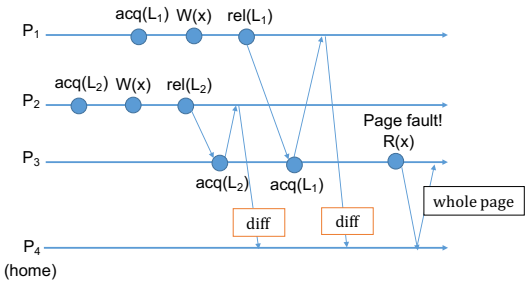


図 2.9: HLRC におけるコヒーレンスのための通信例

表 2.3: Release Consistency を実現できるコヒーレンスプロトコルの比較

プロトコル	ホームノード	release 時の動作	acquire 時の動作	ページミス時の動作
Eager Release Consistency	なし	キャッシュ中の 全ノードに変更送信	何もしない	キャッシュを持っている ノードから取得
Lazy Release Consistency	なし	何もしない	最後の releaser から Write Notices を取得	全ての writer から 必要な diff を収集
Home-based LRC	あり	何もしない	各 writer にホームへの 書き込みを要求	ページ全体を ホームから読み出す
Carina [65]	あり	ローカルの変更点を 必ずホームへ書き込む	ローカルに判断して キャッシュ無効化	ページ全体を ホームから読み出す

ホームベースプロトコルとホームレスプロトコルのどちらがよいかは，議論が分かれる問題の一つである．例えば Zhou ら [61] は，LRC のようなホームレスプロトコルはガーベッジコレクションなどの問題があるため，ホームベースプロトコルの方がホームレスプロトコルと比べて高性能だった，と指摘している．しかし，一般のプログラムに対して，通信とメモリの特性からこれを定量化することは容易ではない．ホームベース・ホームレスの議論と似た議論として，マルチコアプロセッサのキャッシュメモリの文脈 [64] において，上位階層にキャッシュされていれば下位階層に必ずキャッシュがある Inclusive caches と，そうではない Exclusive caches という 2 つの実装方式が存在する．

2.5.5　コヒーレンスプロトコルのまとめ

表 2.3 に，Release Consistency を実装できるコヒーレンスプロトコルの比較を示す．Kariras らによる Carina [65] については 2.6.4 項で解説するが，ここではこれまで述べたプロトコルとの比較について述べる．他の 3 つと比較して，Carina はホームベースかつ Eager プロトコルであることから，“Home-based ERC” に相当する戦略を取っていることが分かる．Release 型はデータ読み出しのレイテンシを縮める効果があり，バンド幅を消費する可能性がある代わりにホームに書き出すタイミングが早くなってレイテンシを削減できる．

2.6　共有プロセスの管理手法

何らかの共通のデータを共有しているプロセスを，一般に**シェアラー (sharer)** と呼ぶ．共有メモリにおいては，あるメモリ領域をキャッシュしているプロセスがシェアラーである．本稿ではこれを共有プロセス (sharing process) とも呼んでいる．

あるプロセスがあるメモリブロックに書き込みを行った場合，コンシステンシモデルにもよるが，原則的に全ての共有プロセス上のキャッシュブロックが更新されなければならない．書き込みが起きた時に共有プロセス上のキャッシュブロックを実際に更新するプロトコルを **Update 型**，更新せずに無効化だけを行っておいてもう一度必要になった時に再度読み込むプロトコルを **Invalidate 型**と呼ぶ．Update 型と Invalidate 型のどちらが優れているかには古くから論争がある [5] が，Update 型のプロトコルは実際にデータが必要ない場合でもデータを更新するという問題があり，ソフトウェアオーバーヘッドが大きくなると予想されるため，本稿では Invalidate 型を基本として議論を進める．

Invalidate 型のキャッシュにおいては全ての共有プロセスに無効化メッセージ (invalidation messages) を送信する必要があり，Update 型は更新するデータ自体を送信する必要があるので，いずれの場合も全ての共有プロセスを特定する

必要がある．共有プロセスをディレクトリに記録する手法のことを**ディレクトリベースプロトコル (directory-based protocols)** といい，これに対して常に「全てのプロセスが共有している可能性がある」として全プロセスに通知する手法を **ブロードキャストプロトコル (broadcast protocols)** あるいは **スヌープベースプロトコル (snoop-based protocols)** という．

ブロードキャストプロトコルが採用されるのは全プロセッサでのブロードキャストが容易な場合のみであり，バス (bus) 型ネットワークで構成できる数十コア程度の小規模なハードウェア共有メモリシステムに限られている．数千以上の大規模なコンピュータシステムはより複雑なトポロジを持ったネットワークであるので，無効化メッセージは共有プロセスそれぞれに対して送信する必要がある．そのため，ブロードキャストプロトコルの場合はプロセス数を P とした時，無効化メッセージの送信先となるプロセス数も P となって，大規模システムでは明らかにスケーラブルではない．ディレクトリベースプロトコルでは，ある時点での共有プロセス数を P_s としたとき，その時点でデータを書き換える場合に Invalidation を送信するプロセス数も P_s となる．ブロードキャストプロトコルに比べると削減できているが，依然として最悪の場合 P になるという問題は残されている．

2.6.1　共有プロセス数の削減と Self-invalidation

Self-invalidation [66] とは，データを書き換えるプロセスが無効化メッセージを送信するより前に，共有プロセスが自発的にキャッシュブロックを破棄する手法を指す．Self-invalidation は共有プロセス数 P_s の削減手法の一つである．Self-invalidation を行うプロセスは，自プロセスのキャッシュを無効化するとともに，ディレクトリに対して自プロセスでの共有の終了を通知する．Self-invalidation が事前に行われていれば，その後書き込みを行うプロセスは無効化メッセージを送信せずに書き込みを始められる．Self-invalidation はノード内の CPU 並列性が十分であればユーザプログラムと並列にシステム内部で実行できるため，無効化メッセージの送信を減らして書き込みのクリティカルパス長（3.2 節）を縮めることができる．

キャッシュを行うシステムは一般に，各プロセスが保持できる最大キャッシュサイズ C に限りがあるので，容量を超えた際に**キャッシュ置換 (cache replacement)** あるいは **追い出し (eviction)** を行って不要なデータを適宜破棄する．Self-invalidation はこのキャッシュ追い出しの特殊な形態であるとみなすことができる．通常のキャッシュ置換ではキャッシュサイズを超えるまでデータをキャッシュし続けるが，Self-invalidation はキャッシュサイズと無関係に将来的な無効化されそうなキャッシュブロックを破棄するという違いがある．

書き込みの前に共有プロセス数を削減する方法は必ずしも Self-invalidation だけではなく，それ以外の方法も考えられる．例えば，ディレクトリを保持しているプロセスが，共有プロセスに対して事前にデータを無効化するよう指示することも可能であろう．そのような手法に対する Self-invalidation の利点として，各プロセスは自身のアクセス履歴を全て追跡することが可能なので，そのアクセス履歴を使うことでキャッシュが不要なのかどうかを判断できることが挙げられる．

2.6.2　Self-invalidation の予測器

キャッシュ置換の分野においては，**OPT アルゴリズム** [67] という最適な置換アルゴリズムが知られている．OPT は未来の全てのメモリアクセスが分かっていることを前提とするので，現実のシステムでは実装不可能である．そのため，現実のキャッシュ置換アルゴリズムは何らかの手法で未来のメモリアクセスを予測し，OPT の近似を行う．最も有名な置換アルゴリズムは Least Recently Used (LRU) であり，最古にアクセスされたキャッシュブロックを破棄する．近年では LRU 以外のキャッシュ置換アルゴリズムも一般的になっており，代表的な手法に Re-Reference Interval Prediction (RRIP) [68] が挙げられる．RRIP はキャッシュブロックごとのカウンタを用いる手法で，全カウンタを一定時間ごとに定期的に更新するとともにヒット時にも更新することで，将来的に必要なキャッシュブロックを判断する．

Self-invalidation も未来の書き込みについての情報が必要であり，キャッシュ置換同様に予測が必要となる．最も単純な手法は “**decay**” [69] と呼ばれる手法で，各キャッシュに寿命を表すカウンタ値を持たせて，その寿命が尽きた時にキャッシュブロックを破棄するというものである．decay の手法は，キャッシュブロックごとのカウンタを使う点

で RRIP と似ている．

Lai ら [70] は，Last Touch Predictors (LTP) と呼んでいる仕組みによって Self-invalidation を実行すべきタイミングの予測を試みている．Lai らは緩和型コンシステンシを前提としておらず，メモリの読み出しが必ず最新の値を読むような厳しいコンシステンシモデルと，MRSW 型のコヒーレンスプロトコルを想定している．その場合，（Single Writer で Exclusive の状態にある）書き込み中のキャッシュブロックを他プロセスから読み込むには，書き込み中プロセスに対して無効化メッセージを送って，そのキャッシュブロックを読み込み可能にする．この際，書き込みを無効化することは読み込みレイテンシの増大につながる．そのため，Lai らは書き込みを行ったプロセスが自発的にキャッシュを無効化する Self-invalidation をいつ発行すべきか予測することを試みた．その場合，書き込んだプロセスが「そのキャッシュブロックを最後に書き換えた (last touch) 直後」にキャッシュを無効化しておけば，後で他のプロセスがアクセスする際に無効化メッセージの送信が必要なくなる．

Lai らは last touch を予測するため，トレースベースの分岐予測器 [71] に似た手法を採用した．具体的には，まずあるキャッシュブロックに書き込んだプログラムカウンタ (Program Counter, PC) の値を履歴（トレース）として記録していく．PC 値を全て記録すると当然記憶容量が不足するので，PC 値を加算して丸め込むことでエンコード値に変換する．そして，無効化メッセージが送信されてしまったトレースを last touch であったとして学習する．そして，次回同じトレースで書き込みが行われた場合に，last touch である確率が高い命令であったとしたらその書き込みを即座に Self-invalidation で無効化する．このようにして，last touch をプログラムの実行パスから精度良く予測することを可能にしている．

2.6.3　Private/Shared 分類

近年になって，マルチコアプロセッサの共有メモリシステムで注目されているのが，**Private/Shared 分類 (Private/Shared classification, P/S 分類)** [72] [69] と呼ばれる手法である．その基本的な考え方は，「キャッシュブロックが共有されてないと各プロセスが独自に判断できれば，コヒーレンス操作を省略できる」というものである．P/S 分類を行う手法では，各ページを読み書きを行えるプロセス数によって次のように分類する．

Private (P)　単一プロセス（オーナーと呼ぶ）がキャッシュブロックを保持している．
Shared (S)　複数プロセスがキャッシュブロックを保持している．

Esteve ら [69] によると，P/S 分類によって改善される状況は次のように挙げられる．

- ディレクトリベースプロトコルで，Private ブロックを追跡しないという**コヒーレンス非活性化 (coherence deactivation)** によって，ディレクトリに必要な容量を削減できる [73] .
  - ハードウェア共有メモリにおいてディレクトリに利用可能な容量は限定的であるため，不要なディレクトリの情報を削減するとキャッシュの効率を上げることができる．
- Non-Uniform Cache Architecture (NUCA) において，アクセスレイテンシを削減する [74] .
  - NUCA においては Last-Level Cache (LLC) のアクセスレイテンシがコアごとに一定でないため，Private ブロックをコアの近くに配置することがレイテンシ削減に貢献する．
- ブロードキャストベースプロトコルにおいて，ブロードキャストによるトラフィックを削減する [75] .
  - Private ブロックに対するスヌーピングをしないことでトラフィックが削減される．
- コヒーレンスプロトコルを Private と Shared で切り替えることで，計算量を削減する [76] .
  - ソフトウェア DSM に効果があるのはこの点である．

2.6.4　P/S3 分類

Kaxiras ら [65] は，コンシステンシモデルとして Carina を提案している．また，Carina に基づいた DSM 処理系として Argo を開発している．Kaxiras らは，近年のハードウェアの事情を考慮すると，DSM 処理系においてソフトウェアオーバーヘッドとネットワークレイテンシに起因した性能低下を避けることが最も重要であり，通信バンド幅

の削減はそれらに比べるとあまり重要でないと指摘している．近年のハードウェアの動向として，CPU の高速化が伸び悩む一方で，ネットワークのバンド幅は依然として向上し続けている [77]．20 年前と比べるとネットワークのレイテンシも削減されているものの，近年ではバンド幅に比べるとその縮み幅が小さくなってきている．

ネットワークレイテンシによる性能低下を防ぐには，可能な限りリモートとの通信を発生させず，それぞれのプロセスが独立してキャッシュコヒーレンスの処理を判断できるようにすることが有効である．そのためには，キャッシュコヒーレンスを保つためのディレクトリ情報を各ノードに分散させる必要があるが，今度は分散したディレクトリ情報同士のコンシステンシの問題が生じる．また，ソフトウェアオーバーヘッドの削減という観点でみると RDMA の活用が有効であるが，RDMA で実行できるメモリ操作は限定的であるため従来のリモート CPU の介在を前提としたコヒーレンスプロトコルとは異なった実装手法が必要となる．このような事情から，Kaxiras らは分散されたディレクトリ構造を提案し，それを RDMA のみで操作する新たなコヒーレンスプロトコルを提案している．

Carina で定義されるメモリバリア操作は，self-invalidation フェンスと self-downgrading フェンスである．self-invalidation フェンスは，あるノード上の全てのキャッシュを無効化する．self-downgrading フェンスは，あるノード上におけるキャッシュへの変更を全てのノード上で大域的に可視にする．先述した Self-invalidation はシステムがあるキャッシュブロックを自発的に無効化することであるが，Carina のメモリバリアである self-invalidation フェンスは（後述するように）全ての場合で Self-invalidation を全キャッシュブロックに対して行うわけではない．

Carina は各種緩和型コンシステンシモデルを実装するために用いることができる．例として Release Consistency（2.4.3 項）であれば，acquire でロック獲得後に self-invalidation フェンスを，release でロック解放前に self-downgrading フェンスを実行すればよい．self-invalidation フェンスを実行すると自プロセス上の全キャッシュが一旦無効化されるので，acquire の際に必ず self-invalidation フェンスを挟めばクリティカルセクション内の命令が acquire の前に実行されるということは起きない．同様に，self-downgrading フェンスの実行時に自ノードの全キャッシュは全てホームに書き戻されるので，クリティカルセクション内の命令が release 後に実行されるということも起きない．

acquire 時に毎回 self-invalidation フェンスによって古い可能性のあるキャッシュを無効化することで，必ず他のノードによる変更を取り込めるようにはなるが，その後データに再度アクセスした際にホームに問い合わせる必要があるので，結果的にクリティカルセクションが起きる度にその後通信が大量に発生する．そこで，他のノードがページを更新していないと分かる場合に，self-invalidation フェンスが実行されても実際には無効化せずに処理を進めることを考える．この際に共有プロセス数をリモートに問い合わせるようなプロトコルでは通信レイテンシの問題が不可避なため，自ノード上にあるディレクトリキャッシュのみで Self-invalidation の必要性を判断することが必要となる．

図 2.10 に，Argo のディレクトリ構造を示す．ディレクトリエントリ 1 つが 1 ページに対応していて，そのページがどのページによって read/write されたかが記録されている．readers/writers の記録には，プロセッサ数分のビット列を使用している．各ページのホームノードには，そのページに対応するディレクトリエントリが必ず存在し，このディレクトリエントリは read/write しているノードが増える度に更新されるので，常に最新である．一方で，各ノードが持つディレクトリキャッシュは，実際に Self-invalidation と Self-downgrading が起きた場合にのみ更新される．

Private ページはオーナーからしかアクセスされていないので，別のノードにキャッシュされない限り，Self-invalidation も Self-downgrading も不要である．しかし，別のノードが新たにキャッシュする際に，そのノードは元のノードが最後に Self-downgrading を実行した時点のデータが観測されるようになる必要がある．これは，Private ページが新たにキャッシュされる場合は，オーナーに Self-downgrading を依頼する必要があるということになる．これは片方向通信だけでは困難であり，新しくキャッシュする際のレイテンシの増大にもつながる．そこで，Argo では，Private なページであっても必ず Self-downgrading を行うというプロトコルを採用することで，新しくキャッシュされた際の処理も片方向通信で行うという手法を取っている．この手法によって，他のノードが同じページをキャッシュして Shared に遷移したとしても，オーナーは常に Self-downgrading を行っているので，即座に最新のデータを取り込むことが出来る．

Private と Shared を区別することで，Private ページの Self-invalidation を防ぐことができるようになったが，依然として Shared になると全てのノードが Self-invalidation を行う必要がある．そこで，Shared ページの状態を，書き込むノードの数に応じて更に 3 つに細分化する．

No Writer (NW)　どのノードもそのページに書き込んでいない．つまり，ページが Readonly である．

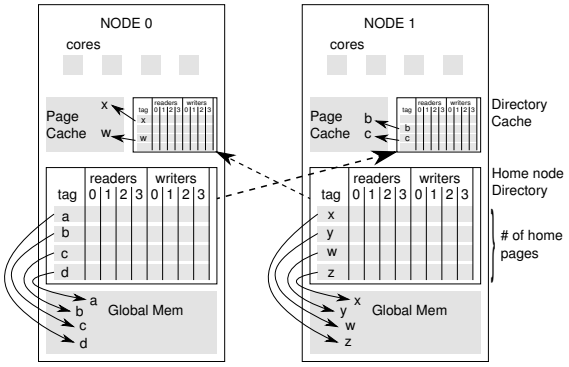


図 2.10: Argo DSM におけるディレクトリ構造 [65]

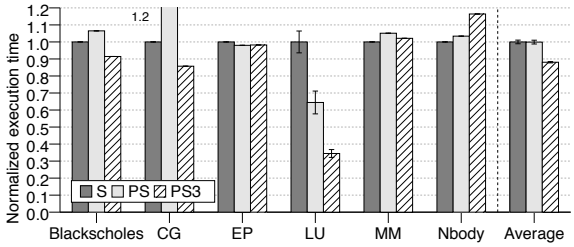


図 2.11: Argo DSM におけるページ分類による実行時間への影響 [65]

Single Writer (SW)　ある 1 つのノードしかそのページに書き込んでいない．
Multiple Writer (MW)　複数のノードがそのページに書き込んでいる．

NW の場合は，全てのノードが Self-invalidation をする必要がない．SW の場合は，そのページに書き込んでいるノードでは Self-invalidation が必要ないが，それ以外にページを読み込んでいるノードは最新の変更を取り込む必要があるので，Self-invalidation が必要である．MW の場合は，全てのノードが他のノードによる変更をマージする必要があるので，常に Self-invalidation を行う必要がある．

図 2.11 に，ページの分類手法による実行性能への影響を示す．S は全てのページが Shared であると仮定する手法（常に Self-invalidation と Self-downgrading を行う手法），P/S は Private なページで Self-downgrading を行わない手法，P/S3 は Carina の分類手法 (常に Self-downgrading を行い，writers の数によっても分類) である．ほとんどのベンチマークで，P/S3 が最も高速であったことが分かる．

2.7　共有メモリシステムのスケーラビリティ

共有メモリシステムが大規模環境でスケーラブルではないと考える研究者は多い．「共有メモリのモデル自体がスケーラブルでないこと」を理論的に証明した研究は筆者の知る限り存在しないが，一方で実際に共有メモリを実装したハードウェアで数千プロセスまでスケールする例は知られていないことから，現時点で実用上問題があることは明らかである．共有メモリのスケーラビリティに関する問題点は，以下のように挙げられる．

1. ユーザプログラムがそもそもスケーラブルでない．
   - 仮にメモリシステムが完全にスケーラブルであり PRAM 同様とみなせたとしても，共有メモリのモデルそのものが許すパターンでスケーラブルでないプログラムは記述可能である．
   - 一方で，システムの事情とはほとんど無関係に，スケーラブルな共有メモリのユーザプログラムは記述可能である．自明に並列化可能 (Embarrassingly parallel) なマルチスレッドプログラムにおいて，全スレッドがそれぞれ異なるアドレスにアクセスしているなら，システムが不必要に単一のリソースアクセスを集中させない限りにおいて自明にスケーラブルである．
2. アドレス空間が全メモリを表現できない．

- 64-bit アドレス空間でも 2^{64} バイトを表現できるので，現在のトップクラスのスーパーコンピュータ [78] は約 1PiB（$\approx 2^{50}$ バイト）のメモリを保持しているので，64bit でも依然として全計算ノードの全メモリアドレスを表現することができる．
- あくまでシステム実装上の問題であり，共有メモリモデルそのものの問題ではない．C や C++ の言語仕様においてポインタサイズが決まっているわけでもない．
- 実装上の解決策として，例えばマルチスレッド環境のソフトウェア DSM においては，Kee ら [79] が述べているように anonymous mapping と fork() システムコールを用いてエイリアシングを起こす方法によって広大なアドレス空間を確保する方法が知られている．また，アーキテクチャがアクセス可能なメモリ領域の制限は，コンパイラでポインタ値を変換するといった手法によって回避できる．
- 現実的な視点では，スーパーコンピュータの全メモリを全てのアプリケーションが必要とするわけではなく，コヒーレンスプロトコルといった他の問題に比べて優先度が低い．

