修士論文

分散タスク並列ライブラリ MassiveThreads/DM の設計とその開発支援ツールの実装

Design and Implementation of Distributed Task Parallel Library MassiveThreads/DM and its supporting tools

平成 25 年 2 月 6 日提出

指導教員　　田浦 健次朗 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-116404 池上 克明

近年の計算機環境はノード内の並列度の向上や階層化など複雑化を続けているが，現在普及しているプログラミング環境はハードウェア構造を単純に写像するもので，プログラマに負担を強いている．例えば MPI は個々のプロセスのアドレス空間を分散したまま見せている．

一方，並列プログラミングの生産性を高める試みの一つに大域アドレス空間モデルが挙げられる．大域アドレス空間はノードごとに分散されたメモリ空間を仮想的な共有メモリ空間として扱い，プログラムの記述を容易にするものである．また，並列計算の記述を容易にする手法としてはタスク並列モデルが挙げられ，オーバーヘッドの小さいタスクを十分多く生成し並列度と負荷分散を両立するもので，共有メモリ環境においていくつかの実装が登場している．

この大域アドレス空間モデルとタスク並列モデルを組み合わせることで，より分散並列プログラミングが容易になると考えられるが，現状この二つを満たす高性能な処理系は存在しない．その理由としては大域アドレス空間へのアクセスに通常のメモリアクセスよりも格段に大きいオーバヘッドが存在するため，タスクに合わせて通信の粒度も細かくなることが性能を阻害するからだと考えられる．そこで，筆者は新しい大域メモリのセマンティクスを導入した分散タスク並列モデルライブラリ MassiveThreads/DM を設計した．

また大規模分散環境は一般に可用性が通常の環境より低く，事前にその環境での性能を予測することが出来ればより有効にプログラムの開発を行うことが可能であると言える．そのため筆者はこのライブラリ MassiveThreads/DM で記述されたプログラムの正当性や性能を，通常の共有メモリ環境下で仮想的に実行することで検証推定する開発支援ツール Mgascheck を汎用動的実行バイナリ解析フレームワークである Valgrind 上に実装した．

目次

図目次

表目次

第 1 章

序論

1.1 背景

近年の並列計算機環境は並列度の向上により性能を高めており，CPU のコア数もノード数も増大する方向である．そのため，計算機の階層構造は深化する一方で，データの配置を階層構造を考慮して最適化することなく性能を発揮することはできない．

一方で，今に至るまで主流であるプログラミング環境は低レイヤなものに留まっている．特に，分散メモリ環境の場合に広く使われている MPI は計算機の持つメモリ空間をそのままユーザに提供し，自ノードのメモリ空間と他ノードのメモリ空間との間で明示的に通信を記述させることで，ユーザが明示的に最適化する余地を与えている．一方でユーザにとっては抽象度が低い MPI を利用してプログラムを記述することが負担になっている．MPI でプログラムを記述する場合，ユーザはデータや処理の位置を CPU のコア数やノード間通信のトポロジーなど実際のハードウェア構成を考慮した上で決定する必要がある．

このように現時点では高性能に大規模並列プログラムを記述するにはアプリケーションを開発するユーザ自身が低レベルな部分まで十分考慮してプログラムを記述しなければならない．これに対し，大域的な視点でプログラムを記述できるようにする試みがかつてから広くなされてきた．そのうち，現在多く研究がされているものが PGAS (Partitioned Global Address Space) モデルである．PGAS モデルはデータに関する大域的なインターフェスを提供する．具体的には共有メモリモデル同様の分散環境下でも一意なメモリアドレス空間を提供し，プログラムの記述を容易にしている．従来から High Performance Fortran [14] など共有メモリインターフェースを提供して分散環境でのプログラミングを容易にする研究はなされてきたが，PGAS モデルではデータの配置をユーザが明示的に管理することで性能と記述性のバランスをとっている．PGAS モデルを採用したプログラミング言語としては UPC [6], Co-array Fortran[20], Chapel [7], X10 [8] などが挙げられる．

一方でノード内の処理を並列化する際に有用なプログラミングモデルとして，軽量タスク並列処理系を利用した並列分割統治法がある．軽量タスク並列処理系とは従来のように OS の提供するスレッドをそのまま利用して並列処理を記述するのではなく，ユーザは処理系の提供する低オーバヘッドなタスクを十分多く生成し，これを処理系が OS スレッドに割り当てることで記述性と並列度と

負荷分散の両立を狙ったものである．現在このような処理系として Cilk [5], Intel TBB [21], Java fork-join framework [13], StackThreads/MP [26] や MassiveThreads [27] が知られている．これらを用いて分割統治法を記述すると再帰的に問題を分割する際に分割した部分問題は並列に解くことが出来ることが多く，自然に並列計算が記述でき，また各タスク内部で利用するメモリ領域も再帰的に問題を分割することで階層的に整理され，ハードウェアの階層的メモリ構造にマッピングされやすくキャッシュを効率よく利用できる [11].