3. システムが厳しいメモリコンシステンシモデル (strict memory consistency models) を採用している（2.4 節）．
   - ハードウェアの共有メモリで採用されているような厳しいメモリコンシステンシモデルでは，一般のプログラムに対してスケーラブルな実行を期待することはそもそも困難である．
   - DSM においては緩和型コンシステンシモデルの採用が一般的であり，それに基づいて記述されたユーザプログラムにはこの問題は発生しない．
4. コヒーレンスプロトコルの都合上，キャッシュの複製数に制限がある（2.5.2 項）．
   - コンシステンシモデルが仮に緩和されていても，キャッシュ複製数を制限していればプログラム実行時の並列性は低下する．
   - MRMW 型コヒーレンスプロトコルにはこの問題は生じないが，diff の使用によってオーバーヘッドが増大する．
5. Invalidate 型プロトコルで，無効化メッセージを送る共有プロセス数が増大する（2.6 節）．
   - ディレクトリベースプロトコルはブロードキャストプロトコルよりはスケーラブルだが，共有プロセス数は最悪で P まで増大する．
   - Self-invalidation のように共有プロセス数を積極的に減らす手法が有効であることは述べた．
6. SPMD 型実行におけるブロードキャストのような典型的通信パターンを簡潔に表現できないため，メモリアクセス時に必要とされるリソースが一箇所に集中する．
   - 共有メモリ上のプログラムとしてブロードキャストに相当する処理を明示的に記述することは可能であるが，システムの事情を考慮に入れてプログラムを記述することになるために非生産的である上，そのような Optimization もあまり一般的ではないため，有望な手法であるとはいえない．
   - システムとして自動的にこのような処理を高速化する手法には，階層的なキャッシュプロトコルの導入が挙げられる．例えば，データを保持しているオーナープロセスを特定するのに，階層構造を利用した Probable Owner [80] という手法が以前から知られている．一方で，階層構造が深くなれば深くなるほどデータアクセスのレイテンシが増大することになるため，本稿では階層的キャッシュについては取り扱わないが，将来的には再検討すべき事項である．

このように共有メモリシステムにはスケーラビリティ上の課題が多く存在するものの，それら全てに対して何らかの解決策が提案されているので，それらを正しく組み合わせることでハードウェア/ソフトウェア DSM 処理系の双方においてスケーラビリティを実現できる可能性は残されている．

第 3 章

並列プログラムの並列性記述とスケジューリングに関する関連研究

並列プログラムの記述としては，逐次プログラムを記述するだけでシステムが自動的に並列化してくれるものが理想的であるが，実際にはそのような並列性の自動抽出は容易ではない．ハードウェアレベルではアウトオブオーダープロセッサが実際に並列性の自動抽出を実現しているが，あくまで細粒度な命令レベル並列において可能な技術であり，スレッドレベルの並列性自動抽出には現時点で有力な手法がない．

そのため，現在主流の並列化手法は，プログラマが並列性に関するヒントをプログラム中に埋め込み，それに基づいてシステムが並列化を行う．そのヒントの方式も多種多様に存在するが，最も柔軟でかつ一般的なのはスレッドとして並列化可能なコード片を切り出す手法である．

3.1　並列性の表現手法 [81][82]

本節では，並列プログラムでの並列性の表現手法について述べる．メモリモデルの場合と同様に，並列性の表現だけで性能が決まるわけではなく，その実装も含めることで初めてスケーラビリティといった定量的指標を導出できる．

SPMD 並列は，並列実行するプログラム領域を指定して，それらを指定数分だけ並列実行するモデルである．その際，通常は最大の並列度を得るために物理的並列性と同じ分の並列性を指定する．SPMD 並列は MPI [1] や OpenMP [2] といった著名なインターフェイスで用いられているため，その概念も名前も広く知られている．SPMD 並列では，プログラム自体が持つ並列性とは無関係に，物理的なプロセッサ数といった値でプログラムの動作が変化する．このため，並列性記述として不自然なものとなり，並列性表現としては生産性が最も低い．

Bag of Tasks (BoT) とは，「関数を別途並列に実行できるが，その並列した処理と通信も同期もできない」というモデルである．分散メモリ上で BoT を実現するには，実行前に実行するプロセスに対して関数とその引数を転送すればよく，コールスタックの転送を考慮する必要がない．また，タスク同士の同期を行う必要もないので，システムがタスク間の依存関係を考慮する必要がない．

Fork/join 並列は，「関数を並列実行させた後，その関数の終了を待ち合わせる」というモデルである．スレッドを生成したスレッドは新しく生成したスレッドを必ず待ち合わせなければならないというのが Fork/join の制約である．Fork/join は柔軟性の高いモデルであり，分割統治法によるアルゴリズムを自然に表現できる利点がある．一方で，Fork/join のみであらゆる種類の並列パターンを記述できるわけではない．例としては，**パイプライン並列 (Pipeline parallelism)** [83] [84] [85] のようなパターンや，ミューテックスのような同期プリミティブを必要とするプログラムなどが挙げられる．

スレッドによる並列化は，これらのどのモデルも表現可能であり，最も柔軟性が高いため，本稿ではスレッドによる並列化の実現を目指して議論する．Pthreads によるプログラミングを Fork/join と称することもあるが，これは「Fork/join を表現できる」という意味であって Fork/join しか対応できないわけではない．Pthreads は fork したスレッドを必ずその関数内で join する必要はなく，detach などの機能も持つ．

3.2　スケジューラと計算量

グリーディスケジューラ (Greedy scheduler) とは，スレッドを実行していないアイドル状態のプロセッサがその時存在する実行可能 (“ready”) なスレッドを必ず実行するようなスケジューラである．PRAM モデルを仮定すると，グリーディスケジューラにおいては，Brent’s lemma [86] と呼ばれる次の式が成り立つ．ここで，P はプロセッサ数，T_1 は逐次プロセッサでの実行時間，T_∞ は無限個のプロセッサでの実行時間（クリティカルパス）である．

$$T_p \leq \frac{T_1}{P} + T_\infty \tag{3.1}$$

ランダムワークスティーリング (Random work-stealing) [87] においては，スケジューリングの挙動がランダムに決まるため，平均の計算量として次のような式が知られている．C はキャッシュ容量，n は十分小さい正整数である．

$$E[T_p] \leq \frac{T_1}{P} + nCT_\infty \tag{3.2}$$

PRAM モデルを仮定しているので，これらの式は「共有中のキャッシュブロックには無効化メッセージの送信が必要」という事実を無視している．最悪の場合だと「全てのキャッシュブロックが全プロセッサで共有されている」ということになるので，その場合第 2 項は P に比例した値になる．すなわち，大規模環境における共有メモリの式としては，これらの式だけではワークスティーリングスケジューラがスケーラブルであるとは言えず，共有プロセス数の上限となるようなより強い仮定が必要である．また，キャッシュアルゴリズムとして LRU を仮定しており，self-invalidation（2.6.1 項）のような特殊なキャッシュ機構も存在しないと仮定しているので，それらの仮定が満たされなければ C に基づく考察も成り立たなくなる．

3.3　ワークスティーリングスケジューラの実装手法

ワークスティーリングスケジューラは，一般によく知られたスケジューラの実装方式である．本節では，その実装手法について概要のみ述べる．まず，各プロセッサ（ワーカーとも）は実行可能なスレッドを保持するための **“ready deque”** を持ち，その deque にスレッドを push/pop することができる．各ワーカーはアイドル時に他のワーカーからスレッドを盗む (“steal”) ことが可能であり，その場合はワーカーが push/pop する方向の逆側からスレッドを取り出す．

3.3.1　ワーカーの振る舞い

スレッドの fork 時は，新たなスレッド ID を確保して新しいコールスタックに移った後，元のスレッドに継続のコンテキストを設定して，指定された関数を実行する．継続をスレッドとして ready deque に積む方式を一般に **work-first principle** [88] と呼ぶ．その逆は **help-first** で，生成したスレッドを ready deque に積む方式である．Scheme のように継続を陽に扱える言語も存在するが，C 言語を含む多くの言語では継続を取り出すことが難しいため，help-first の方が実装は容易である．一方で，work-first は逐次プログラムの実行順序を原則的に保存するため，逐次処理を想定したキャッシュメモリの挙動と相性が良いという利点がある．

あるスレッドを join する時は，そのスレッドが終了している場合は何もしないが，終了していない場合は自スレッドが待機 (“block”) する必要があるため，自スレッドの継続をそのスレッドから参照できるようにしておく．するとそのワーカーはアイドル状態になるので，ready deque からスレッドを取り出すか，スレッドを steal することでその実行に移る．

スレッドが exit する時は，まず自スレッドが他のスレッドから join されていないか調べ，join されている時はそのスレッドを再開する．そうでない場合はアイドル状態になるので，join で block した場合と同様に別スレッドに移る．

3.3.2　ready deque の実装

Cilk の実装 [89] において用いられた手法は THE プロトコル，あるいは ABP work-stealing と呼ばれる．ready deque の実装には長らく ABP が使われてきたが，Chase ら [90] は ABP が固定サイズの循環バッファに依存していることを指摘し，Chase-Lev deque と呼ばれる実装を提案しており，現在はこれが ready deque の基本的な実装方式として知られている．

Lê ら [91] は，Chase-Lev deque について，緩和型のコンシステンシモデルにおける厳密なセマンティクスに基づいた実装を提示し，POWER や ARM といった緩和型コンシステンシのアーキテクチャにおいて，実際に性能が向上することを示している．

3.4　コルーチンの実装手法

中断/再開できる関数のことを，サブルーチン (Subroutines) と対比して**コルーチン (Coroutines)** と呼ぶ．コルーチンを一旦中断させることで，プログラムの処理を並行な単位に分割できる．ワークスティーリングスケジューラにおいて継続を取り出すことと，コルーチンを中断できるようにすることはほとんど同一の機構である．

コルーチンを実現するにあたって，最初に問題になるのはコールスタックの扱いである．サブルーチンの場合は呼出時にコールスタックが伸長し，復帰時にコールスタックが縮小するだけでよかったが，コルーチンの場合は中断している間コールスタックを保存しなければならない．ワークスティーリングスケジューラにおいては，“cactus stack” と呼ばれる特殊なデータ構造としてこの問題を取り扱うことがある．

3.4.1　スタックフルコルーチン (Stackful Coroutines)

スタックフルコルーチン (Stackful Coroutines) とは，「各コルーチン呼出ごとにコールスタックを別々に確保するコルーチン」のことである．スタックフルコルーチンを単独で実装している例には，Boost.Coroutine [92] [93] が挙げられる．スタックフルコルーチンを並列スケジューラと組み合わせた処理系がユーザレベルスレッドであり，3.6 節で述べるユーザーレベルスレッド処理系も暗黙のうちにスタックフルコルーチンの実装を包含している．

スタックフルコルーチンの重要な技術は，「ユーザレベルのコンテキストスイッチ」である．カーネルレベルではなくユーザレベルでコンテキストスイッチを行うことで，システムコールのオーバーヘッドを減らし，利用者に合わせて保存するコンテキスト情報を減らすことも容易となる．POSIX [94] には ucontext というインターフェイスが存在したが，最新の POSIX の仕様では移植性の問題から既に削除されている．その他の実装として，POSIX の枠組みの範囲内では setjmp/longjmp に sigaltstack を組み合わせる方法 [95] が知られている他，Boost.Coroutine の下位レイヤーとなっている Boost.Context [96] も存在する．

スタックフルコルーチンあるいはユーザレベルスレッドの問題点は，以下のように挙げられる [97] ．

- コンテキストスイッチのオーバーヘッドが不可避．
  - カーネルレベルのコンテキストスイッチよりは高速であるが，しかしゼロにはできないという問題がある．特に，ジェネレータのような中断/再開を多発させるプログラミングパターンにおいて，そのオーバーヘッドは無視できないレベルに大きい．
- スレッド 1 つに対し，コールスタック 1 つ分の仮想アドレスを消費する．
  - コールスタック全体を保存しなければならないので避けられない問題である．
  - デフォルトスタックサイズとして，本来なら Linux の場合は 2MiB が必要であるが，プログラムが実際にそれだけ使用しなければそれより小さくすることはできる．
- スレッド 1 つに対し，最低でもページサイズ分の物理ページを消費する．
  - スタックオーバーフローを検出するために，コールスタックの端に相当する仮想ページをアクセス禁止にして，ページフォールトを検出することでエラーとしたりスタックを伸長したりすることができる．
  - スタックオーバーフローの検出を諦めて，ページサイズ未満の大きさしかコールスタックを用意しないと

いう割り切った手法も考えられるが，スタックオーバーフローが起きるようなプログラムをデバッグすることが困難になる．

Stackful Coroutines あるいはユーザレベルスレッドの問題点は，メモリ消費の観点からいってページベース DSM（2.5.1 項）の問題点と似ている．両者とも，OS と MMU によるメモリ保護機構を流用して，キャッシュ存在確認やスタックオーバーフロー検出といった高度な機能を実装しているという共通点がある．そして，メモリ保護機構におけるページサイズは固定されているため，それより細粒度の単位でデータを扱うことが出来ないということが，Stackful Coroutines やページベース DSM に共通の問題となっている．逆にいえば，ページサイズより細粒度にメモリ管理ができるハードウェア機構が備わっていれば，これらの問題は発生せず，より単純に Stackful Coroutines やページベース DSM を実装可能である．

Stackful Coroutines とページベース DSM の共通点は，コンパイラやハードウェアの改変なしにライブラリのみで実装可能であるという点にもある．対立した手法として，Stackless Coroutines とコンパイラベース DSM は，言語とコンパイラを変更することで同じ問題に対処しているという点でやはり共通しているといえる．

3.4.2　Split stacks

Split stacks [98] は，コンパイラの機能によってスタックの伸長毎にスタックあふれをチェックする手法である．Split stacks はスタックサイズを確認するために特殊なコードを挿入するため，コンパイラに特殊なオプションを要求する．Split stacks は通常のコールスタックの取り扱いをほとんど変化させずに，コールスタックのサイズを小さくすることが可能な手法である．一方で，Split stacks の問題は次のように挙げられる．

- 全ての関数呼び出しにスタックあふれ検査が挿入されるため，関数呼び出しのオーバーヘッドが大きい．
- 既存のライブラリを再コンパイルできないとすると，その中の関数に突入する度に通常のスタックを確保する必要があり，やはりオーバーヘッドが生じる．

3.4.3　スタックレスコルーチン (Stackless Coroutines)

スタックレスコルーチン (Stackless Coroutines) [97] [99] とは，コールスタックが保存されないコルーチンのことである．本項では，スタックフルコルーチンに対立する手法として，現在 C++ の標準化委員会で議論されているスタックレスコルーチンについて述べる．

スタックレスコルーチンの基本的アイディアは，中断/再開時に保存しなければならない自動変数を，コンパイラによってコールスタックではない特殊な領域（コルーチンフレーム）に配置することである．コルーチンフレームには自動変数に加えて最低限保存すべきレジスタなどしか置く必要がないので，その大きさは数十バイト程度でよく，中断されたコルーチンに必要なアドレス空間を圧縮できる．また，コンパイラによって中断/再開が認識されていると，コンテキストの保存を一部省略でき，場合によっては完全にインライン化することでコンテキストスイッチのオーバーヘッドを消滅させることも可能である．そのような特長から，スタックレスコルーチンの用途はスレッド処理系だけでなく，ジェネレータのようなパターンにも応用できるという点で汎用性が高い．

一方で，現在のところスタックレスコルーチンを利用可能なコンパイラは現在のところ Visual C++ しかなく，仕様策定中であることから研究目的でも利用することは困難である．また，スタックレスコルーチンは逐次プログラムに対して追加のアノテーションを要求 [100] し，スレッドに基づく通常の並列プログラムに対して修正が必要な点も問題であるといえる．

3.5　スタックフルコルーチンのプロセス間移動手法

分散メモリシステムにおいてプロセス間でワークスティーリングを行う際，最初にコールスタックの扱いが問題となる．グローバル変数やヒープ領域へのアクセスはプログラムを注意深く記述すれば回避できるが，コールスタックへのアクセスはプログラマが意識するだけでは回避不可能であることが背景にある．コールスタックの制御には当然

```
void f(int* p) {
    *p = 1;
}
int g() {
    int x;
    thread t{&f, &p};
    t.join();
    return x;
}
```

リスト 3.1: コールスタックのエイリアシングによって動作しなくなるマルチスレッドプログラム

ながらポインタが使われており，スタックフルコルーチンを別プロセスで再開するにはそのポインタが正常に機能するように取り扱わなければならない．

3.5.1　iso-address

自動変数の扱いは，コンパイラに基づく手法だとスタックレスにするなど実装系の自由度が高いが，ライブラリで実装する場合にはコールスタックとしてしか扱えないので特に問題となる．ライブラリ実装のための最もよく知られた手法が **iso-address** [101] で，「全ての一意なスレッド ID に対し，コールスタック用のアドレスを一意に割り振り，全プロセスでアドレス割り当てを共通化する」ことで，C 言語のコンパイラを変更せずに，コールスタックにアクセスするスレッドをそのまま別プロセスで再開することを可能とするものである．

iso-address は「異なるコールスタックに異なるアドレスを割り振る」手法であり，これはグローバルアドレス空間の考え方そのものである．そのため，共有メモリの観点で捉えることもでき，その場合は iso-address は**「コールスタック専用の緩和型コンシステンシ」**であると捉えられる．具体的には，スレッドが別プロセスに移動するときだけ，移動元と移動先の間でコールスタックを指すアドレス空間のコンシステンシが保たれている．iso-address は移動したスレッドのコールスタック領域のみしかコンシステンシを保たないので，DAG Consistency（2.4.4 項）よりも緩いモデルである．

3.5.2　仮想アドレス空間の節約手法

秋山ら [81] によって提案された uni-address は，仮想アドレス空間の節約を目的とした手法である．その基本的なアイディアは，あるプロセスのワーカー上で動作しているスレッドは 1 つだけにできるので，全てのスレッドのコールスタックを同じ仮想アドレスに割り振れば，仮想アドレス空間を節約できるというものである．また，仮想アドレスを節約することによって，RDMA に用いるレジストレーションの領域も節約することができる．

uni-address は，スタック領域に必要な仮想アドレスを節約するために意図的にエイリアシングを発生させているとみなせる．そのため，エイリアシングが起きているコールスタック同士でポインタを利用できなくなるという問題がある．uni-address に基づく処理系は，スタック領域を特殊なメモリ領域として扱うことになるので，比較的抽象度の高いモデルの中では PGAS ならば組み合わせ可能であるが，真の共有メモリ処理系として動作させることはできない．例えば，リスト 3.1 のような単純なコードを動作させることが不可能になり，その必然性をプログラマに説明することも困難となる．

uni-address は，スレッドを生成する度にコールスタックを退避しなければならないという問題点もある．そのため，各スレッドが消費するコールスタックのサイズが十分小さいことが前提となり，中断/再開を繰り返すようなスレッドの場合オーバーヘッドが大きい．

原ら [57] は，各プロセスごとにコールスタックの仮想アドレスをランダム化しておき，それらが実際に衝突した時にはフォールバックとして新しい OS プロセスを生成する random-address を提案している．random-address も uni-address 同様にエイリアシングを発生させる可能性があるので，真の意味での共有メモリシステムを実現することができない．uni-address にはコールスタックの退避が必要な問題があったが，random-address は衝突が起きない限り

においてオーバーヘッドが削減できる．一方で，コールスタックの衝突時に OS プロセスを生成する処理はシステムコールを要求し，オーバーヘッドも大きい．また，新規子 OS プロセスを生成することは環境によっては不可能な場合もある．さらに，同一ノード内であるにも関わらずプロセス間通信を使用することはシステム開発者にとって多大な負荷になるため，システム開発のコストの上でも問題である．

3.6　既存のユーザーレベルスレッド処理系

ユーザレベルスレッドライブラリの例として，MassiveThreads [102] [103] [104] （及びその先行研究である StackThreads/MP [105] [106] [107]），QThreads [108]，Argobots [109], TBB [110]，Nanos++ [111] などがある．言語の例としては，Cilk [89]，Cilk Plus [112]，OpenMP 3.0 以降 [2] などがある．これらの実装に共通していることは，ワークスティーリングとスタックフルコルーチンという 2 つの基礎技術に基づいていることである．但し，TBB に関しては work-first スケジューリングを採用していない点に注意する必要がある．

表 3.1 に，分散メモリ上のタスクスケジューラ処理系の比較を示す．継続がプロセス間で移動できないようなモデルは “tied tasks” と呼ばれ，スタックフルコルーチンのプロセス間移動（3.5 節）について考えなくてよいため実装が容易であるが，ワークスティーリングの利点を生かせず並列性が低下する．

表 3.1: 分散メモリ上で動作するタスクスケジューラの比較 [82]

モデル	ライブラリ	スレッドの中断/再開	継続のプロセス間移動 (untied tasks)	並列性表現
Distributed Cilk [7]	N	Y	Y	fork/join
Satin [113]	N	Y	N	fork/join
Tascell [114] [115]	N	Y	Y	fork/join
Scioto [116]	Y	N	N	BoT
HotSLAW [117]	Y	Y	N	fork/join
X10/GLB [118]	Y	N	N	BoT
Grappa [36]	Y	N	N	BoT
MassiveThreads/DM [81]	Y	Y	Y	fork/join

3.7　並列性記述とスケジューリングについての要約

これまでの論点をまとめると，スケジューリングやスレッドに関する問題は，以下のように挙げることができる．

1. スタックフルコルーチン（あるいはユーザレベルスレッド）は，（全プロセスの）全コルーチンに対してコールスタックが必要なため，それを保持できるだけの仮想アドレスが足りない．また，コルーチン 1 つに対して物理ページも 1 つ消費する．
   - 3.5.2 項でみたような仮想アドレスの節約手法では，真の共有メモリを実現できない．
   - 解決策としては，スタックレスコルーチンのようなコンパイラベースの自動変数の管理手法がある．歴史的にはユーザレベルスレッドはライブラリのみで実装できたことに価値があったため，コンパイラベースの手法はある意味では回帰しているということもできる．既存のプログラムの修正を最小化できるかがコンパイラベースの主な争点である．
2. （例えユーザレベルであっても）コンテキストスイッチのオーバーヘッドが存在する．
   - コンパイラによる手法によって同様に解決されると期待されるが，コンパイラがコンテキストスイッチの行き先も含めて適切に実行パスを認識することが必要である．
3. ランダムワークスティーリングより理論的に高速であると示されたスケジューリング手法は，現在でも存在しない．
   - PRAM モデルが非現実的なのは明らかだが，具体的なメモリモデルや通信モデルに基づいてクリティカル

パス長を必ず縮めると保証したスケジューリング手法は知られていない．

コルーチンの実現にはコールスタックの扱いが問題となり，そのコールスタックはメモリ上に置かれているから，メモリモデルと分離して議論することがそもそもできない．このことが，本稿で述べる DSM と ULT の組み合わせという概念に発展する根拠の一つになっている．スレッドの実現だけを切り出した課題として取り組むよりも，メモリモデルを含めて包括的に議論したほうが，モデルとしても実装としても単純で扱いやすい．

第 4 章

高性能な低水準通信ライブラリの設計と実装 [119]

インターコネクトの機能を抽象化することは，システムソフトウェアの記述性と再利用性を向上するために重要である．本研究では，DSM や ULT といった高水準なライブラリの実装のために，並行して低水準通信ライブラリについても設計と実装を行ってきた．本稿では，低水準通信ライブラリに求められる機能と現実のハードウェアとの差分を，どのように低オーバーヘッドに埋め合わせていくべきかについて解説していく．

本研究で取り扱ったのは，Tofu [120] [121] と InfiniBand （2.2 節）という 2 種類の HPC 向けインターコネクトである．その他の HPC 向けインターコネクトとして，uGNI [122]，BlueGene/Q [123]，OmniPath [124] などが挙げられる．

4.1 低水準通信ライブラリの概要

“低水準通信 (low-level communication)” という名称自体はそれほど一般的ではないが，4.2 節で述べるようにそのようなライブラリは数多く存在している．それらに共通の発想は，ハードウェアベンダーの意向に振り回されずにシステムソフトウェアの実装の互換性を保ちたい，という考え方である．HPC 向けのインターコネクトは各ベンダーが多種多様な規格を提案しているが，新規システム開発には長期間を要するので，ハードウェアの更新毎に全体を再開発することは非現実的である．ハードウェアが違ってもインターコネクトの基本的な機能は概ね同じであり，インタフェースを統一するソフトウェア階層が存在すれば，システムソフトウェアの互換性を保つことができる．このような背景から，低水準通信システムの利用者は専らシステム開発者であり，アプリケーションから直接利用することは通常想定されない．

それぞれのインターコネクトシステムが全く同じ機能を有しているとすれば，単純に関数の引数を転送・変換するだけでよいはずだが，現実の通信システムに関してはそれほど単純ではない．2.3 節で述べたように，RMA や AM といった比較的単純なインターフェイスを一つ取っても，メモリレジストレーションや Eager limit といった問題が内在しており，理想的なインターフェイスと現実のハードウェア機能はどこかで乖離していることが分かる．また，複数のハードウェアに対応させる都合上，しばしば機能の不足したハードウェア向けにインターフェイスを合わせる必要がある．できるだけ低オーバーヘッドなラッパを実装するためには，それらのインターコネクトの微妙な差異についての理解が必要であり，必ずしも自明な解決策があるわけではない．

本章で主に掘り下げるのは，低水準通信システムのマルチスレッディングに関する問題である．インターコネクトの資源はノード内スレッドごとではなくノードごとに存在するので，それらのスレッド間では必然的に資源の共有が発生し，衝突の原因となる．ノード内に 1 スレッドしかないシングルスレッドの通信システムでは主に通信と計算のオーバーラップのみに注力すればよかったが，マルチスレッドでは資源の衝突という問題が常につきまとう．

2.1 節でみたように，LogGP モデルにおいてオーバーヘッドが存在することの意味は，ハードウェアに対する通信指示には一時的にであっても CPU のパワーを借りる必要がある，ということを示している．そして，オーバーヘッドはアプリケーション性能に影響しやすい指標であり，その削減は通信システムとして取り組む意義のある問題である．オーバーヘッドはスレッド間の衝突に影響されやすく，マルチスレッド対応とも根深く関係している．

近年になってマルチスレッド対応が重要視されるようになったのは，マルチコアプロセッサが市場の大半を占めるようになり，メニーコアプロセッサも登場するなど，計算ノード内の CPU 並列性が増大していることが背景にある．通信システムとしては，理想的にはノード内コア数に比例して 1 ノード当たりの通信資源も増加するべきであるが，現実のハードウェアはコア数よりも通信資源が少ないため，ノード内コアで通信資源を共有せざるを得ない．この際，コア間の排他制御に粗粒度のロックを用いるだけでは，通信発生毎に衝突が発生してノード内のスケーラビリティ低下の原因となる．このようなマルチスレッドに関する問題に対応するには，通信システムがそれを前提として実装されることが性能上不可欠になっている．

本章の提案手法となっているのが，**通信オフローディング (communication offloading)**，より詳細には**ソフトウェアオフローディング (software offloading)** の考え方である．ソフトウェアオフローディングとは，通信に必要な CPU 処理を通信専用コアに委譲する考え方である．最近のマルチコアプロセッサでは，コア数が数十にもなるものも珍しくなく，それらのうちいくつかを通信専用に割り当てる (dedicate) ことが性能上さほど問題にならない．ソフトウェアオフローディングは，（通信投入の）オーバーヘッドやコア間衝突を避け，さらには通信集約を可能にすることで，結果的に全体のアプリケーション実行時間を短縮する働きがある．一方で，通信の高速化のためにノード内の CPU 並列性を犠牲にしているという問題は残るので，通信性能とのトレードオフの関係にある．

本章では，ソフトウェアオフローディングを行う低水準通信ライブラリの実装について述べ，その性能をマイクロベンチマークに基づいて評価して考察し，今後の課題について探る．

4.2　低水準通信ライブラリの関連研究

4.2.1　既存の低水準通信システム

GASNet [22][125] は，PGAS 向けの低水準通信ライブラリとして広く使われていて，AM（2.3.2 項），RMA（2.3.3 項），集団通信の主に 3 つをサポートしている．GASNet の主な用途は PGAS の下位層であり，GASNet 自身も元々は Unified Parallel C (UPC) [32] のために開発された．UPC の他に GASNet を使用している処理系としては，UPC++ [34]，Chapel [40]，OpenSHMEM [38] の参照実装などが挙げられる．

GASNet の問題点として，RMA の一種であるリモートアトミック（他ノードのメモリをアトミックに書き換える機能）のような比較的新しいインターコネクトの機能がサポートされていないことが挙げられる．GASNet の登場時期は 2002 年頃であり，API は当時のハードウェア事情を反映したものとなっている．例として，秋山ら [81] の処理系ではノード間ワークスティーリングを実装するためにリモートアトミックが必要だったが，GASNet が対応していなかったために AM でのエミュレーションを行っている．この問題は GASNet のインターフェイスの不足が引き起こしており，ノード間ワークスティーリングの手法自身の問題ではないため，本来であれば低水準通信システムとして一般的に改善すべき問題である．

ARMCI [29] も PGAS 向けの低水準通信レイヤーの一つで，PGAS ライブラリである Global Arrays [35] のために開発された．ARMCI の機能は GASNet に類似しているが，AM はサポートされていない．最近になって，Global Arrays の低水準通信レイヤーは ARMCI から ComEx [126] というライブラリに置換された．ComEx は MPI との相互運用性を目指して実装されているが，未だ開発途上のため基本的な RMA の機能しか提供しておらず，ARMCI 同様に AM はサポートされない．

近年になって，低水準通信ライブラリとして新しく 2 つのライブラリが登場している．一つ目は Unified Communication X (UCX) [23] であり，GASNet と同様に AM，RMA，集団通信の 3 つを提供する他，MPI の実装に必要とされるタグマッチング機能を備えるなど，API としては網羅性が高い．二つ目は libfabric [30] で，InfiniBand ドライバを開発しているプロジェクトである OpenFabrics [127] のサブプロジェクトして運用されている．libfabric は InfiniBand Verbs (IBV) よりも抽象度の高い API を提案しているが，プロジェクトの性質上 API が IBV に強く影響されており，サポートされているのは RMA のみである．

MPI は基本的に低水準通信層よりも上位に位置する API だが，MPI の仕様は広範に渡っており，低水準通信 API として使うことも可能ではある．MPI-2 からは RMA が導入され，MPI-3 ではより RDMA に近いモデル*も登場し

* MPI-3 では，RMA の機能として Passive target というリモート側プロセスの CPU が明示的に介在しないモデルが導入された．これに対し

たため，この上で PGAS を構築することもできる [128]．MPI の仕様は広範に渡るが，PGAS の下位レイヤーとして必要な機能はその一部にすぎない．また，MPI-3 の片方向通信は RDMA だけで実装できない場合もあり，リモート CPU の介在を要求する場合もある [27] など，現実のインターコネクトからの乖離も見受けられる．

4.2.2　通信システムのマルチスレッド性能についての既存研究

前項で取り上げた既存の低水準通信レイヤーのいずれも，マルチスレッド実行に関する特別な配慮はなされていない．GASNet や ARMCI が開発された当時はマルチコアプロセッサが一般的でなかったという事情があるが，近年になって登場したライブラリですらマルチスレッド性能の重要性を認識しているものは少ない．

低水準通信ライブラリとしてマルチスレッド環境での性能向上に取り組んだ例として，PAMI [26] [129] がある．PAMI はソフトウェアオフローディングを実装しており，アプリケーションスレッドから委譲された通信処理は予め用意されたスレッドプールからどれかのスレッドが実行するという仕組みになっている．通信処理の移譲は，ノンブロッキングキューを介して行う．

本研究の基本的アイディアも，PAMI と同様にオフローディングをノンブロッキングキューによって行うものである．本研究の PAMI との相違点は，以下のように挙げられる．

- 複数インターコネクトへの対応．
- PAMI で使用されている wakeup signal のような特殊な CPU 命令を使用しない．
- PAMI はネットワーク FIFO キューが CPU ごとに専有できることを前提としているが，本研究ではそれらが共有されてスケジューリングされることを念頭に置いている．これは，コア数の増大に伴ってネットワーク資源が CPU より貴重となりうることが背景にある．
- ポーリングスレッドを分離している．PAMI のポーリング処理はアプリケーションスレッドで行われていたが，本研究の実装ではそれを別スレッドに分離し，その分離に際してスレッド間の同期についても高速な手法を提案している．

PAMI の実験結果においてマルチスレッド性能についての言及は僅かであり，ハードウェアの改良といった異なるトピックも含まれている．本研究での低水準通信ライブラリは，LogGP モデルをベースとし，各指標についてマイクロベンチマーク結果を分析している点でも異なっている．

近年になって，MPI 実装の開発者の間で通信とマルチスレッドの関係についての議論が盛んに行われている．そのきっかけの一つが Balaji ら [130][131] の研究で，MPI の実装の 1 つである MPICH [132] をマルチスレッド環境で実行すると，MPICH 内の粗粒度ロックによってスレッド数増加とともにメッセージレートが大幅に低下していくことを示した．このマルチスレッド性能の向上を目的としたのが，現在 MPI-4 の仕様策定中で議論されている MPI Endpoints [133] という機能である．これまでの MPI では 1 ランク 1 プロセスというのが原則であったが，エンドポイントを使うと 1 プロセス内にランクが複数あるかのように扱える．同一プロセス内にある別々のスレッドが別々のエンドポイントを持てるようにすることで，プロセス内の通信資源による衝突を避けることができる．

Vaidyanathan ら [134] は，MPI におけるソフトウェアオフローディングによって，並行性と非同期性の両方を向上させることを提案している．Vaidyanathan らのアイディアも PAMI と似ており，「コマンドキュー」と呼ばれるノンブロッキングな循環バッファに通信要求を溜めていき，通信専用スレッドがそれを処理する．このような手法によって，通信専用スレッドのみが実際の通信を発行しているため，通信資源をロックで排他制御する必要がなくなって並行性が向上する．また，通信を要求しているスレッドは，コマンドキューへの挿入さえ完了すれば，`MPI_Isend()` のようなノンブロッキング関数において即座に呼び出し元に復帰できるため，非同期性も向上しているといえる．

Amer ら [135] は，OS が提供するミューテックスが非公平であるために，それに依存した MPI のランタイムの性能が低下することを示した．その解決策として，First-In First-Out (FIFO) で調停されるロックの使用に加えて，通信を進行 (progress) させる処理のために優先度をつけたロック手法を提案している．

Lu ら [136] は，ユーザーレベルスレッド (ULT)（3.3 節）を用いて MPI の通信レイテンシを隠蔽する機構を提案し

て，MPI-2 までは Active target という，お互いのプロセスの介在を必要とする RMA インターフェイスしかなかった．本稿では RDMA での実装を念頭に置き，RMA というと暗黙のうちに Passive を指しているが，MPI の用語としては両方を含んでいる点に注意する必要がある．

```
using process_id_t = /*integer*/;
struct remote_address { size_t offset; /*...*/ };
struct local_address  { size_t offset; /*...*/ };
struct callback { void (*f)(void*); void* d; };
struct read_params {
    process_id_t   src_proc;
    remote_address src_raddr;
    local_address  dest_laddr;
    size_t         size_in_bytes;
    callback       on_complete;
};
bool try_read_async(const read_params&);
```

リスト 4.1: RDMA READ の API 関数

ている．ULT はコンテキストを高速に切り替えることができるので，ユーザのスレッド内で通信を待ち合わせる間に計算をオーバーラップさせることが容易になる．

PGAS システムである ACP [137][138] でも通信のマルチスレッド化に取り組んでおり，通信要求の管理には Vaidyanathan らと同様に循環バッファによる手法を採用している．ACP の Tofu 上の実装はソースコードが公開されていないが，設計上は本稿の Tofu 用実装と類似する点が多いため，4.7.1 項で比較を行う．

4.3　低水準通信ライブラリの設計

MPI や PGAS よりも低水準なレイヤーとしてオフローディングを実装することには，次のような利点がある．

- システムの直交性が向上し，並行性バグを減らせる．スレッドセーフなシステムを構築するにはシステムを細かいモジュールやオブジェクトに分割し，それらごとにスレッドセーフを満たすよう設計することが望ましい．
- 分散メモリ用の複数のシステムをマルチスレッド環境で両立させることが容易となる．
- Offloading する通信の単位がインターコネクトの API に近くなるため，それを活かしたチューニングが可能である．4.5.2 項で述べる InfiniBand での通信集約がその一例である．

提案処理系の機能としては，GASNet と同様に RMA，AM，集団通信の 3 つを実装した．但し，プロセスの起動/終了など一部の処理に MPI を使用しており，集団通信の処理も基本的に MPI に移譲される．PGAS にとって重要なのは RMA であるため，本稿では主に RMA の実装手法を解説していく．

4.3.1　低水準通信の API

提案システムでは，既存の低水準通信レイヤーの API を参考にしながら，Tofu や InfiniBand で良好な性能が得られるよう設計を行った．リスト 4.1 に，例として RDMA READ の API を示す．RDMA WRITE やリモートアトミックなどでもほとんど同様の API が提供され，それらも含めて全てスレッドセーフである．

提案する API の特徴的な点は次のように挙げられる．

- try という接頭辞は「この関数が**失敗しうる**」ことを示す．このような仕様は，後述する Offloading の実装を踏まえている．通信要求失敗の原因は，通信資源やコマンドキューのサイズが有限であるため，通信要求が溜まりすぎてその処理が追いつかなくなることである．
- async という接尾辞は「完了通知が**コールバック関数**によって実現される」ことを意味する．この関数が成功して復帰することはあくまで通信要求が生成されたことを意味するのみで，実際の完了通知はコールバック関数の実行をもって行われる．MPI や GASNet などの既存のライブラリでは，“リクエスト” や “ハンドル” などと呼ばれるオブジェクトに問い合わせることで通信完了を調べるものが多いが，ユーザにとっては問い合わせの手法が限定されているために自由度が低い．最も単純な完了通知手法はフラグをセットすることであるが，状況によっては条件変数による待ち合わせも考えられるため，より柔軟な仕組みが必要といえる．ポーリング

を実行したスレッドがコールバック関数を直接実行するモデルは柔軟でかつ軽量であるため，本提案ではこれによって完了通知を行うこととした．

- RDMA で読み書きされるアドレスは，**ポインタよりも大きな構造体**である．ユーザは `offset` メンバを増減することでアドレスを指定する．これは，多くのインターコネクトにおいて RDMA を実行する際，メモリレジストレーション時に割り当てられる ID が必要であるという事情が背景にある．API としてはポインタだけを授受し，内部では ID を逆引きするような実装では，逆引きによるオーバーヘッドが避けられない．
- リモートバッファだけでなく，ローカルバッファに対してもメモリレジストレーションを要求する．これに対し，既存処理系ではローカルバッファのレジストレーションを求めないものが多い．レジストレーションのコストは細粒度の転送時に問題となる [139] ため，予めレジストレーションした領域にコピーするといった高速化手法が広く用いられているが，そのようなコピーは上位層が予めレジストレーションしておけば不要であることもある．
- 実際の通信の実行順序と，コールバック関数の呼び出し順序について，通信要求時と異なっていてもよい．順序保証の導入は，通信のリオーダリングを行うインターコネクトにおける性能向上の機会を失わせ，ランタイムシステムのマルチスレッド化においても内部の実装上の制約となりかねないためである．

このように，既存処理系と比べてより低水準な API を提供し，各種インターコネクトを使う上での問題をシステムのプログラマにあえて明示することを意図している．一方で，インターコネクト間の差異が吸収されているため，下位の実装を入れ替えることで上位システムをほぼそのまま別環境で高速に実行できると期待される．また，通信要求の関数は全てスレッドセーフであり，上位層ではその層での排他制御のみに注力すればよい．

提案する API では，ユーザプログラム中で明示的に呼び出すポーリング関数は存在せず，通信要求時に内部でポーリングが行われるということもない．このことは，ポーリング処理が通信専用スレッドなどによって自動的に実行されることを意味する．ユーザが計算処理中に適切な間隔でポーリングを挟むことは困難であり，各種インターコネクトによってもその間隔は異なる．また，ポーリング中にも計算をオーバーラップできるため，並列性も低下しているといえる．一方で，ポーリング処理を別スレッドで実行する場合，完了通知時にコア間通信によるソフトウェアオーバーヘッドが発生する．ポーリングを同一スレッドで実行すればこれを削減できる可能性があるが，ポーリング時に使用する通信資源がコア間で共有されている場合は，ポーリングの結果が必ずしも同一スレッドの通信要求であるとはいえない点に注意する必要がある．

4.3.2　通信システムとユーザレベルスレッドとの連携

提案システムの元々の API 設計では，次のような動機から `try_*`という形で通信関数が失敗することを可能にしていた．

- オフローディングの実装に循環バッファを使った場合，バッファが満杯の場合に通信要求を投入できなくなる．通信要求関数が必ず成功するようにした場合，循環バッファが使えない間は待機せざるを得ないので，通信要求が投入できない場合に他のスレッドに切り替える必要がある．
- IBV の仕様では，`ibv_post_send()` は通信投入に失敗するような API になっている．

ULT を使用した場合，3.4.1 項で検討したようにコンテキストスイッチのコスト自体は十分小さく，通信待ちの数千サイクル程度であればオーバーラップすることに問題はない．一方，ULT を使用した場合，通信毎にスレッドを新規に生成するのでアドレス空間の枯渇と言った各種資源の問題があることには注意すべきである．

ユーザは「どの資源が満杯になっているのか」分からないという問題があるので，通信システムの内部に比べるとスイッチの方法に限りがある．この問題は，通信システムが自動的に複数の資源を切り替えるような場合に顕在化する．ユーザにとっては複数の通信資源があっても `yield` する以外の選択肢がないが，通信システムはどの資源を必要とするか管理しているので，「今必要となっている資源」を積極的に解放するよう他スレッドに対して要求することも可能である．設計段階においてはこうしたスケジューリングに関する問題が明らかでなかったので，ユーザに一任するという方針を立てていた．

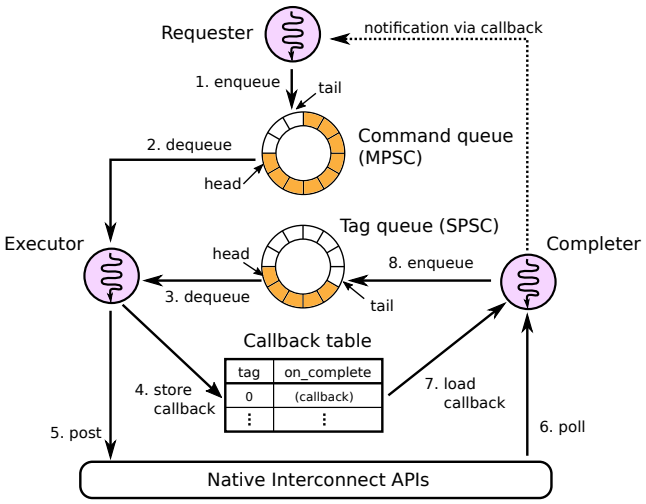


図 4.1: 提案システムの設計

4.4　低水準通信ライブラリの実装

図 4.1 に，提案システムの設計を示す．次節以降で以下の 3 つのモジュールに分けて解説を行う．

通信要求の生成 (Requester)
: 通信要求をコマンドキューに投入し，通信を移譲する．アプリケーションスレッドが通信関数を呼び出した時，Requester のコードに直接突入し，Executor に通信を移譲後即座に復帰する．