このように共有メモリ環境に於いては高い性能を発揮する軽量タスク並列処理系であるが，現状の分散メモリ環境ではこれをうまく実現したプログラミング言語処理系は存在しない．先に挙げた PGAS モデル言語の例でみても，UPC や Co-array Fortran の実行モデルは SPMD であり，Chapel や X10 は動的に並列タスクを生成することは可能ではあるが，その負荷分散はあくまでノード内にとどまり，ノード間ではユーザが明示的にタスクの実行されるノードを指定する必要がある．これが並列分割統治法を分散環境上で利用する妨げの一つとなっている．

1.2　目的

以上のように並列分割統治法を軽量タスク並列モデルを用いることで平易なプログラムの記述で高いキャッシュ効率と並列実行の負荷分散を両立し，結果としてユーザはハードウェアの階層構造を意識した明示的な最適化を行わなくても階層構造を効率的に利用することが出来る．そこで，現状ではプログラミングが容易とは言えない分散並列環境においてもそのプロセッサやメモリ等計算資源の階層化に自ずと対応する並列分散統治法を利用することが出来れば，分散並列環境におけるプログラムの生産性を大きく高めることが出来ると考えられる．そのためにはタスクの位置によらずデータを共有できる必要があるので，PGAS モデルのような大域的なデータモデルが求められる．そのため PGAS モデルを提供する言語に対してタスク並列モデルを組み込むことは並列分散統治法を記述するモデルとして適切である．

しかし，既存の PGAS モデルは SPMD モデル，あるいは粗粒度なタスク並列モデルをその実行モデルとして対象にしているため，細粒度タスク並列モデルと親和性が高くない．PGAS モデルにおいても他ノード上の GAS のデータに対するアクセスはオーバーヘッドが大きいため，通信の集約を行うことが必要になる．SPMD モデルやノードを明示的にユーザが指定するモデルの場合はデータにアクセスする前後にまとめて通信を行い，実際の計算はローカルなデータに対して行うことでユーザが明示的に通信の集約は可能であるが，この記述方法では大域的な視点でのプログラミングモデルが崩れてしまっている．更に，細粒度分散タスク並列処理系においては自動的な負荷分散を行うためタスクが実際に実行されるノードはユーザではなく処理系が決定する．そのため同じノードで実行されるタスク間でユーザが PGAS 上のデータを明示的に共有することはできない．結果として並列分割統治法によってプログラムを記述した際にリーフのタスクで毎回 PGAS に対してアクセスすることが要求され，オーバヘッドの影響を受けて性能が大幅に劣化することが予想される．

これから，ノード間でのタスクマイグレーションを伴う分散タスク並列モデルはその提供するメモリモデルとして既存の PGAS とは異なるモデルを採用する必要があることが分かる．また，タスク

並列モデルが共有メモリ環境で高い性能で動作するのはハードウェアキャッシュの恩恵であることから，PGAS モデルに対して何らかのキャッシュを導入することで分散タスク並列における通信の粒度の問題を解決しうることが予想される．

そこで本論文においては並列分割統治法を分散メモリ環境上でも利用できるような PGAS タスク並列モデル MassiveThreads/DM の設計を提案する．この MassiveThreads/DM は既存の PGAS モデルと同様にデータの配置や通信の発生する記述をユーザが明示的に行うことで性能の最適化を容易にしているが，PGAS 上のデータに対するノード内キャッシュを処理系が管理しユーザがこれに操作を行うことで通信の集約を自明な形で記述することが出来る．

加えてこの PGAS のセマンティクスはまだ多くのプログラマに対して知られているものではなく，記述ミスにより不適切なコードを記述してしまうことが考えられる．そのようなバグを事前に検証可能なツールを提供する事でプログラムの生産性を高めることが可能である．実際に実行した際の通信がどの程度発生するかを予測することは大規模環境での実行の前にアルゴリズム自体やコードに問題がないかを確認可能にする点で有意義である．MassiveThreads/DM のプログラミングモデルに基づいて逐次実行ではあるがプログラムを実行した際の通信の性能を擬似的に計測し，またメモリのセマンティクスを適切に順守しているかを検証するツールをあわせて提案する．