通信実行 (Executor)
: コマンドキューを監視し，コマンドがあればそれを取り出して実行し，インターコネクトに通信開始を指示する．

通信完了通知 (Completer)
: インターコネクトのポーリング関数を呼び出し，完了通知を取得した場合は対応するコールバック関数を実行する．

インターコネクトの事情によって異なるが，これらのモジュールはそれぞれ複数存在する場合があり，それらが別のスレッドとして並列動作可能な場合もある．

4.4.1　通信要求の生成 (Requester)

通信処理は通信専用スレッドに移譲されるが，この際用いられるのはノンブロッキングな循環バッファである．一般的に，Requester 側のスレッドは複数あり，Executor のスレッドは（キュー 1 つに対して）1 つなので，Multiple-Producer Single-Consumer (MPSC) であると仮定することができる．

リスト 4.2 に，本章の実装で用いるノンブロッキング MPSC 循環バッファのデータ構造を示す．要素の配列に加えて，整数カウンタ 2 つ（`head` と `tail`）と，Producer が追加した要素を Consumer から可視にするフラグ（`vis`）が必要となる．各コマンドを表す構造体には，コマンドの種類に応じたデータの他に，完了通知を管理するためのタグが付与される．

通信要求の移譲処理はリスト 4.3 のようなコードになる．Producer (= Requester) は Compare And Swap (CAS) によって `tail` を加算して伸長させることを試みる．その際，キューが満杯になっていないかを調べて，満杯であれば `false` を返して復帰する．CAS が成功した場合は，新規タグを確保してコールバックを書き込み，コマンドを設定して，最後にフラグ `vis` をセットして Consumer (Executor) に通知する．ABA 問題を避けるため，`tail` は `N` を超えて加算し続けておき，上位ビット列の違いで CAS を失敗させる．

```
constexpr size_t N_CMDS = /* ... */;
struct command { atomic<bool> vis; callback cb; /*...*/ };
command cmds[N_CMDS];
atomic<uint64_t> cmd_head, cmd_tail;

using tag_t = /*integer*/;
constexpr size_t N_TAGS = /* ... */;
tag_t tags[N_TAGS];
atomic<uint64_t> tag_head, tag_tail;

callback cbs[N_TAGS];
```

リスト 4.2: MPSC 循環バッファのデータ構造

```
void copy_params(const read_params&, command*);

bool try_read_async(const read_params& params) {
    uint64_t ct = cmd_tail.load(relaxed);
    do {
        if (ct - cmd_head.load(acquire) >= N_CMDS)
            return false;
    }
    while (!cmd_tail.compare_exchange_weak(ct, ct+1, acquire));

    command& cmd = cmds[ct % N];
    copy_params(params, &cmd);
    cmd.cb = params.on_complete;
    cmd.vis.store(true, release);
    return true;
}
```

リスト 4.3: Requester の実装例

通信要求関数が失敗できると規定されていることにより，通信システムの実装はキューが満杯になった際に `false` を返すことで速やかに呼び出し元関数に復帰できる．失敗通知を受け取ったユーザ側の関数には，次の 2 つの選択肢が与えられる．

- すぐさま再試行する．要求している通信処理のレイテンシを削減するのに有効である．この場合，要求が成功するまでスピンすることになるため，それを避けるために Pthreads の `sched_yield()` に相当する CPU の解放処理（以後，`yield()` と表記）を挟むことが可能だが，その場合はコンテキストスイッチを行うオーバーヘッドが加算されるために平均レイテンシは増加する．
- 別の計算/通信処理を実行した後で再試行する．要求している通信処理以外にも並行して実行可能な処理がある際に有効である．

どの手法が適しているかは低水準通信レイヤーのみでは判断不可能であるため，通信に関する最低限の処理のみを実行し，スケジューリングの方針は上位層に委ねるという方法を採用している．

タグ生成（`alloc_tag` 関数）には，使用中でないタグを区別できるフリーリストが必要となる．現時点の実装ではスレッドアンセーフな循環バッファをスピンロックで排他制御して使用しており，性能面で課題が残っている．解決策としては高速なアロケーション機構を実装する必要があり，各スレッドが独立した MPSC のキューを持つなどの手法が考えられる．4.5.1 項で述べる Tofu 上の実装では，Executor がタグ生成を行うことでロックの必要性を排除している．

```
void execute(tag_t, const command&);
// ...
uint64_t ch = cmd_head.load(relaxed);
uint64_t th = tag_head.load(relaxed);
while (true) {
    while (ch == cmd_tail.load(acquire)) { /*yield();*/ }
    command& cmd = cmds[ch % N_CMDS];
    while (!cmd.vis.load(acquire)) { /*yield();*/ }

    while (th == tag_tail.load(acquire)) { /*yield();*/ }

    tag_t tag = tags[th % N_TAGS];
    cbs[tag] = cmd.cb;
    execute(tag, cmd);

    cmd.vis.store(false, relaxed);
    cmd_head.store(++ch, release);
    tag_head.store(++th, release);
}
```

リスト 4.4: Executor の実装例

4.4.2　通信実行 (Executor)

Executor は，リスト 4.4 に示すようなコードでコマンドキューを監視する．Executor は新規コマンドの実行に際してタグの獲得も担当するので，コマンドキューとタグキューの両方に対して Consumer として働く．Executor は Producer による `cmd_tail` の変更を監視し，コマンドが存在していればそれを実行し，`cmd_head` を前進させる．Single-Consumer なので `head` の衝突を考慮する必要はない．

キューに大量の通信要求が存在する場合，この関数は通信要求を実行し続ける．問題となるのは通信要求が無い場合であり，キューが空の際にこのスレッドはスピンすることになる．レイテンシ削減にはスピンが有効だが，一方でスピンしているスレッドが多すぎると CPU の利用効率が低下してしまうという問題点がある．これは，インターコネクトが通信要求に必要な資源を複数持ち，Executor を並列実行できるような場合に特に問題となる．解決策の一つとしては `yield` を挟むことができるが，CPU の実行権限が回ってくるまで通信要求が遅延されるためにレイテンシの増大を招く．ノンブロッキングなキューではなくミューテックスと条件変数を用いたキューを使用することは可能だが，それらの同期プリミティブによるレイテンシ増加とスケーラビリティ低下が避けられない．

スピンを減らすための試みとして，ノンブロッキングキューと条件変数を組み合わせたキューについても検討した．通信要求がない場合は Consumer が条件変数によって待機し，その後最初の通信要求を行う Producer が Consumer に通知を行う．Consumer がコマンドを処理している間に Producer がさらなる通信要求を生成すると，ノンブロッキングキューに通信要求が追加されていく．このような仕組みにより，高負荷時の Requester 間の衝突を防ぎつつも，スピンするスレッドを減らすことができる．一方で，条件変数を使うことでレイテンシは増加することになる．

Consumer が条件変数で待機する場合は，コマンドが追加された際に Producer がそれに通知する必要があるが，高負荷時にはアトミック命令のみ動作させこの通知を削減することが性能上重要である．本稿の評価結果からは割愛するが，試験的な実装では `tail` の最下位 1 ビットを「待機中」であることを示すフラグとして利用することで，Producer が CAS を実行する際に Consumer に通知すべきか同時に判断できるようにしている．

4.4.3　通信完了通知 (Completer)

通信完了処理を行う Completer は，リスト 4.5 のようにインターコネクトに対してポーリングを行い，通知を取得できた場合はコールバック関数を実行し，そしてタグを解放して再利用できるようにする．

Completer に関しても，現在の実装では Executor と同様にスピンの問題を抱えている．InfiniBand の場合は，完了

```
bool poll(tag_t* tag_result);
// ...
uint64_t tt = tag_tail.load(relaxed);
while (true) {
    tag_t tag;
    while (!poll(&tag)) { /*yield();*/ }

    callback& cb = cbs[tag];
    (*cb.f)(cb.d);

    tags[tt] = tag;
    tag_tail.store(++tt, release);
}
```

リスト 4.5: Completer の実装例

通知をファイルディスクリプタに関連付けて `select()` などで待ち合わせることが可能である．但し，レイテンシは増大する点に注意する必要がある．

現在の実装では Completer は 1 つだけであるが，ポーリングが並列に実行可能ならば複数に増やすことも可能である．但し，Completer が並列動作する場合は，タグの解放を正しく排他制御する必要がある．

API としてコールバック関数は Requester とは別スレッドで実行される可能性があるため，スレッドセーフでなければならない．また，API 上はポーリングスレッドは 1 つであると仮定していないため，ポーリング処理も並列に実行されている可能性がある．例えば，通信処理をまとめて待ち合わせるにはカウンタを加算して一定値になるまで監視すればよいが，コールバック関数内ではフェッチアンドアッド命令などで排他制御する必要がある．

4.5　低水準通信ライブラリのインターコネクト毎の実装詳細

4.4 節における汎用的な実装手法を踏まえて，各種インターコネクトの仕様に起因する問題と，それらの解決策について述べる．

4.5.1　Tofu 用実装

Tofu の RDMA API には，次のような制約がある．

- 全ての API 関数がスレッドセーフではない．並列にでなければ複数スレッドから呼び出すことはできる．
- 複数の NIC が存在するが，異なる NIC を使用する場合でも API 関数を並列に呼び出すことはできない．
- 通信要求関数とポーリング関数も並列に実行できない．
- MPI と内部のデータ構造が共有されており，全ての RDMA 関数に加えて MPI も並列に実行できない．

以上のような事情から，必然的に Tofu 上の実装では前述した設計において逐次化される範囲が広くなる．具体的には，次のような実装上の配慮を行っている．

- 通信要求関数を実行する Executor と，ポーリングを行う Completer は，単一の通信専用スレッド上でのみ動作させる．これによって，通信要求やポーリングが並列に実行されることを防ぐ．
- MPI の呼び出しが必要になる場合は，RDMA と同様にコマンドキューに投入してから通信専用スレッドで MPI を呼び出す．

現在の Tofu 用実装では，タグの確保を Executor 内で行っており，図 4.1 で示した設計とは若干異なっている．その場合は Requester 間の排他制御を減らせるため，マルチスレッドのベンチマーク性能としては安定するが，Executor スレッドの負荷が増大しているといえる．タグの確保を Requester 内で高速に実行する方法については現在検討中である．

表 4.1: Tofu 用実装の評価環境 (FX10) [140]

CPU	SPARC64™ IXfx, 1.848 GHz, 16 cores/node
メモリ	32GB/node
インターコネクト	Tofu
OS	XTCOS (GNU/Linux 2.6.25.8)
コンパイラ	GCC 4.6.3 (with the option “-O3”)
MPI	Fujitsu MPI Library

4.5.2　InfiniBand 用実装

InfiniBand Verbs (IBV) は，InfiniBand を用いる際の標準的な API である．IBV のサービスタイプには複数あるが，本稿では Reliable Connection (RC) のみを仮定する．Tofu と異なり，IBV では API が全てスレッドセーフという違いがある．本稿に関係する範囲として，IBV は以下のような構成要素を持つ．

Queue Pair (QP)
: TCP/IP におけるソケットに相当し，通信要求が投入されるオブジェクトである．全対全で通信するには宛先のプロセスごとに QP が必要となる．実際の IBV の実装では，QP に対して通信要求を行う `ibv_post_send()` は QP ごとに独立したスピンロックで守られているため，QP が異なれば並列実行可能である．

Completion Queue (CQ)
: 通信完了通知が溜め込まれるオブジェクトであり，`ibv_poll_cq()` を呼び出すことで調べることができる．QP 同様に，CQ ごとに独立したスピンロックを持つ．

IBV はスレッドセーフな実装を提供しているが，内部ではスピンロックに頼っているのが実情である．スピンロックは通信レイテンシ削減には効果的であるものの，複数スレッドによって衝突が起きると 1 スレッドを除いて全スレッドがスピンするので，CPU の利用効率が低下する．通信レイテンシ隠蔽の観点からいって，IBV でも Offloading を用いる意義はあるといえる．

さらに，IBV で Offloading を用いることで，メッセージレート向上も期待できる．`ibv_post_send` には複数の通信要求（Work Request (WR) と呼ばれる）をリストにしてまとめて渡すことが可能である．`ibv_post_send` の実装では，所定のメモリ領域にアドレス情報などを書き込んだ後，doorbell と呼ばれる機構を用いてハードウェアに通信を要求する．この一連の動作を実行するのに少なくとも数百サイクルは消費するが，複数まとめて書き込むことでこのオーバーヘッドもまとめることができる．Offloading を行うと通信要求がコマンドキューに集約されるので，`ibv_post_send` を呼び出す際に複数のコマンドを一括して送り出すことが可能となる．

現在の実装では，（後述する図 4.8 の実験を除き）Executor スレッドは 1 つとしているため，複数の QP が存在していて並列に実行できる場合も逐次化され，並列性が Offloading なしの場合と比べて低下する問題がある．この点については，Executor の設計で述べたように条件変数を併用するといった対策を含め，いくつかの実装手法を比較検討中である．

4.6　低水準通信ライブラリの性能実験

表 4.1 に Tofu 用実装の評価環境を，表 4.2 に InfiniBand 用実装の評価環境を示す．

性能評価で測定するのは，LogGP モデルにおけるレイテンシ L，オーバーヘッド o，ギャップ g の 3 つの値である．スループット（あるいはバイトあたりギャップ G）は，メッセージサイズを十分大きく取った際のギャップから導出できる．図 4.8 の結果を除く全てのベンチマークにおいて，1 プロセス/ノードで 2 プロセス間の性能測定であり，そのうちの 1 つのプロセスに複数のスレッドを用意し，それらがもう片方のプロセスに RDMA READ を繰り返すという実験を行う．

表 4.2: InfiniBand 用実装の評価環境

CPU	Intel® Xeon® E5-2680 v2 2.80GHz, 2 sockets× 10 cores/node
メモリ	16GB/node
インターコネクト	Mellanox® Connect-IB® dual port InfiniBand FDR 2-port
ドライバ	Mellanox® OFED 2.4-1.0.4
OS	Red Hat® Enterprise Linux® Server release 6.5 (Santiago)
コンパイラ	GCC 4.4.7 (with the option “-O3”)

```
uint64_t get_clock(); // returns CPU clock
atomic<bool> flag;
void callback_func() {
    flag.store(true, memory_order_release);
}
// ...
while (/*...*/) {
    flag.store(false, relaxed);
    uint64_t t0 = get_clock();
    while (!try_read_async(/*...*/)) { /*yield();*/ }
    uint64_t t1 = get_clock();
    while (!flag.load(memory_order_acquire)) { /*yield();*/ }
    uint64_t t2 = get_clock();
}
```

リスト 4.6: レイテンシとオーバーヘッドを測定するマイクロベンチマーク．

```
atomic<uint64_t> count;
void callback_func() {
    count.fetch_add(1, release);
}
// ...
while (/*...*/) {
    while (!try_read_async(/*...*/)) { /*yield();*/ }
}
```

リスト 4.7: メッセージレートを測定するマイクロベンチマーク

リスト 4.6 に，レイテンシとオーバーヘッドを測定するマイクロベンチマークのプログラムを示す．各スレッドはまず通信要求を行ったのち，その終了をフラグを監視するスピンループで待機する．フラグはコールバック関数の中でセットされる．このベンチマークでは，レイテンシは $(t_2 - t_0)$，オーバーヘッドは $(t_1 - t_0)$ で与えられる．

リスト 4.7 に，メッセージレートを測定するマイクロベンチマークを示す．このベンチマークでは，各スレッドは常に通信要求をし続け，以前に発行した通信の終了を待つことなく常に発行を続ける．十分な時間が経った後，その時点で処理されていたメッセージの数を時間で割ることでメッセージレートが得られる．

4.7　低水準通信ライブラリのマイクロベンチマーク結果

評価結果におけるレイテンシの結果は，全て往復分で算出している．グラフ上の各点に対して 5 秒間の計測を 32 回繰り返し，レイテンシとオーバーヘッドは平均化された値になっている．エラーバーは，正規分布を仮定した 95% 信頼区間を意味している．

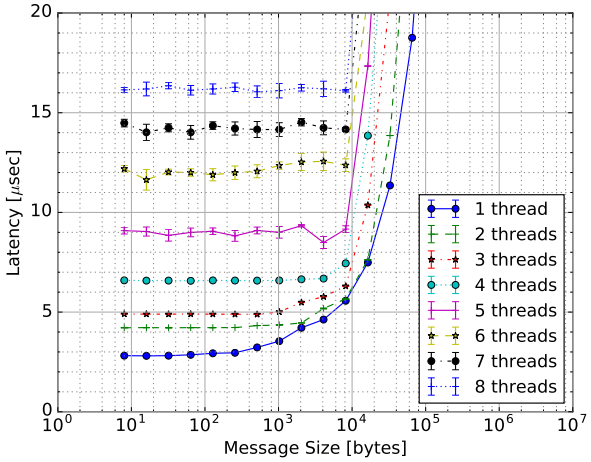


図 4.2: Tofu での RDMA READ の往復レイテンシ

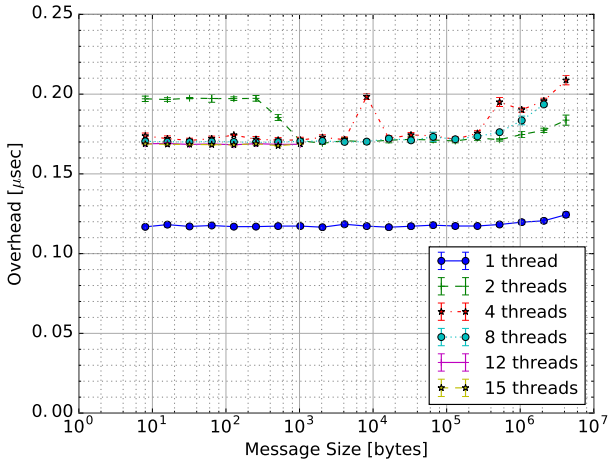


図 4.3: Tofu でのメッセージ投入のオーバーヘッドの測定結果

4.7.1　Tofu における実験結果

図 4.2 に，Tofu での RDMA READ の往復レイテンシの測定結果を示す．レイテンシが最小となるのは Requester スレッドが 1 つだけのときで，2.813μsec (5,200 サイクル) であった．提案システムとは別に，Tofu の RDMA API のレイテンシを独立して測定した結果は，2.365μsec (4,372 サイクル) であった．提案システムによって増大したレイテンシの差分は 0.448μsec であり，原因はソフトウェアオフローディングによるコストであろうと推察される．Tofu の RDMA API はスレッド安全性が一切ないが，提案システムではこれを保証していることから，完全に平等な比較ではないことに注意する必要がある．

Tofu での性能比較として，Tofu 2 上における ACP [141] のベンチマークでは，リモートアトミック命令の元々のレイテンシが 2.26μsec だったのに対し，ACP を使用した場合は 4.26μsec まで増大すると報告している．ACP のマルチスレッド対応には +88% ものレイテンシ増大が発生しており，それに比べると提案システムでは +19% の増加に留まっている．但し，この結果については，ACP は PGAS であるためにアドレス変換のフェーズを経ていること，Tofu ではなく Tofu 2 という世代の異なるハードウェアを用いていることに注意する必要がある．それらを踏まえても，提案システムではオフローディングが原因によるレイテンシ増大を既存研究と比べて十分抑えられているといえる．

図 4.3 に，Tofu でのメッセージ投入のオーバーヘッドの測定結果を示す．オーバーヘッドが最小になったのは 1 Requester スレッド時でメッセージサイズが 8 バイトのときで，0.117μsec (216 サイクル) であった．レイテンシを

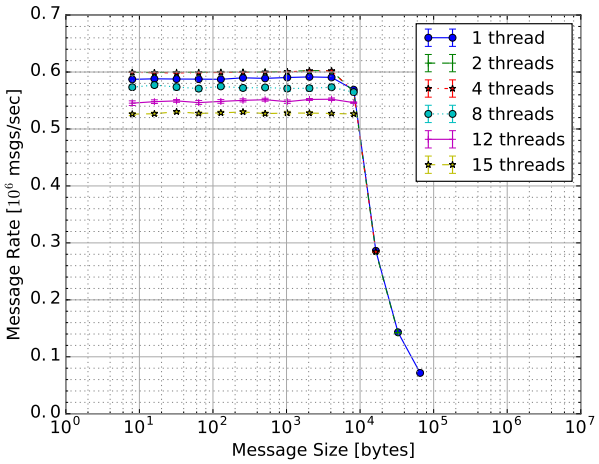


図 4.4: Tofu での RDMA READ のメッセージレートの測定結果

1 としたときのオーバーヘッドの割合は 4.19% であり，残りの 96.81% はオーバーラップ可能[†]な時間であるといえる．このオーバーヘッドは Requester スレッド数の増加に伴って増大するが，最大でも 2 Requester スレッド時に 0.196μsec (364 サイクル) に留まっている．

図 4.4 に，Tofu における RDMA READ のメッセージレートの測定結果を示す．最大のメッセージレートは 4 Requster スレッド時で，0.599×10^6 messages/sec となった．15 Requester スレッドで実行しても依然として 0.526×10^6 messages/sec を保っており，通信システムに高い負荷がかかっている状態でもメッセージレートを最大時の 88% に保つことができた．

Tofu における公称バンド幅は，1 リンクあたり 5GB/sec である．提案ライブラリでは 4 本のリンク全てを使っているが，メッセージレートから算出されるバンド幅は合計 4.700 GB/sec に留まっており，1 リンクあたり 1.175 GB/sec という結果になっている．追加の実験として，提案ライブラリで 1 リンクだけ使用するように変更したところ，同様の条件でバンド幅は 3.929 GB/sec となり，公式のベンチマーク結果 [142] とほぼ等しいことが分かった．これらの結果から，Tofu が持つリンク 4 本はお互いに独立して並列動作しておらず，1 本当たりのリンク速度は複数動作時に維持できない場合があることがわかった．

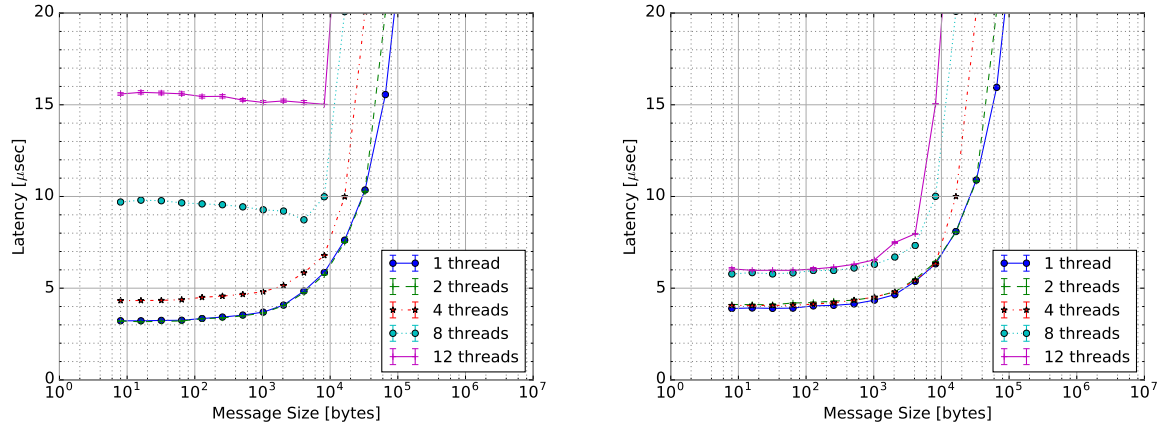
4.7.2　InfiniBand における実験結果

図 4.5 に，InfiniBand での RDMA READ の往復レイテンシの測定結果を示す．往復レイテンシの最小値はオフローディングなしでメッセージサイズが 8 バイトの時，3.222μsec (9024 サイクル) であった．オフローディングを有効にした場合，3.892μsec (10899 サイクル) まで増加している．Tofu における結果と同様に，レイテンシの増加分 0.670μsec は通信要求をキューに投入する際のオーバーヘッドであると考えられる．

提案システムとは別に，OpenFabrics [127] によって提供されている perftest というベンチマークのうちの `ib_read_lat` を用いて，RDMA READ のレイテンシを計測したところ，往復で 4.03 μsec (11288 サイクル) であった．標準のベンチマークよりも提案システムの方がレイテンシが短かった理由は不明だが，ポーリングを別スレッドで行っていることが影響している可能性がある．perftest のベンチマークでは，通信要求とポーリングの両方を単一のスレッドで実行している．

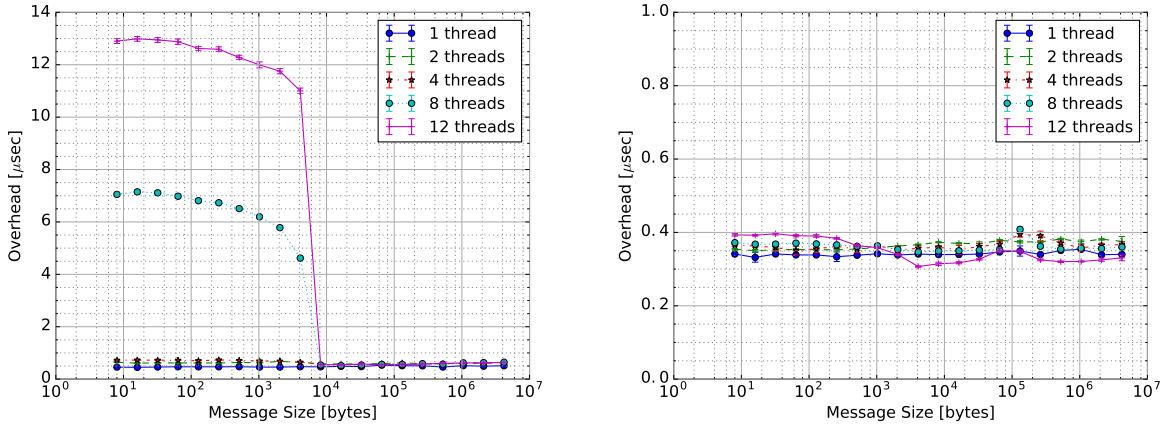
図 4.6 に，InfiniBand におけるオーバーヘッドの測定結果を示す．最小のオーバーヘッドは，オフローディングを行なわない場合の 0.327μsec (915 cycles) である．オフローディングなしだと，この値は Requester スレッドを増やした際に急激な上昇を始める．その原因は，`ibv_post_send` の中で用いられている spinlock であろうと推察される．オフローディングを有効にすると，オーバーヘッドは Requester スレッド数を増やしても一定であることが分かる．Tofu におけるオーバーヘッドの場合は単一 Requester スレッド時と比べて複数 Requester スレッド時にオーバーヘッ

[†] 通信完了を待ち合わせる時間を無視していることに注意する必要がある．



(a) オフローディングなし　(b) オフローディングあり

図 4.5: InfiniBand での RDMA READ の往復レイテンシの測定結果



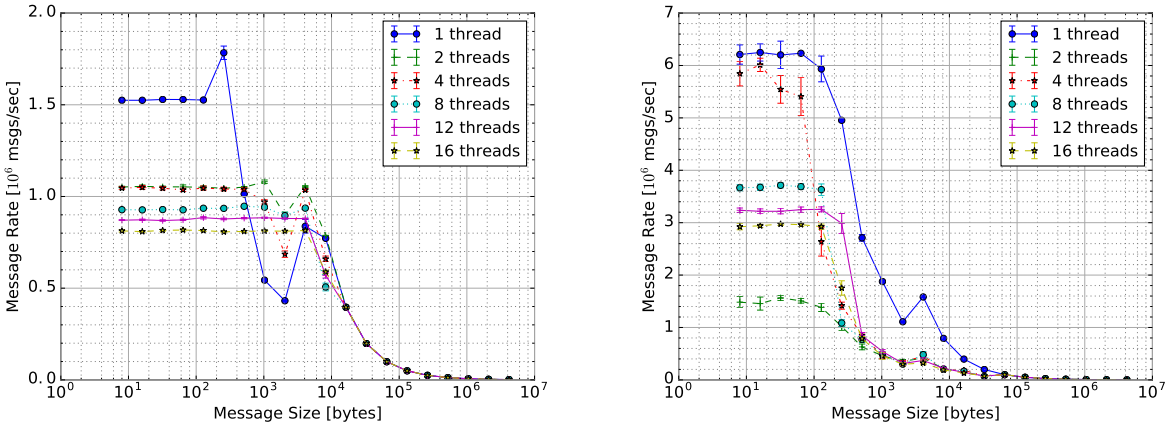
(a) オフローディングなし　(b) オフローディングあり

図 4.6: InfiniBand での RDMA READ のメッセージ投入によるオーバーヘッドの測定結果

ドが増大していたが，InfiniBand の場合はそのような現象もみられない．

図 4.7 に，InfiniBand におけるメッセージレートを示す．オフローディングなしの場合，最大のメッセージレートはメッセージサイズが 8 バイトでスレッド 1 つの時で，1.524×10^6 messages/sec であった．Requester スレッド数が 1 のままメッセージサイズを 2KiB まで増加させると一旦メッセージレートが上昇するがこの挙動については原因が分かっていない．それ以上メッセージサイズを増やすとメッセージレートは低下していくが，$10^3 - 10^4$ バイトの付近で極大となっていて，この結果についても原因は不明である．オフローディングを有効にした時，メッセージレートの最大値は 6.209×10^6 messages/sec まで改善しており，この値は UCX [23] のベンチマーク結果とほぼ等価である．UCX には “Accelerated Verbs driver” による実装が含まれていて，これを用いると 14×10^6 messages/sec までメッセージレートが改善すると報告されているが，本提案システムにはそのような機能には現在対応していない．IBV においてオフローディングを行った際のメッセージレートが向上する理由は，メッセージ集約によるものである．このメッセージレートは複数スレッド時に低下するが，16 スレッドでも依然として 3×10^6 messages/sec を保っている．2 スレッド時にメッセージレートが最低になる理由は現時点では分かっていない．実験時には NUMA の影響を考慮して，`numactl` を用いてコアの割り当てを変更することを試みたが，2 スレッド時の性能に改善はみられなかった．

図 4.8 に，InfiniBand において複数の Executor スレッドを用いた際のメッセージレートの測定結果を示す．このマイクロベンチマークでは，32 ノードを用いて各ノードに Requester スレッドをそれぞれ 1 つ用意し，それらが自ノードを除くランダムに選択されたノードに対して RDMA READ を繰り返す，という動作を行う．4.5.2 項で述べたよう



(a) オフローディングなし　　(b) オフローディングあり

図 4.7: InfiniBand での RDMA READ のメッセージレートの測定結果

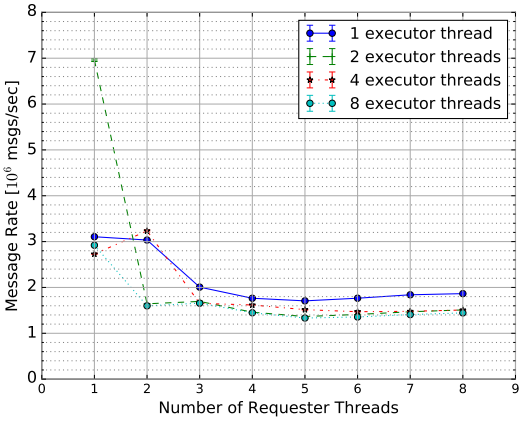


図 4.8: InfiniBand で複数の Executor スレッドを用いたときのメッセージレートの測定結果 (32 ノードで実験)

に，RDMA の各宛先ノードはそれぞれに QP を必要とし，QP ごとに独立のスピンロックを持っている．Requester スレッドが 1 つだけの際は，Executor スレッドを 2 つ用意した方が 1 つの場合よりメッセージレートが向上した．この向上幅は Executor の並列化によるものである．しかし，Requester スレッドを複数に増やした場合は，Executor スレッドが 1 つの場合のほうが概ね良好な性能を示しており，この原因も現在分かっていない．

4.8　低水準通信ライブラリについての要約

本章では，低水準通信ライブラリについて，マルチスレッド化を前提とした設計手法や実装について示した．提案システムでは，既存研究のようにオフローディングを実装するだけでなく，詳細なマイクロベンチマークを行うことでレイテンシ，オーバーヘッド，メッセージレートといった通信性能が改善することを示した．更に，試験的にオフローディング自体を並列化することを試みて，性能が向上する場合があることも示した．

低水準通信システムに関しての今後の課題は，以下のように挙げられる．

- スレッドスケジューラとの緊密な連携．
  - Active Messages のメッセージ処理や，複数の Executor スレッドの活用のため，スケジューラと連携することが必要である．
- 固定キューのサイズ問題と，そのより一般的な解決策．
  - 単に ULT で実装されたミューテックスに置き換えるのではなく，オフローディングに相当する機構が一

般的にスケジューラの一機能として解決できる可能性がある．

- よりハードウェアに近い層についての理解と，それを活用した低オーバーヘッドな実装手法．
  - InfiniBand に関しては，特に XRC [143] といった QP を削減する手法についても調査する．
  - スレッドや QP を増やした際の性能については不明な点が多く，マイクロベンチマークを増やして性能低下の原因を探る必要がある．

第 5 章

データ再配置とアドレスキャッシュのコンシステンシ [144]

5.1 データ再配置可能な PGAS の概要

2.3.4 項で述べたように，PGAS のメモリモデルはグローバル領域とローカル領域に分かれており，両者間の転送は明示的に get/put 関数を呼び出すことで行う．特にグローバルビュー PGAS においては，グローバル領域のデータ配置はシステムによって自動的に決定されるため，プログラマがその配置について細かく悩む必要がないという特長がある．具体的にどのようにデータ配置が行われるかというと，Unified Parallel C (UPC) に代表される多くの PGAS 処理系においては，グローバル領域のアドレスと配置の関係は静的かつ規則的なパターン（cyclic や block-cyclic など）に基づいている．そのような静的なパターンを用いることで，グローバルアドレスからローカルアドレスへの変換を単純化し，ランタイムオーバーヘッドを削減できる利点がある．一方で，アプリケーションが静的な配置パターンに適合しない限り，リモートアクセスが頻発することが避けられない．例えば，ノード間ワークスティーリング（3.5 節）を活用する場合，各ノードに動的にスレッドが割り当てられるので，PGAS の静的データ配置に対応させることはほぼ不可能である．スケジューリングを手動で行うにしても，Adaptive Mesh Refinement (AMR) [145] のようなアルゴリズムを静的データ配置に適合させることは容易ではない．そのような観点から，例として原ら [57] によって PGAS のデータ配置はより柔軟であるべきとの指摘がなされている．

PGAS 処理系において，アドレス変換のオーバーヘッドを許容できるのであれば，データ配置を柔軟にしたモデルを実装することが可能である．PGAS のグローバル領域のデータを動的に再配置することも可能であり，秋山ら [146] によって開発された MassiveThreads/GAS (MGAS) は実際に動的再配置を実装している．リスト 5.1 に示すように，MGAS は通常の PGAS と同様に get/put を実装するとともに，**own** という特殊なデータ再配置用の関数を有する．own 関数を呼び出すことで，グローバル領域の一部のメモリ領域を自ノードに動的再配置することができる．

図 5.1 に，MGAS の get 関数のレイテンシのマイクロベンチマーク結果を示す．実験は Tofu インターコネクト上で行っている．但し，このベンチマークに用いた処理系は筆者がマルチスレッド対応を試みたものであり，秋山らが実装した処理系に修正が加わっているが，基本的にはほぼ同一の実装である．レイテンシが最小となるブロックサイズ＝ 128 バイトの結果を見ると，プロセス内 (Intra-process) でのレイテンシは約 1000 サイクル程度で収まっているが，ノード内プロセス (Intra-node Inter-process) で約 97000 サイクル程度，ノード間プロセス (Inter-node) で約 130000 サイクル程度と，プロセス間通信には著しいレイテンシの増大が起きることが分かる．4.7.1 項で示したように，Tofu における RDMA のレイテンシは 4400 サイクル程度であるから，get 一回に占める RDMA 単体のレイテン

```
template <typename T> class global_ptr;
template <typename T> void get(global_ptr<T> gp, size_t size, T* lp);
template <typename T> void put(const T* lp, size_t size, global_ptr<T> gp);
template <typename T> void own(global_ptr<T> gp, size_t size);
```

リスト 5.1: MGAS の API

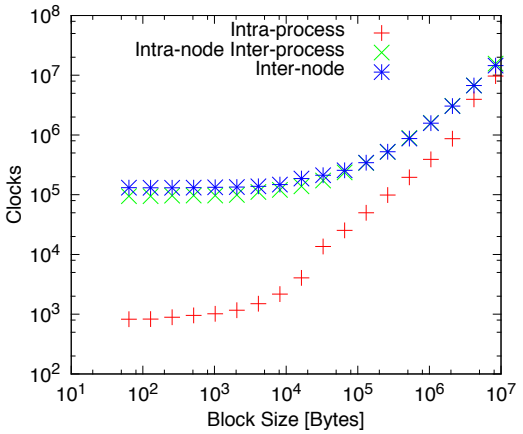


図 5.1: MGAS の get 関数のレイテンシのマイクロベンチマーク結果（1 ノードあたり 16 プロセス） [147]

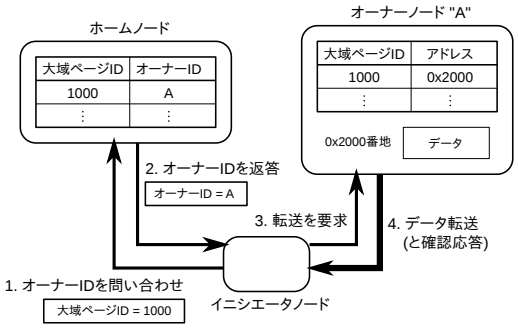


図 5.2: MGAS における get/put 関数の実装

シは 3.3% とごくわずかで，残りの全ては PGAS 由来のオーバーヘッドであるということになる．

図 5.2 に，MGAS における get/put 関数の実装を示す．MGAS のホームノードは，OS でいうページテーブルを保持しているノードである．データを要求するノード（ここではイニシエータノードと呼ばれている）がホームノードに問い合わせることでオーナーの情報を得て，それを元にオーナーノードに転送を要求する，という仕組みになっている．MGAS のレイテンシ問題の原因は，このように “データ再配置” を実装するためのディレクトリが集中管理されていることにある．get/put 毎にディレクトリの問い合わせを行うことで，**“get/put が RDMA にそのままマップできる”** という PGAS の特長が失われている．

RDMA を発行するにはリモートノード ID やリモートアドレスが必要（2.2 節）であり，これらの情報のことを本章では **メタデータ (metadata)** と呼ぶことにする．get/put を RDMA 1 回で済ませるには，get/put を行うノード上にメタデータが存在していなければならないことは明らかである．あるページに対して get/put を行うノードは複数存在しうるので，メタデータを複製できたほうが性能上望ましい（2.5.2 項）．すなわち，get/put を高速化するためには，メタデータのキャッシュは分散させる必要がある．一方で，own 関数というデータ再配置を認めることで，メタデータ自身も更新されうるキャッシュという性質を持ち，2.4 節で述べたコンシステンシの問題が出現することになる．

筆者が開発した MassiveThreads/GAS 2 (MGAS-2) [144] は，このアドレスキャッシュのコヒーレンスプロトコルを内蔵した PGAS である．その基本的なアイディアは，ディレクトリベースプロトコル（2.6 節）に基づいてアドレスキャッシュのコンシステンシを保つというものである．具体的には，ディレクトリを保持するノード（ホームノード）に加えて，他のノードがアドレスキャッシュを保持できるようにし，それら共有している全ノードを予めディレクトリに記録する．一方で own によってアドレス無効化が必要になった場合は Update 型のコヒーレンスプロトコルを用いてそれらのアドレスキャッシュを更新する．このようなプロトコルによって，アドレスキャッシュが行われていれば get/put を基本的に RDMA 1 回で実現する一方で，own によるデータ再配置も実現できる．

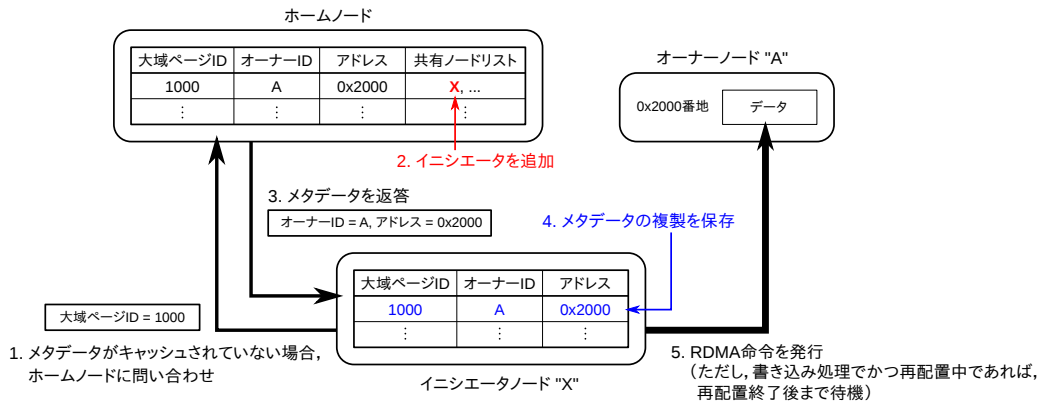


図 5.3: RPC に基づいて実装された再配置可能な PGAS 処理系の get/put 関数の処理

5.2　PGAS のアドレス変換と再配置に関する関連研究

Farreras ら [148] は，PGAS 言語である UPC の処理系（IBM XLUPC コンパイラ）での RDMA の活用について論じている．XLUPC の元々の実装ではメタデータの集中管理を採用していたため，get/put 関数のレイテンシが大きくなっていた．Farreras らは，メタデータキャッシュを導入することで get/put 関数を高速化している．Dalton ら [34] はその後改良された XLUPC のアドレス変換手法について論じており，仮想アドレスをエイリアシングさせることでアドレスを一致させ，メタデータの量を削減する手法を提案している．この手法は，3.5.2 項で述べたコールスタックのための仮想アドレスの削減手法と実質的に同じである．

Chavarría-Miranda ら [149] は，PGAS ライブラリである Global Arrays でのメタデータ管理におけるスケーラビリティについて論じている．Global Arrays では元々全ノードにメタデータをキャッシュしていたため，メモリ使用量の観点からスケーラブルでなかった．そこで，一部のノードをメタデータサーバとして運用し，さらに計算ノードにもメタデータキャッシュを配置することで，メタデータのメモリ使用量を削減しつつ RDMA も活用できる PGAS を提案している．

本章で提案する PGAS と似た機能を実装しているのが，Active Global Address Space (AGAS) [150] という PGAS 処理系である．AGAS はソフトウェア実装とハードウェア実装があり，ソフトウェア実装は本章の処理系と似ているが，アドレスキャッシュのコンシステンシについて掘り下げてはいない．AGAS や本章で提案する PGAS では，再配置をサポートするため，前述の PGAS におけるアドレス変換よりも複雑なプロトコルが要求される．通常の PGAS ではメタデータが不変であるので，メタデータキャッシュの導入も比較的容易である．

DSM，特にホームベース DSM という観点からみると，データ再配置とはホームの変更に相当する．ホームベース DSM においては，実際に書き込みを行うノードにホームを変更すれば，diff の適用が高速化されるという利点がある．ホームの移動についてはいくつかの先行研究 [151] [62] で述べられており，キャッシュへの書き込み性能を決定づける重要な機能の一つである．

5.3　データ再配置とアドレスキャッシュを両立する PGAS 処理系の実装

MGAS-2 の実装は 2 つ存在しており，RPC に基づいた初期の実装と，ディレクトリ操作も含めて RDMA で実装したものがある．評価に用いたのは RPC に基づいた実装の方である．

5.3.1　RPC に基づいた実装

本項では，RPC に基づいた実装について述べる．図 5.3 に，get/put 関数の処理の流れを示す．get/put 関数の動作は，イニシエータ上のメタデータキャッシュの有無によって異なる．メタデータがキャッシュされていれば，即座に RDMA 転送を開始することが可能である．メタデータがキャッシュされていない場合は，ホームへの問い合わせが

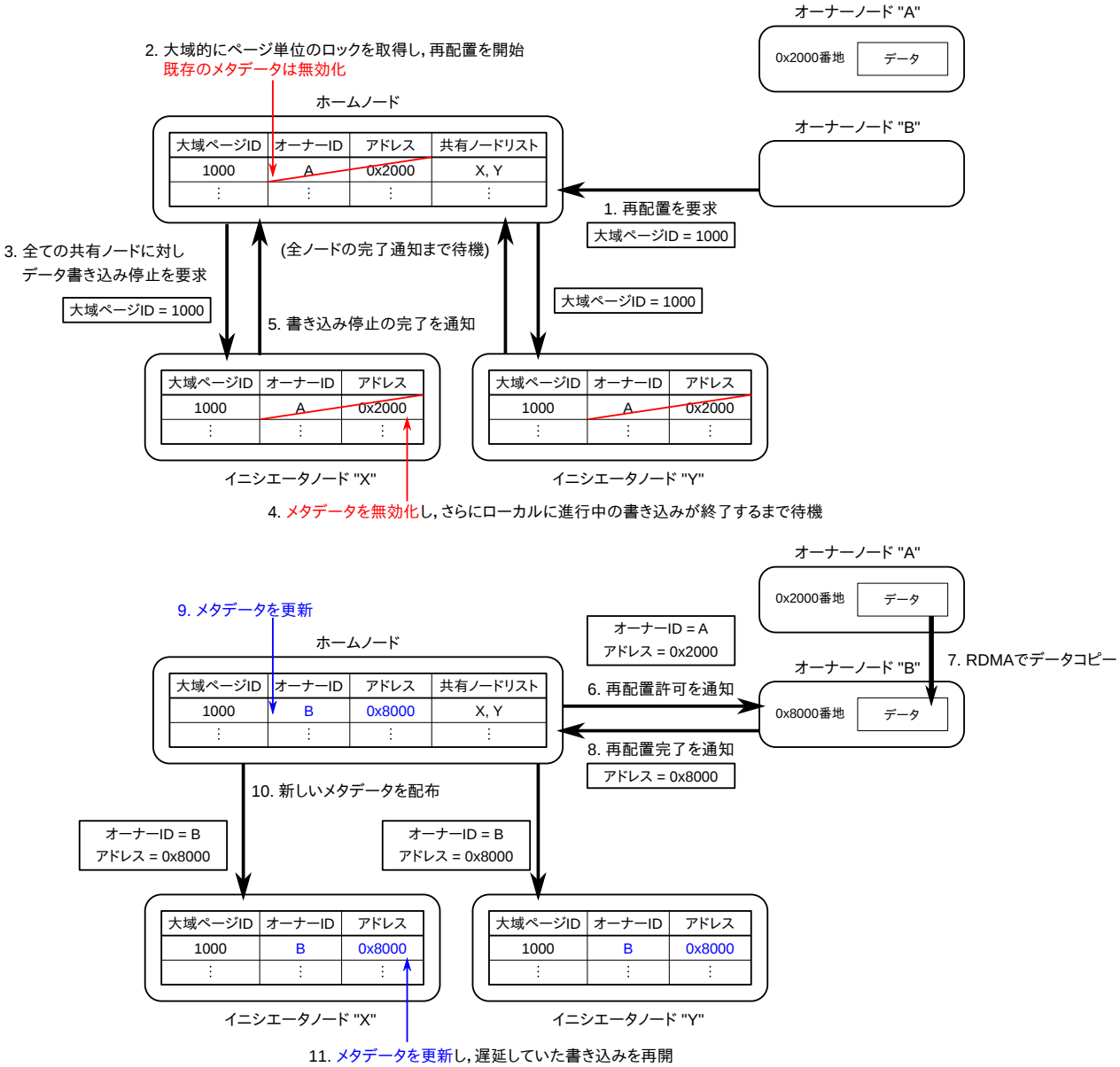


図 5.4: RPC に基づいて実装された再配置可能な PGAS 処理系の own 関数の処理

必要である．ホームは，この際にイニシエータを「共有ノード」としてディレクトリ上に追記する．これは，後述する再配置を行うためである．ホームからのメタデータの返答後に，イニシエータはメタデータのキャッシュを保存しつつ，RDMA 転送を開始する．

get/put 時にメタデータのキャッシュミスが発生した場合は，ホームからメタデータを取得することとなる．このため，最悪ではメタデータ取得分の 1 往復分のメッセージに加えて，RDMA 1 回分のレイテンシが発生する．メタデータのキャッシュは存在するが，データはリモートにある場合，RDMA 1 回分のレイテンシが発生する．最短なのはメタデータがキャッシュされていて，かつデータがローカルである場合で，ローカルコピーのみで終了する．オーナーとイニシエータが同じ場合は，メタデータがオーナー上にキャッシュされていれば，get/put 関数はノード間通信を必要としない．

図 5.4 に，own 関数の処理の流れを示す．再配置に関しては並列実行できないため，大域的なロックを管理する必要がある．提案手法では，ホームノード上でメタデータを無効化することで，再配置が複数スレッドから並列に実行されることを防ぐ．メタデータキャッシュ先はホームディレクトリ上で追跡されているため，全てのキャッシュ先ノードに対して RDMA によるデータ書き込みを中断するよう要求する．ホームが全てのキャッシュ先ノードから書き込み中断の完了通知を受け取れば，大域ページの状態は最新であり，再配置を開始できると判断する．再配置が完

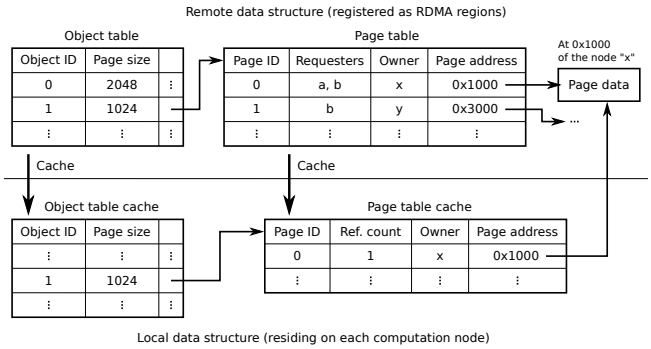


図 5.5: RDMA に基づいて実装された MGAS-2 のディレクトリ構造

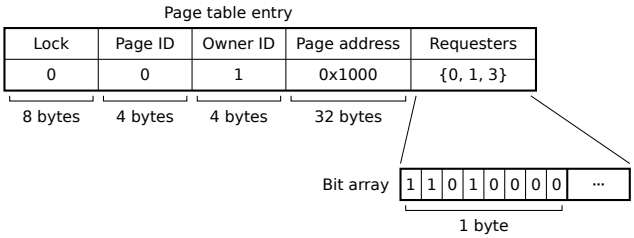


図 5.6: RDMA に基づいて実装された MGAS-2 のページテーブルエントリ

了したのち，ホームは新しいオーナー上のデータを示したメタデータを各キャッシュ先ノードに送信し，書き込み再開を指示する．この他，図 5.4 では省略しているが，全てのデータ読み込みが終了したことを確認して，元々のオーナー上のメモリを解放する機能も必要である．

再配置を実行している間，データへの書き込みを一時的に中断して，再配置が終了した後に再開する必要がある．一方で，データの読み込みだけであれば，再配置中であっても元のデータにアクセス可能である．ノード間でデータの読み書きの順序を保証する必要がある場合は，プログラマが明示的に排他制御を行う．

5.3.2　RDMA に基づいた実装

本項では RDMA に基づいた MGAS-2 の実装を示すが，この実装においては get/put 関数のみしか実装されておらず，own 関数については設計のみで実際には実装できなかったため，後の評価にも使用されていない．原因としては，RDMA によってディレクトリ管理を行うことが容易でなかったことが挙げられる．

図 5.5 にディレクトリ構造を示す．“ページ” は大域配列を構成するメモリブロックのことで，ページ内ではデータが必ず連続しているという保証があり，“オブジェクト” とはグローバルアドレス空間上で連続したページの集合である．本実装の MGAS-2 のディレクトリ構造は多段になっており，OS のページテーブルの構造に似せている．これによってオブジェクトごとにページサイズを分けたり，ページテーブルキャッシュの容量を変更したりなどの柔軟な運用が可能となる．ディレクトリは全プロセスで 1 つであり，これを保持しているノードを “ホーム” と呼ぶこととする．テーブルのキャッシュは全てのプロセス上で別々に保持されている．

図 5.6 に，ページテーブルエントリの詳細なデータ構造を示す．ページテーブルエントリにはそのエントリをキャッシュしているリクエスタのプロセス ID 一覧が記録されている．このようにエントリを共有しているプロセス ID の集合を記録する手法は，プロセッサのキャッシュコヒーレンスとの類似で見るとディレクトリベースキャッシュに相当する．プロセス ID の集合を管理するためにリモートアトミック命令を使用できるよう，通常の整数配列ではなくビット配列を利用している．整数配列を用いて要素の追加/削除を RDMA のみで実装することは不可能ではないが，RDMA の発行回数が増える傾向にある．一方，ビット配列ではフェッチ・アンド・アッド 1 回のみで集合要素の追加/削除が可能であるが，プロセス数を P とすると集合 1 つの使用メモリが $O(P)$ で増大するという問題点がある．

get/put が呼び出された際，まずローカルのキャッシュディレクトリを探索する．この際，対応するページのエント

リがキャッシュされていない場合，ページテーブルエントリをホームノードから取得する必要がある．ページテーブルエントリの取得は RDMA READ によって行うが，その前にエントリ中のリクエスタの一覧に自身のプロセス ID を記録する必要がある．

キャッシュエントリが見つかった場合，あるいはキャッシュミス後にエントリをキャッシュし終えた後は，キャッシュエントリの参照カウントを増加させる処理を行う．そして，キャッシュエントリ上のオーナー ID やアドレスに基づいて，get/put で要求されたデータ転送が RDMA によって実行される．RDMA 転送が終了次第，参照カウントを減少させてキャッシュエントリへの参照を解放し，get/put が終了する．

キャッシュエントリに参照カウントを導入する意図は，RDMA 転送中にキャッシュエントリが無効化されることを防ぐことである．具体的には，次の 2 つの場合にキャッシュエントリが無効化されうる．1 つ目は，get/put が複数スレッドで呼び出されたり，複数回ノンブロッキングで呼び出されたりした時に，キャッシュエントリの位置が衝突して置換しなければいけない場合である．2 つ目は，own が実行されてエントリそのものが全プロセス上で無効化される場合である．これらの場面でキャッシュを無効化する側のスレッドを適切に待機させるために参照カウントが用いられている．

実際の get/put の実行時には，転送したいデータが複数ページにまたがっている場合がある．複数ページを転送するには上記の処理の並行実行が性能上有効であるが，そのためにはページごとの転送状況を別々に管理するためのメモリ領域が必要となる．現在開発中の実装では，オブジェクトテーブルキャッシュを検索すると転送対象のオブジェクトのページサイズが分かるため，これを用いて転送されるページ数も計算でき，ページ数分に必要なメモリ領域をヒープから確保している．

get/put によるデータの RDMA 転送を実行している間，own によってデータ再配置に伴う RDMA 転送が開始されてしまうとデータの不整合が生じてしまうため，これは避けなければならない．一方で，get/put を高速化することは PGAS において最重要であり，再配置が実行されているかを毎回ホームに問い合わせるような手法では RDMA の利点を打ち消してしまうので，get/put の実行時にはホームへの問い合わせなしにキャッシュエントリが必ず有効であると保証することが性能上重要である．get/put によるキャッシュエントリの使用は own よりも優先されるため，own を実行する場合は他の get/put を実行し終えるまで待機することが必要となる．

own を実行している間のエントリはロックされた状態 (Lock=1) になっており，その間に RDMA で読み込まれたエントリは get/put 中で無効と判断され，ロックが解除されるまでエントリ取得が繰り返される．

own を実行するには own 同士が同時に実行されないように排他制御を行う必要があるので，ページテーブルエントリのロックを RDMA のコンペアアンドスワップ (CAS) で書き換える操作を行う．CAS が成功した場合は own 同士の同時実行が防げているものの，その時点では get/put を実行しているプロセスが他に存在する可能性があるため，即座にデータを再配置することはできない．そのため，CAS が成功した後，キャッシュエントリを持つ他のプロセスに対して RPC を送信し，それらのキャッシュを無効化する処理を行う．RPC を受け取ったプロセスは，対象のキャッシュエントリの参照カウントを監視し，0 になったらエントリを破棄して own を実行したプロセスに無効化が完了したことを返答する．全てのリクエスタがキャッシュエントリを無効化したことが確認でき次第，データ再配置のための RDMA READ が実行される．その後，ページテーブルエントリに新しいページのオーナー ID（own を実行したプロセス ID）とアドレスが書き込まれ，最後にロックを解除して own が終了する．

5.4　データ再配置可能な PGAS の簡易評価

本節では 5.3.1 項で述べた提案手法を実装し，簡易的な評価を行った．この評価を行ったのは 4 章の低水準通信ライブラリを実装する前であり，通信層については PGAS 処理系の一部として独自に実装したものを使用していた．評価に用いた環境は，低水準通信レイヤーのベンチマークと同様に表 4.1 の FX10 である．但し，コンパイラは C++11 のコードをコンパイルするために GCC 5.1.0 をビルドしたものを使用している．Tofu の RDMA には 4 つの NIC を選択可能であるが，この評価では NIC0 のみ使用している．

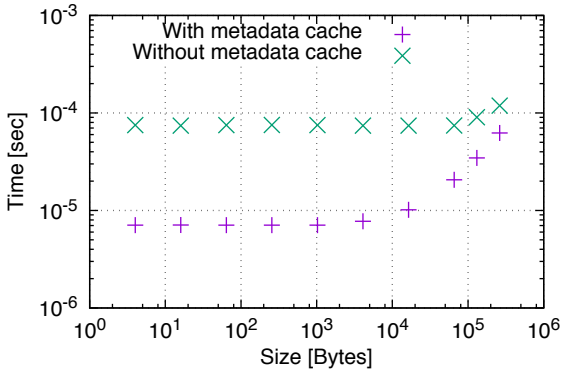


図 5.7: MGAS-2 の get 関数のレイテンシ

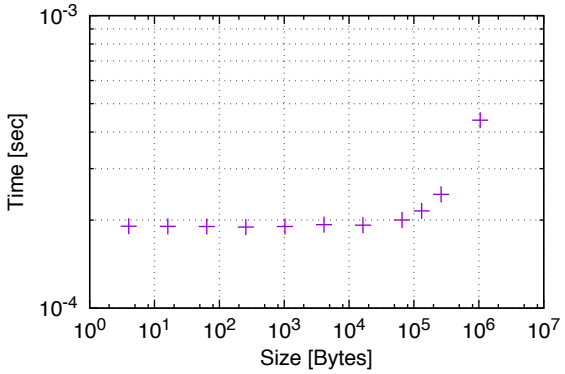


図 5.8: MGAS-2 の own 関数のレイテンシ

5.4.1　get 関数の評価結果

get 関数のレイテンシについては，ホーム，オーナー，リクエスタの 3 つをそれぞれ別々のノードとして測定を行っている．ページテーブルキャッシュを利用する場合と，キャッシュを利用せずに毎回破棄する場合で比較を行った．get 関数の呼び出しはグラフ上の各点ごとに 10000 回繰り返して算術平均を取っている．

図 5.7 に，get 関数でリモートの大域メモリからローカルにコピーする際のレイテンシを示す．メタデータキャッシュを利用している場合のレイテンシは最短で 7.1μsec である．4.7.1 項で述べたように，Tofu における RDMA READ の最小レイテンシは 4372 サイクル (= 2.36μsec) であるため，オーバーヘッドは約 4.74μsec となる．このオーバーヘッドの原因は，主に初期のプロトタイプの実装上の問題が大きい．

メタデータキャッシュを利用しない場合のレイテンシは 75.1μsec であり，メタデータキャッシュを利用した場合の 10 倍以上の時間がかかっている．この結果から，メタデータキャッシュが get のレイテンシ改善に有用であることが分かる．この差は初期のプロトタイプにおける RPC 処理が遅かったことが主な原因である．

5.4.2　own 関数の評価結果

own 関数の評価では，ホームとして 1 ノード，オーナーとして 2 ノードを使用する．オーナー間で own 関数の呼び出しを交互に繰り返すことで，同じページに対する own 関数のレイテンシを測定する．own 回数の呼び出しはグラフの各点ごと 1000 回繰り返して算術平均を取っている．

図 5.8 に，own 関数のレイテンシを示す．最小のレイテンシは約 190μsec となっており，エントリのキャッシュミスを起こした get よりもさらに 2.5 倍近い時間がかかっている．own はメッセージ往復回数が多く，現状では RDMA 化することも困難なため，低水準通信レイヤーにおける RPC を高速化することが性能上重要であるといえる．

5.5　データ再配置可能な PGAS の要約

本章では，データ再配置可能かつ RDMA を活用できる PGAS の設計を提案した．RDMA に必要なメタデータを共有することで，データアクセスでは RDMA を活用しながら，必要に応じてデータ再配置も行うことができる．評価においては，メタデータを使用した場合の方が，使用しない場合と比べて高速なデータアクセスが可能になることを示した．

第 6 章

ユーザレベルスレッドと協調する分散共有メモリ処理系の設計

2 章では，共有メモリモデル自体にはスケーラビリティの問題は存在しないこと，スケーラブルな分散共有メモリシステムが実現できる可能性があることを示した．既存研究の DSM 処理系で明らかになったスケーラビリティの問題を回避するには，以下に挙げられる設計判断が重要となる．

- 緩和型コンシステンシモデル（2.4 節）．
- MRMW 型コヒーレンスプロトコル（2.5.2 項）．
- 階層型コヒーレンスプロトコル．
- Self-invalidation の機構と，必要な時刻に正確に発行する予測器（2.6.1 項）．
- P/S3 分類によるコヒーレンス操作の削減（2.6.4 項）．

また，スケーラビリティに直接関与しない要素に関しては，実現可能性を加味して次のような手法を採用する．

- ページベース DSM（2.5.1 項）．
- ホームベース DSM（2.5.4 項）．

これらの基本的な設計方針は，既に Argo DSM [65] において試みられた方針である．本研究で実装する DSM について，Argo DSM との実装上の差分は以下のように挙げられる．

- モジュールとして分離され，高速化された低水準通信システム（4 章）．
- RMA にこだわらず，必要に応じて AM も使用する現実的な実装．
- メモリブロックの自動的な動的再配置（5 章）．
- OS ページサイズ以上での通信粒度の変更．
- ユーザレベルスレッド（3 章）との密な連携．

DSM システムが十分スケーラブルであり十分なアドレス空間を管理できるとすると，並列プログラムのある時点での全てのスレッドのコールスタックを DSM 上に配置することが可能である．本研究では，システムとしての簡潔性の観点から，DSM によってスタック領域とヒープ領域をまとめて管理することを最初の前提とした．従来研究のようにスタックかそれ以外かでコンシステンシモデルを複数用意することは，プログラマに複数のメモリモデルへの理解を強要し生産性が低い．既存研究の問題は，コールスタックの扱いを特別視しすぎることで，グローバルアドレッシングという共有メモリのセマンティクスを諦め，既存の共有メモリプログラムを動作させられなくなることであった（3.5.2 項）．

これまでの検討から，ユーザレベルスレッドと分散共有メモリ処理系を協調させることには次のような意義があるといえる．

- スレッドの依存関係に基づいたコヒーレンスプロトコル（2.4.4 項）．
  - DSM にとって，ULT に基づいて計算 DAG を把握できれば，緩和型コンシステンシに基づいてコヒーレ

ンス操作を高速化できる．

- ノード内でのスレッド切り替えによる，通信レイテンシの隠蔽．
- コールスタック（3.5 節）の自動的なコヒーレンス管理．
- 局所性を考慮したスケジューリングへの応用．

6.1　提案システムの API

本研究のユーザレベルスレッド処理系は，基本的な API は Pthreads と互換の API であり，MassiveThreads [104] の基本的なコンセプトを継承している．各スレッドには一意な “スレッド ID” が割り振られ，それらをプログラマが自由に管理できる．

本研究における ULT の API の特徴として，新規スレッドのスタック上に任意オブジェクトを配置できるよう，スレッド ID の確保の関数を分離している．これによって，新規スレッドにデータを渡す際に動的アロケーションを行う必要がなくなる．C++ においては，関数オブジェクトを *placement new* で構築することで，任意サイズの関数オブジェクトをオーバーヘッドなしでコピーして実行できる．この改良は，通常の共有メモリシステム上であっても DSM であっても同様に適用可能である．

```
struct ult_id;
struct allocated_ult {
    ult_id  id;
    void*   ptr;
};
allocated_ult allocate(size_t alignment, size_t size);
void fork(allocated_ult th, void (*func)(void*));
void join(ult_id id);
void detach(ult_id id);
void yield();
void exit();
```

DSM に関しては，まずコールスタックに関しては ULT が自動管理するので，DSM の API としては公開されない．DSM 上でのヒープ領域の確保に関しては，現在適切なインターフェイスを検討中であるが，malloc/free と同様のアロケーションを実装することは可能である．

```
void* malloc(size_t size);
void* aligned_alloc(size_t alignment, size_t size);
void free(void* ptr);
```

DSM 上のメモリ領域のコヒーレンスは ULT によって自動的に保たれるので，通常の場合プログラマは明示的にバリアを挿入することはない．

DSM と ULT の双方に関して，C や C++ の言語標準であったり，Pthreads のような OS によって提供される API であったりといった，標準的なインターフェイスを上書きする手法がある．例えば MassiveThreads では，Pthreads のインターフェイスを乗っ取ることで，既存のマルチスレッドプログラムを再コンパイルなしにユーザレベルスレッドで動作させることが可能である．DSM について同様の手法を適用する場合は，`malloc/free` といったメモリアロケーション関数が乗っ取りの対象になる．このような既存 API とのバイナリ互換性は，しばしば既存の別のライブラリ（例えば MPI）との互換性を失う原因となるため，議論の余地がある．本稿で述べるシステムでそのような手法が実現不可能であるというわけではないが，完全なバイナリ互換性については実用面からも優先度が低いため，本稿ではその実現手法については割愛する．

6.2　DSM と ULT を組み合わせたシステムの設計

6.2.1　プロトタイプの設計

本研究の DSM におけるプロセスの役割として，以下の 3 つの用語を導入する．

“マネージャ (manager)”
　あるページについて，ディレクトリを管理しているプロセス．
“オーナー (owner)”
　ホームベース DSM における “ホーム” であり，diff の適用先となるメモリブロックが配置されているプロセス．
“シェアラー (sharer)”
　キャッシュブロックの複製を持ち，実際にデータに読み書きを行っているプロセス．

概念的には全てのホームベース DSM がこうした区別を行っているが，他の研究では異なる名前になっている場合があることに注意する必要がある．特に，“ホーム” という用語はマネージャとオーナーの両方の意味で使われることがあるので混乱しやすいため，本研究の DSM においては使用しないこととした．マネージャとオーナーが常に同一であるような処理系の場合はこれらを区別する必要がなかったが，提案処理系ではデータ再配置を実装できるようにそれらを明確に区別している．

コヒーレンスのための通信が発生する際に，インデクスを調整して転送することが行えるのであれば，オーナー上のデータと各シェアラーのキャッシュは必ずしも同一サイズで区分けされる必要がない．そこで，オーナー上のデータ分割の単位を「ページ」，各プロセスのキャッシュの分割単位を「ブロック」と呼び，これらを区別することとした．これらの各用語は，TLB を活用した P/S 分類において，それぞれページとキャッシュラインに対応している．

さらに，「ページ」のサイズが異なる複数のメモリ領域を扱えるように，その上位概念として「セグメント」を導入した．「ページ」と「セグメント」という用語の対比関係は，仮想メモリシステムの一方式である「ページ化セグメンテーション方式」から借用したものである．これらの用語をまとめると，提案システムでのメモリは次のような階層で細分化されていく．

“スペース (space)”
　DSM 処理系が管理する空間全体のこと．
“セグメント (segment)”
　space 中の連続したアドレス空間で，ページの集合．
“ページ (page)”
　P/S3 分類，データ配置，再配置を行う基本単位．
“ブロック (block)”
　データ転送を行う単位．あるいは，メモリ保護を一括で変更する単位．
- マネージャはブロックの大きさに関与しない．書き込みのマージに diff を使っているため，このような実装でも問題は起きない．

6.2.2　スレッド依存関係とメモリバリア

現在の実装は，BACKER アルゴリズムという Distributed Cilk の基本的な実装方式に沿っている [7] [8]．提案システムの ULT によって呼び出されるバリアは，`release_all` と `acquire_all` であり，これらは Distributed Cilk での `reconcile` と `flush` に，Argo DSM [65] における self-downgrading フェンスと self-invalidation フェンスにそれぞれ対応している．

```
void release_all();
void acquire_all();
```

`release_all` は，自プロセスが書き込みを全て完了するまで待機するバリアである．`acquire_all` は，それが呼び出された後は全てのメモリブロックが最新のデータを読むことを保証する．あるプロセスが他プロセスにデータを送る場合，まず送信元のプロセスがデータを書き込んで `release_all` を呼び出し，それが完了するのを待ってから受信先で `acquire_all` を呼び出すと，確実にデータをやり取りすることができる．緩和型コンシステンシであるので，この 2 つのバリアがない状況ではいつデータが同期されるかが何ら保証されない．

DSM のメモリバリアはリモートプロセスとの通信を伴うので，通常のメモリバリアと同様に出来る限り実行しないほうがよい．ULT がメモリバリアを発行するのは，「リモートプロセスで動いている/動いていたスレッドとの同期が発生した場合」に限定される．具体的には，次のような状況下でメモリバリアを呼び出す必要がある．

- `release_all` を呼び出す状況
  - 自プロセス上の ready deque にあるスレッドが，他のプロセスから steal された時．
    - 他のプロセスからの steal が起きたということは，他のプロセスがそのスレッドを実行しようとしているということなので，スレッドが書き込んでいたデータも書き込む必要がある．
  - 既に exit したスレッドが，リモートプロセス上で動作しているスレッドから join される時．
    - join が起きた時，リモートプロセスに対して `release_all` を実行するよう明示的に問い合わせる必要がある．
- `acquire_all` を呼び出す状況
  - exit 時に自スレッドを join してブロックしていたスレッドが，元々リモートプロセス上で動いていた場合．
    - exit 時にはブロックしていたスレッドを再開するため．
  - リモートプロセス上で動いていた，既に終了したスレッドを join する時．
    - 既に終了していたスレッドの変更を取り込む必要がある．
    - まだ終了していない場合はブロックするので当てはまらない．

このように，いずれの場合もリモートプロセスが関係する際のみメモリバリアを実行する必要があることが分かる．逆に，リモートプロセス上のスレッドと関与しない限り，メモリバリアを発行する必要が一切なく，自プロセス上の書き込みはシステムが適宜書き戻せばよく，他プロセスの変更を取り込む必要も一切ない．これらの事実が，DAG Consistency に基づく DSM が有用であると根拠である．

スレッドの依存関係をより活用した手法としては，Cilk-3 において計算 DAG を遡っていくような実装方式 [152] が試みられている．2.5.3 項で述べたようにこのような Lazy 型プロトコルは全体のバンド幅を減少させる効果があるが，2.6.4 項で述べたように現代的なハードウェアでバンド幅の削減はあまり重要ではなく，オーナーへの書き出しを出来るだけ早期に行うような Eager 型プロトコルの方が現実的である．

6.2.3　自動データ再配置

5 章で述べたデータ再配置について，DSM の機能として実装することを検討した．PGAS におけるデータ再配置は，ホームベース DSM の文脈ではオーナーの再配置に相当する．

P/S3 分類を行うためにはある時点での共有プロセス数を記録しておく必要があり，その際に記録された情報を転用して再配置を行うべきかどうかの動的な判断に用いることができる．まず，明らかに共有されていないと分かっているメモリブロックは，再配置すべきであると見積もることができる．また，書き込みを行っているプロセスがないリードオンリーのブロックに関しても，再配置を行うことで他のプロセスの書き込みを遅延させるおそれはない．

データ再配置はそれを行っている最中にディレクトリ上のロックを取得するので，その間のデータアクセスが阻止される．オーバーヘッド削減の観点からはデータ再配置によってデータ書き込みが高速化されるという期待がある一方，スケーラビリティの観点からいってこれは望ましくないため，トレードオフが存在する．

6.2.4　DSM 上でのスタックフルコルーチンの切り替え

本章の提案システムの設計方針の一つが，DSM 上にコールスタックを配置することである．コルーチンの中断/再開といった問題は，DSM ではない通常の共有メモリ上のシステムと全く共通である．そのため，共有メモリでユーザレベルスレッドをデバッグした後，その実装をほとんどそのまま DSM 上に移植することができる．この事実は，本研究のシステム開発の負担削減に大きく貢献している．

一方で，コールスタックをページベース DSM 上に置くことによって特有の問題が生じる．具体的には，次のような場合に再帰的ロックが発生する．

1. ULT 上のあるスレッドが，通信用のロックを取る．
2. 通信関数内で関数を再帰的に呼び出して，スタックを伸長させる．
3. 伸長したスタック領域がキャッシュされておらず，リモートノードからの読み出しを開始する．
4. 通信関数を呼び出して，再びロックを取る．

この問題はプロトタイプ実装において問題となったため，解決策を検討した．

1. 通信関数を全てリエントラントにする．すなわち，全ての通信用ロックを再帰ロックに変える．
2. ユーザのスレッドから通信用ロックを一切取らない．
3. コールスタックを必ずキャッシュしておく．

1. の対策はあまり現実的ではない．標準ライブラリなどの内部実装には，排他ロックが隠されている場合があるからである．従って，この方式ではシグナルハンドラ中で呼び出せる関数が大幅に限定される．

2. の対策は，現時点ではユーザのスレッドも直接通信を実行するので採用できないが，fork によってプロセスを分けるといった対策を組み合わせた場合，実現できる可能性がある．

これらの事情から，3. の対策を現在の処理系では採用している．ある領域をコールスタックとして使っている間は，そのスタック領域がフォールトを起こしてシグナルハンドラを呼び出すことが確実にないことを保証すればよい．そこで，DSM に対してスタック領域を固定 (pin) / 固定解除 (unpin) する新たな関数を導入する．

```
void pin(void* ptr, size_t size);
void unpin(void* ptr, size_t size);
```

実際にこれらの関数を呼び出すのは ULT である．ULT はコールスタックを別のスレッドのものに移す際，まず移動先のコールスタックを “pin” する．そして，そのコールスタックに移動した後，元のコールスタックを “unpin” する．このような動作によって，実行中のコールスタックによって必ずページフォールトが起きないことを保証できる．

6.3　DSM の要約

現時点では，DSM で動作するスレッドライブラリを設計して実装を試みたものの，主にデッドロックが原因で動作しないという状態に陥っている．今後はバグを修正していくとともに，性能上のボトルネックを特定して優先的に取り組んでいく予定である．

第 7 章

結論

本研究では，高生産で高性能な並列分散プログラミングシステムを設計するために，共有メモリとスレッドを基本としたプログラミングモデルの可能性について述べた．そして，それを実現するための構成要素として，ノード間の通信性能を最大限引き出す低水準通信ライブラリと，PGAS 上のデータ再配置という 2 つの技術について，実際に実装して評価した結果を示した．さらに，現在実装中の DSM について，設計の概略を述べ，現時点での問題とその解決策についても示した．現時点では DSM とスレッドを組み合わせて動作させる段階まで至っていないため，まずは動作させるということを目標に実装を継続しており，バグ修正と各種改良を積み重ねることで DSM として実用的なレベルに高めていくことを目指している．

低水準通信ライブラリについては，実装面でも基本的機能の実装は達成しており，オフローディングやマルチスレッド化という切り口からインターコネクトの性能を引き出せる高速化手法について詳細に解説した．性能に重要なギャップとオーバーヘッドはオフローディングを用いることで緩和できるが，オフローディングに頼りすぎることは CPU 資源の減少につながるので，常にトレードオフが存在する．そして，低水準通信ライブラリですらスレッド処理系と不可分であるということが徐々に明らかになってきている．インターコネクトの通信待ちは I/O 待ちの一種であり，スレッド処理系によってこれが解決されるというのはシステムプログラミングの世界でごく自然な帰結である．通信要求が動的に出現することから，オフローディングの仕組み自体もやはりスレッド処理系によって動的にスケジューリングされる必要が出てくる．ここでもやはり，ユーザレベルスレッドのような汎用的スケジューリング手法の必要性が浮き彫りとなる．

低水準通信の性能のマイクロベンチマーク結果は詳細に得られているため，これを用いることで DSM や ULT のような高度なシステムを定量的モデルで評価できることが理想である．正しいモデルを得るには実装の詳細を要求するため，まずは定性的に妥当な実装を決定することが求められる．これまで議論してきた DSM の設計は，筆者の理解の範疇では概ねスケーラビリティの問題を回避できると考えられるので，通信プロトコルを厳密化してこうしたモデルに置き換えていくことも検討している．理論と実装の両面から DSM について検討していくことは，スケーラビリティの低下が実装上の問題なのか手法そのものの問題なのかを突き止める上で重要である．もちろん今後早期にオーバーヘッドが小さくスケーラビリティもよい実装が得られることが理想であるが，DSM 研究がこれだけ長きに渡っていてかつ成功していないことを踏まえると，それほど単純に全問題が解決するとは考えがたいというのが筆者の見解である．

メモリモデルとスレッドモデルが互いに深く関係しているということも，本研究を進める中で明確になってきた．例えば，仮想アドレス空間の使用量削減に関しては，DSM と ULT の双方で取り組まれている問題で，それらの多くにエイリアシングによる解決策が盛り込まれている．グローバルアドレス性の放棄は共有メモリモデルの実現の放棄も意味するので，単なる実装の問題として楽観視してはならない．アドレス空間が潤沢に使えるという仮定自体は実装できなくなる可能性があるので危険だが，システムが本来実現すべきプログログラミングモデルを放置して実装上の問題ばかり取り上げることはシステム全体の方向性を歪ませることにつながりかねない．

筆者は学部 4 年から計 3 年間，PGAS や DSM といった分散システム上のメモリシステムの開発に取り組んできたが，現実のアプリケーションで評価したり，実際に利用者に使ってもらったりといった段階に至らなかったことに忸怩たる思いがある．その根本的な原因として，「アプリケーションを最低限動作させる」という段階に至る前にシステムの高速化手法について取り組んだこと，すなわち「早すぎる最適化」に取り組んだことが大きいと筆者は考えてい

る．MGAS-2 の RDMA 化を試みたことはその最たる例であったが，低水準通信システムのマルチスレッド対応も，PGAS のアドレスキャッシュも，「実装しなければそもそも動作しない」といった深刻な問題ではないという点では共通している．まずはシステムのインターフェイスを確定させ，それに基づいて著名なベンチマークセットを試すために細かい性能問題を無視してプロトタイプシステムを実装し，その評価に基づいて次の設計を練る，というソフトウェア工学でいうアジャイル開発的な手法が，研究分野においても重要なのではないかというのが筆者の現在の考えである．一方で，「インターフェイスを確定させる」というフェーズがそもそも困難であり，「共有メモリとスレッドだけあればよい」という基本的思想に至るまでに紆余曲折を経ているため，その間に「低レイヤーなシステムから確実にしていく」というボトムアップな方針が誤りだったとまではいえない．通信性能特性についての正しい理解は，見た目上のプログラミングモデルの違いばかりが先行する定性的議論ではなく，現実の並列計算機についての定量的な性能評価を行うことを可能にする．例えプログラミングモデルを修正しなければならないとしても，ソフトウェア階層の多くが再利用可能になっているということがボトムアップに作り上げていくことの利点であり，今後の DSM 開発にとっての第一歩としての役割を果たしている．

謝辞

指導教員の田浦 健次朗教授には日々大変手厚いご指導を頂き，心より感謝しております．本研究の基本的なアイディアも，元々は先生が発案したものが多くあります．来年度からもそのままご指導賜ることになるかと思いますが，改めまして宜しくお願い致します．

B4 から M1 にかけてお世話になった秋山 茂樹先輩にも，非常に熱心なご指導を頂きました．秋山さんの研究があったからこそ，私の研究の方向性が次第に定まったという経緯があるので，大変感謝しております．

M1 のころのポスドクの佐藤 重幸さんには，私の研究について様々なアドバイスを頂いた他，学生以外の方に気軽にお話できたことが有難かったです．私が博士課程に進む決意ができたのも，佐藤さんに相談できたからこそだと思っています．

計算機クラスタを使用させていただいた東京大学情報基盤センターとそのスタッフの方々に感謝致します．特に，同センターの塙 敏博准教授には，本研究について貴重なアドバイスを下さいましたこと感謝致します．

研究室の同期の岩崎 慎太郎君は，研究についてもそれ以外についてもいつも親身に話し相手になってくれるので有難く思っています．同じく同期のナムスライジャブ ビャンバジャブ君とも楽しく会話できて，良い同期に恵まれたと思います．中島 潤先輩には時々研究の細かい話を詳しく教えて頂き，大変勉強になりました．ついーとん君は定期的に絡んでくれて楽しかったです．来年度からも wheel がんばってください．利光君も主に横から絡んでくれたので楽しかったです．バスケがんばってください．とくさんとはお互い声が通らないので意思疎通が難しいのですが，時々オタク話をして盛り上がりました．フィンさんは突然研究室に登場するのでいつも驚いていました．今度よかったらどこにいるのか教えてください．秘書の高野 弘美さんには，私が暗い顔をしている時にすぐに相談に乗ってもらえたので，おかげで何とか立ち直ることができました．研究室の他の皆さんにも，和気あいあいと過ごせたことを感謝しています．

図目次

表目次

参考文献

[1] MPI Forum, : MPI: A Message-Passing Interface Standard Version 3.1 (2015).

[2] OpenMP Architecture Review Board, : OpenMP Application Programming Interface, Technical report (2015).

[3] Pacheco, P. S.: MPI 並列プログラミング, 培風館 (1997).

[4] Protic, J., Tomasevic, M. and Milutinovic, V.: Distributed Shared memory: Concepts and Systems, *IEEE Parallel and Distributed Technology*, Vol. 4, No. 2, pp. 63–77 (1996).

[5] Culler, D. E., Gupta, A. and Singh, J. P.: *Parallel Computer Architecture: A Hardware/Software Approach* (1999).

[6] Zhang, J., Behzad, B. and Snir, M.: Optimizing the Barnes-Hut algorithm in UPC, *SC '11: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11 (2011).

[7] Blumofe, R., Frigo, M., Joerg, C., Leiserson, C. and Randall, K.: Dag-Consistent Distributed Shared Memory, in *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pp. 132–141 (1996).

[8] Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E. and Randall, K. H.: An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms, in *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pp. 297–308 (1996).

[9] Frigo, M. and Luchangco, V.: Computation-centric memory models, in *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pp. 240–249, New York, New York, USA (1998), ACM Press.

[10] Fortune, S. and Wyllie, J.: Parallelism in random access machines, *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114–118 (1978).

[11] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R. and Eicken, von T.: LogP: Towards a Realistic Model of Parallel Computation, *PPoPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Vol. 28, No. 7, pp. 1–12 (1993).

[12] Alexandrov, A., Ionescu, M. F., Schauser, K. E. and Scheiman, C.: LogGP: Incorporating Long messages into the LogP Model, *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pp. 95–105 (1995).

[13] Martin, R. P., Vahdat, A. M., Culler, D. E. and Anderson, T. E.: Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture, *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, Vol. 25, No. 2, pp. 85–97 (1997).

[14] InfiniBand Trade Association, : Infiniband Architecture Specification (2007).

[15] Mellanox Technologies, : RDMA Aware Networks Programming User Manual rev 1.7 (2015).

[16] Taura, K.: 連載講座: 高生産並列言語を使いこなす (1), スーパーコンピューティングニュース (2012).

[17] Ino, F., Fujimoto, N. and Hagihara, K.: LogGPS: A Parallel Computational Model for synchronization analysis, *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, Vol. 36, No. 7, pp. 133–142 (2001).

[18] Ino, F., Fujimoto, N., Hagihara, K. and Adi, P.: LogGPS : メッセージ通信プロトコルの切替えを考慮した高水準通信ライブラリ向けの並列計算モデル, 情報処理学会論文誌. ハイパフォーマンスコンピューティングシステム, Vol. 42, No. 3, pp. 145–157 (2001).

[19] Hoefler, T., Schneider, T. and Lumsdaine, A.: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation, in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High*

Performance Computing, Networking, Storage and Analysis, pp. 1–11, IEEE (2010).

[20] Communicator Info Assertion: No Wildcards #461, https://github.com/mpi-forum/mpi-forum-historic/issues/461.

[21] Eicken, von T., Culler, D. E., Goldstein, S. C. and Schauser, K. E.: Active Messages: A Mechanism for Integrated Communication and Computation, in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pp. 256–266 (1992).

[22] Bonachea, D.: GASNet Specification Version 1.8, Technical report, EECS Department, University of California, Berkeley (2006).

[23] Shamis, P., Venkata, M. G., Lopez, M. G., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G. and Bouteiller, A.: UCX: An Open Source Framework for HPC Network APIs and Beyond, in *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 40–43, IEEE (2015).

[24] Zheng, Y., Kamil, A., Driscoll, M. B., Shan, H. and Yelick, K.: UPC++: A PGAS Extension for C++, *IPDPS '14: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114 (2014).

[25] Willcock, J. J., Hoefler, T., Edmonds, N. G. and Lumsdaine, A.: AM++: A Generalized Active Message Framework, *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, p. 401 (2010).

[26] Kumar, S., Mamidala, A. R., Faraj, D. A., Smith, B., Blocksome, M., Cernohous, B., Miller, D., Parker, J., Ratterman, J., Heidelberger, P., Chen, D. and Steinmacher-Burrow, B.: PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer, *IPDPS '12: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 763–773 (2012).

[27] Si, M., Pena, A. J., Hammond, J., Balaji, P., Takagi, M. and Ishikawa, Y.: Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures, *IPDPS '15: IEEE International Parallel and Distributed Processing Symposium*, pp. 665–676 (2015).

[28] Belli, R. and Hoefler, T.: Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization, in *IPDPS '15: IEEE International Parallel and Distributed Processing Symposium*, pp. 871–881 (2015).

[29] Nieplocha, J., Tipparaju, V., Krishnan, M. and Panda, D. K.: High Performance Remote Memory Access Communication: The Armci Approach, *International Journal of High Performance Computing Applications*, Vol. 20, No. 6, pp. 233–253 (2006).

[30] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H. and Squyres, J. M.: A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency, in *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 34–39 (2015).

[31] Hammond, J. R., Dinan, J., Balaji, P., Kabadshow, I., Potluri, S. and Tipparaju, V.: OSPRI: An Optimized One-Sided Communication Runtime for Leadership-Class Machines, *PGAS '12: The 6th Conference on Partitioned Global Address Space Programming Models* (2011).

[32] El-Ghazawi, T. and Cantonnet, F.: UPC Performance and Potential: A NPB Experimental Study, in *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, p. 26 (2002).

[33] Kuchera, W. and Wallace, C.: The UPC Memory Model: Problems and Prospects, *Ipdps* (2004).

[34] Dalton, B., Tanase, G., Alvanos, M., Almási, G. and Tiotto, E.: Memory Management Techniques for Exploiting RDMA in PGAS Languages, in *LCPC '14: Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing*, pp. 193–207 (2015).

[35] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Aprà, E.: Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit, *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231 (2006).

[36] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S. and Oskin, M.: Grappa : A Latency-Tolerant Runtime

for Large-Scale Irregular Applications, Technical report (2014).

[37] Numrich, R. W. and Reid, J.: Co-array Fortran for parallel programming, *ACM SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1–31 (1998).

[38] Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C. and Smith, L.: Introducing OpenSHMEM, in *PGAS '10: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, No. c, pp. 1–3, New York, New York, USA (2010), ACM Press.

[39] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Praun, von C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Vol. 40, pp. 519–538 (2005).

[40] Chamberlain, B., Callahan, D. and Zima, H.: Parallel Programmability and the Chapel Language, *International Journal of High Performance Computing Applications*, Vol. 21, pp. 291–312 (2007).

[41] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J.: The directory-based cache coherence protocol for the DASH multiprocessor, in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pp. 148–159, IEEE Comput. Soc. Press (1990).

[42] Keleher, P., Cox, A. L., Dwarkadas, S. and Zwaenepoel, W.: TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems, in *WTEC '94: Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 115–132 (1994).

[43] Adve, S. V. and Hill, M. D.: Weak ordering - a new definition, *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, Vol. 18, No. 3, pp. 2–14 (1990).

[44] Batty, M., Owens, S., Sarkar, S., Sewell, P. and Weber, T.: Mathematizing C++ Concurrency, in *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p. 55 (2011).

[45] Boehm, H.-J. and Adve, S. V.: Foundations of the C++ concurrency memory model, *ACM SIGPLAN Notices*, Vol. 43, No. 6, p. 68 (2008).

[46] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers*, Vol. 28, No. 9, pp. 690–691 (1979).

[47] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z. and Myreen, M. O.: x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors, *Communications of the ACM*, Vol. 53, No. 7, p. 89 (2010).

[48] Keleher, P., Cox, A. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, in *ISCA '92: Proceedings the 19th Annual International Symposium on Computer Architecture*, pp. 13–21, IEEE (1992).

[49] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors, in *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26 (1990).

[50] Peng, L., Wong, W., Feng, M. and Yuen, C.: SilkRoad: a multithreaded runtime system with software distributed shared memory for SMP clusters, in *CLUSTER '00: Proceedings of IEEE International Conference on Cluster Computing*, pp. 243–249 (2000).

[51] Peng, L., Wong, W. F. and Yuen, C. K.: SilkRoad II: Mixed paradigm cluster computing with RC_dag consistency, *Parallel Computing*, Vol. 29, pp. 1091–1115 (2003).

[52] Cheong, H. and Veidenbaum, A. V.: Compiler-Directed Cache Management in Multiprocessors, *Computer*, Vol. 23, No. 6, pp. 39–47 (1990).

[53] Bershad, B. N., Zekauskas, M. J. and Sawdon, W. A.: The Midway distributed shared memory system, Technical report (1993).

[54] Li, K.: IVY: a shared virtual memory system for parallel computing, in *ICPP '88: Proceedings of the 1988 International Conference on Parallel Processing*, pp. 94–101 (1988).

[55] Iftode, L.: Scope Consistency: A Bridge between Release Consistency and Entry Consistency, in *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, Vol. 31, pp. 451–473 (1998).

[56] The LLVM Compiler Infrastructure, http://llvm.org/.

[57] Hara, K.: 再構成可能な高性能並列計算のための PGAS プログラミング処理系 (2011).

[58] Carter, J. B., Bennett, J. K. and Zwaenepoel, W.: Implementation and Performance of Munin, in *SOSP '91: Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Vol. 25, pp. 152–164 (1991).

[59] Noronha, R. and Panda, D.: Designing High Performance DSM Systems using InfiniBand Features, in *CCGrid '04: IEEE International Symposium on Cluster Computing and the Grid*, pp. 467–474, IEEE (2004).

[60] Mattern, F.: Virtual Time and Global States of Distributed Systems, in *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pp. 215–226, Elsevier Science Publishers B. V. (1989).

[61] Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, in *OSDI '96: Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 75–88, New York, New York, USA (1996), ACM Press.

[62] Yu, B., Huang, Z. and Cranefield, S.: Homeless and home-based lazy release consistency protocols on distributed shared memory, in *ACSC '04: Proceedings of the 27th Australasian Conference on Computer Science*, Vol. 26, pp. 117–123 (2004).

[63] Singh, J., Li, K. L. K. and Iftode, L.: Understanding Application Performance on Shared Virtual Memory Systems, in *ISCA '96: 23rd Annual International Symposium on Computer Architecture* (1996).

[64] Martin, M. M. K., Hill, M. D. and Sorin, D. J.: Why on-chip cache coherence is here to stay, *Communications of the ACM*, Vol. 55, No. 7, p. 78 (2012).

[65] Kaxiras, S., Klaftenegger, D., Norgren, M., Ros, A. and Sagonas, K.: Turning Centralized Coherence and Distributed Critical-Section Execution on their Head: A New Approach for Scalable Distributed Shared Memory, in *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 3–14, ACM Press (2015).

[66] R. Lebeck, A. and A. Wood, D.: Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors, in *ISCA '95: Proceedings of 22nd Annual International Symposium on Computer Architecture*, pp. 48–59 (1995).

[67] Belady, L. A., Nelson, R. A. and Shedler, G. S.: An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine, *Communications of the ACM*, Vol. 12, No. 6, pp. 349–353 (1969).

[68] Jaleel, A., Theobald, K. B., Steely, S. C. and Emer, J.: High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), in *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 60–71 (2010).

[69] Esteve, A., Ros, A., Gomez, M. E., Robles, A. and Duato, J.: Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, No. 3, pp. 748–761 (2016).

[70] Lai, A.-C. and Falsafi, B.: Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction, in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, No. June, pp. 139–148, ACM Press (2000).

[71] Nair, R.: Dynamic Path-Based Branch Correlation, in *MICRO-28: Proceedings of the 28th annual international symposium on Microarchitecture*, pp. 15–23, IEEE Computer Society Press (1995).

[72] Ros, A., Cuesta, B., Gomez, M. E., Robles, A. and Duato, J.: Temporal-Aware Mechanism to Detect Private Data in Chip Multiprocessors, in *ICPP '13: 42nd International Conference on Parallel Processing*, pp. 562–571, IEEE (2013).

[73] Cuesta, B., Ros, A., Gomez, M. E., Robles, A. and Duato, J.: Increasing the Effectiveness of Directory Caches by Avoiding the Tracking of Noncoherent Memory Blocks, *IEEE Transactions on Computers*, Vol. 62, No. 3, pp.

482–495 (2013).

[74] Hardavellas, N., Ferdman, M., Falsafi, B. and Ailamaki, A.: Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches, *Proceedings of the International Symposium on Computer Architecture*, No. June, pp. 184–195 (2009).

[75] Kim, D., Ahn, J., Kim, J. and Huh, J.: Subspace snooping, in *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, p. 111, New York, New York, USA (2010), ACM Press.

[76] Ros, A. and Kaxiras, S.: Complexity-Effective Multicore Coherence, in *PACT '12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, p. 241, New York, New York, USA (2012), ACM Press.

[77] Ramesh, B.: Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems (2013).

[78] Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C. and Yang, G.: The Sunway TaihuLight supercomputer: system and applications, *Science China Information Sciences*, Vol. 072001, (2016).

[79] Kee, Y.-S., Kim, J.-S. and Ha, S.: Memory management for multi-threaded software DSM systems, *Parallel Computing*, Vol. 30, No. 1, pp. 121–138 (2004).

[80] Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359 (1989).

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

[82] Taura, K.: A Quest for Unified, Global View Parallel Programming Models for Our Future, in *ROSS '16: Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, pp. 1–1, New York, New York, USA (2016), ACM Press.

[83] MacDonald, S., Szafron, D. and Schaeffer, J.: Rethinking the Pipeline as Object-Oriented States with Transformations, in *HIPS 2004: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Vol. 00, pp. 12–21, IEEE (2004).

[84] Reed, E. C., Chen, N. and Johnson, R. E.: Expressing Pipeline Parallelism using TBB constructs: A Case Study on What Works and What Doesn't, in *SPLASH '11 Workshops: Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, p. 133, New York, New York, USA (2011), ACM Press.

[85] Lee, I.-t. A., Leiserson, C. E., Schardl, T. B., Sukha, J. and Zhang, Z.: On-the-Fly Pipeline Parallelism, in *SPAA '13: Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, Vol. 2, p. 140, ACM Press (2013).

[86] Brent, R. P.: The Parallel Evaluation of General Arithmetic Expressions, *Journal of the ACM*, Vol. 21, No. 2, pp. 201–206 (1974).

[87] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *Journal of the ACM*, Vol. 46, No. 5, pp. 720–748 (1999).

[88] Mohr, E., Kranz, D. and Halstead, R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).

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

[90] Chase, D. and Lev, Y.: Dynamic Circular Work-Stealing Deque, in *SPAA '05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, No. c, p. 21, ACM Press (2005).

[91] Lê, N. M., Pop, A., Cohen, A. and Zappa Nardelli, F.: Correct and Efficient Work-Stealing for Weak Memory Models, in *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel*

programming, No. 1, p. 69, New York, New York, USA (2013), ACM Press.

[92] Kowalke, O.: Boost.Coroutine, http://www.boost.org/doc/libs/1_63_0/libs/coroutine/doc/html/index.html.

[93] Kowalke, O. and Goodspeed, N.: N3985: A proposal to add coroutines to the C++ standard library, Technical report (2014).

[94] IEEE, T. and Group, T. O.: The Open Group Base Specifications Issue 7 (2008).

[95] Engelschall, R. S.: Portable Multithreading: The Signal Stack Trick for User-space Thread Creation, in *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, p. 20, Berkeley, CA, USA (2000), USENIX Association.

[96] Kowalke, O.: Boost.Context, http://www.boost.org/doc/libs/1_63_0/libs/context/doc/html/index.html.

[97] Nishanov, G. and Radigan, J.: N4402: Resumable Functions (revision 4), Technical report (2015).

[98] Taylor, I. L.: Split Stacks in GCC, https://gcc.gnu.org/wiki/SplitStacks (2011).

[99] Nishanov, G., Maurer, J., Smith, R. and Daveed Vandevoorde, : P0057R7: Wording for Coroutines, Technical report (2015).

[100] Riegel, T.: P0073R2: On unifying the coroutines and resumable functions proposals, Technical report (2016).

[101] Antoniu, G., Bougé, L. and Namyst, R.: An efficient and transparent thread migration scheme in the PM2 runtime system, in *RTSPP '99: Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, Vol. 1586, pp. 497–510 (1999).

[102] Nakashima, J.: 高効率な I/O と軽量性を両立するマルチスレッド処理系の設計と実装 (2011).

[103] Nakashima, J. and Taura, K.: 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 (2011).

[104] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Vol. 8665, pp. 222–238 (2014).

[105] Taura, K.: *Efficient and Reusable Implementantion of Fine-Grain Multithreading and Garbage Collection on Distributed-Memory Parallel Computers*, PhD thesis (1997).

[106] Taura, K. and Yonezawa, A.: Fine-grain Multithreading with Minimal Compiler Support—A cost Effective Approach to Implementing Efficient Multithreading Languages, *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, Vol. 32, No. 5, pp. 320–333 (1997).

[107] Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, in *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, Vol. 34, pp. 60–71, New York, New York, USA (1999), ACM Press.

[108] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, in *IPDPS Miami 2008 - Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Program and CD-ROM* (2008).

[109] Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Castello, A., Genet, D., Herault, T., Jindal, P., Kale, L., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y. and Beckman, P. H.: Argobots : A Lightweight , Low-Level Threading and Tasking Framework, Technical report (2016).

[110] Intel Threading Building Blocks, https://www.threadingbuildingblocks.org/.

[111] Nanos++, https://pm.bsc.es/nanox.

[112] Intel Cilk Plus, https://www.cilkplus.org/.

[113] Van Nieuwpoort, R. V., Wrzesińska, G., Jacobs, C. J. H. and Bal, H. E.: Satin: A High-Level and Efficient Grid Programming Model, *ACM Transactions on Programming Languages and Systems*, Vol. 32, No. 3, pp. 1–39 (2010).

[114] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based Load Balancing, in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, p. 55, ACM Press (2008).

[115] Hiraishi, T., Yasugi, M. and Umatani, S.: 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論文集, pp. 55–63 (2011).

[116] Dinan, J., Krishnamoorthy, S., Larkins, D. B., Nieplocha, J. and Sadayappan, P.: Scioto: A Framework for Global-View Task Parallelism, in *ICPP '08: Proceedings of the 37th International Conference on Parallel Processing*, pp. 586–593, IEEE (2008).

[117] Min, S.-J., Iancu, C. and Yelick, K.: Hierarchical Work-Stealing Framework for Multi-core Clusters, in *PGAS '11: Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models* (2011).

[118] Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V. and Takeuchi, M.: GLB: Lifeline-based Global Load Balancing Library in X10, in *PPAA '14: Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, pp. 31–40 (2014).

[119] Endo, W. and Taura, K.: PGAS 向け低水準通信レイヤーのマルチスレッド実装, 研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2016, No. 28, pp. 1–11 (2016).

[120] Ajima, Y., Sumimoto, S. and Shimizu, T.: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers, *Computer*, Vol. 42, No. 11, pp. 36–40 (2009).

[121] Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S. and Shimizu, T.: The Tofu Interconnect, in *HOTI '11: Proceedings of IEEE 19th Annual Symposium on High Performance Interconnects*, pp. 87–94 (2011).

[122] Alverson, R., Roweth, D. and Kaplan, L.: The Gemini System Interconnect, in *HOTI '10: Proceedings of the 18th IEEE Symposium on High Performance Interconnects*, pp. 83–87, IEEE (2010).

[123] Chen, D., Parker, J. J., Eisley, N. a., Heidelberger, P., Senger, R. M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D. L. and Steinmacher-Burow, B.: The IBM Blue Gene/Q Interconnection Network and Message Unit, in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 1, ACM Press (2011).

[124] Birrittella, M. S., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K. D. and Zak, R. C.: Intel® Omni-path Architecture: Enabling Scalable, High Performance Fabrics, *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 1–9 (2015).

[125] Bonachea, D. and Jeong, J.: GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages, *CS258 Parallel Computer Architecture Project*, pp. 1–27 (2002).

[126] Daily, J., Vishnu, A., Palmer, B., Dam, van H. and Kerbyson, D.: On the Suitability of MPI as a PGAS Runtime, *HiPC '14: The 21st annual IEEE International Conference on High Performance Computing* (2014).

[127] OpenFabrics Alliance, : OpenFabrics, https://www.openfabrics.org/.

[128] Zhou, H., Mhedheb, Y., Idrees, K. and Glass, C. W.: DART-MPI : An MPI-based Implementation of a PGAS Runtime System, *PGAS '14: Proceedings of the 8th International Conference on PGAS Programming Models* (2014).

[129] Kumar, S., Sun, Y. and Kale, L. V.: Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q, in *IPDPS '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 689–699, IEEE (2013).

[130] Balaji, P., Buntinas, D., Goodell, D., Gropp, W. and Thakur, R.: Toward Efficient Support for Multithreaded MPI Communication, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 120–129, Springer Berlin Heidelberg (2008).

[131] Balaji, P., Buntinas, D., Goodell, D., Gropp, W. and Thakur, R.: Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming, *International Journal of High Performance Computing Applications*, Vol. 24, No. 1, pp. 49–57 (2010).

[132] MPICH, http://www.mpich.org/.

[133] Dinan, J., Balaji, P., Goodell, D., Miller, D., Snir, M. and Thakur, R.: Enabling MPI Interoperability Through Flexible Communication Endpoints, *EuroMPI '13: Proceedings of the 20th European MPI Users' Group Meeting*, p. 13 (2013).

[134] Vaidyanathan, K., Kalamkar, D. D., Pamnany, K., Hammond, J. R., Balaji, P., Das, D., Park, J. and Joó, B.: Improving Concurrency and Asynchrony in Multithreaded MPI Applications using Software Offloading, in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 30:1–30:12 (2015).

[135] Amer, A., Lu, H., Wei, Y., Balaji, P. and Matsuoka, S.: MPI+Threads: Runtime Contention and Remedies, in *PPoPP '15: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 239–248 (2015).

[136] Lu, H., Seo, S. and Balaji, P.: MPI+ULT: Overlapping Communication and Computation with User-Level Threads, in *HPCC '15: IEEE 17th International Conference on High Performance Computing and Communications*, pp. 444–454 (2015).

[137] Saga, K., Ajima, Y., Nose, T., Miura, K. and Sumimoto, S.: ACP 基本層の実装と初期評価, 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-HPC-1, No. 10, pp. 1–6 (2014).

[138] Nose, T., Ajima, Y., Saga, K., Shida, N. and Sumimoto, S.: ACP ライブラリの性能最適化に関する検討, 研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2015-HPC-1, No. 39, pp. 1–6 (2015).

[139] Frey, P. W. and Alonso, G.: Minimizing the Hidden Cost of RDMA, in *ICDCS '09: Proceedings of 29th IEEE International Conference on Distributed Computing Systems*, pp. 553–560 (2009).

[140] 富士通株式会社：FX10 スーパーコンピュータシステムについて, http://www.cc.u-tokyo.ac.jp/system/fx10/fx10-tebiki/chapter2.html.

[141] Sumimoto, S., Ajima, Y., Saga, K., Nose, T., Shida, N. and Nanri, T.: The Design of Advanced Communication to Reduce Memory Usage for Exa-scale Systems, in *VECPAR '16: 12th International Meeting on High Performance Computing for Computational Science* (2016).

[142] 富士通株式会社：FX10 利用者ガイド MPI 編 1.1 版 (2014).

[143] InfiniBand Trade Association, : Supplement to InfiniBand Architecture Specification Volume 1.2.1 Annex A14: Extended Reliable Connected (XRC) Transport Service Revision 1.0 (2009).

[144] Endo, W. and Taura, K.: 再配置可能な大域アドレス空間システムの設計と RDMA を用いた実装, 研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2015-HPC-1, No. 5, pp. 1–8 (2015).

[145] Berger, M. J. and Colella, P.: Local adaptive mesh refinement for shock hydrodynamics, *Journal of Computational Physics*, Vol. 82, No. 1, pp. 64–84 (1989).

[146] Akiyama, S. and Taura, K.: 軽量マルチスレッディング向け大域アドレス空間ライブラリ, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 135, No. 7, pp. 1–6 (2012).

[147] Endo, W.: タスク並列処理に適した大域アドレス空間システムのマルチスレッド実装 (2014).

[148] Farreras, M., Almási, G., Caşcaval, C. and Cortes, T.: Scalable RDMA performance in PGAS languages, in *IPDPS '09: Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium* (2009).

[149] Chavarría-Miranda, D., Agarwal, K. and Straatsma, T. P.: Scalable PGAS Metadata Management on Extreme Scale Systems, in *CCGrid '13: Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 103–111 (2013).

[150] Kulkarni, A. and Lumsdaine, A.: Active Global Address Space (AGAS): Global Virtual Memory for Dynamic Adaptive Many-Tasking Runtimes (2015).

[151] Midorikawa, H. and IIzuka, H.: ユーザレベル・ソフトウェア分散共有メモリ SMS の設計と実装, 情報処理学会論文誌: ハイパフォーマンス・コンピューティングシステム, Vol. 42, No. SIG 9 (HPS 3), pp. 170–190 (2001).

[152] Joerg, C. F.: *The Cilk System for Parallel Multithreaded Computing*, PhD thesis (1996).