第 2 章

MassiveThreads/DM

本節ではまず従来の PGAS モデルとタスク並列モデルを併用する際にプログラミングモデルから性能上の制約が生じることを示す．そして，この制約を受けることなく高い生産性を実現する GAS のインターフェースを提案する．次に MassiveThreads/DM の PGAS 上でのタスク並列インターフェースを提案する．

2.1 概要

MassiveThreads/DM は MassiveThreads [27] を分散環境向けに拡張したタスク並列ライブラリである．ユーザは C 言語や C++ 言語でプログラムを記述し，MassiveThreads/DM とリンクすることでプログラムを分散環境上で並列に実行させることができる．MassiveThreads/DM のプログラミングモデルは主に実行を並列化する部分のタスク並列モデルとデータの共有を指示する PGAS モデルからなる．

2.2 共有メモリでのタスク並列モデル

Cilk, Intel TBB, MassiveThreads など共有メモリ上のタスク並列ライブラリは多く研究実装されているが，MassiveThreads/DM も同じ実行モデルを分散環境上で実現したものである．そのため，本項ではまず共有メモリ上でのタスク並列モデルの動作を解説する．

タスク並列モデルでは並列計算の記述として**タスク**を最小の構成要素として提供する．ここで，タスクとはスレッド同様の，プログラム中のある部分の実行を並行に行うというセマンティクスを持っている．一方で，各タスクは実際の OS のスレッドに対応しない．OS の提供するネイティブスレッドをプロセッサの個数より多く生成して同時に実行した場合，コンテキストスイッチの際に大幅なオーバヘッドを伴うので実行効率の低下を招いてしまう．そこで，タスク並列モデルのランタイムはプロセッサの個数だけ**ワーカスレッド**を生成し，ランタイム自身で各タスクをワーカスレッドに割り当てるというユーザレベルスケジューリングを行う．この際にタスク並列処理系はワーカスレッドごとにタスクキューを用意し，これを用いてタスクのスケジューリングを行う．

タスク並列処理系のこれらのスケジューリングに関してはいくつかの方式があるが，LIFO スケジューリングを行うことで高い性能を得ることが出来る．LIFO スケジューリングとはタスクキューに後に追加されたものから実行する方式で，通常の逐次プログラムの実行順と同様に再帰の深いタスクから実行される．

また，タスク生成の際に，生成されたタスクを先に実行し親タスクをタスクキューに入れて一時停止させ work-first と，親タスクの実行を継続し，生成したタスクをタスクキューに入れる help-first との二つのポリシーが考えられるが，work-first は後述するタスクスチールがない場合逐次プログラムと実行順序が一致することや，lazy task creation と親和性が高いことから多くのタスク並列処理系がこれを採用している．

また，上に示した実行モデルだけでは実はプログラムが並列に実行されることはない．なぜなら，あるタスクはタスクを生成し実行がそちらに移るが，元のタスクは子タスクが終了するまで実行されないからである．その間に問題となるのは他のワーカスレッドであり，他のワーカスレッドは消費すべきタスクがない．そこで，ワーカスレッドは自分のタスクキューの中で実行可能なタスクがなくなった場合，他のワーカスレッドに対してタスクスチールを行う．このタスクスチールは FIFO スチールと言ってなるべくタスクキューの先頭にあるタスクを取ってくる．このため，FIFO スチールではより以前に作られたタスク，つまり再帰の浅いタスクを取ってくることになり，スチールした後十分な子タスクが生成されてタスクスチールのオーバヘッドが相対的に無視できることが期待される．このタスクスチールを導入することでタスクが十分多い状況では負荷分散を図ることができる．タスクスチールを行うには相手を定める必要があるが，これはランダムに定めることで最適な性能を得ることが証明されている．

2.3　MassiveThreads/DM のタスクモデル

上で説明した共有メモリ上のタスク並列モデルと同様に MassiveThreads/DM では並列計算の記述として**タスク**を最小の構成要素として提供する．図 2.1 に MassiveThreads/DM のタスクモデルの概要を示す．図中の円は各タスクを意味し，その間の矢印はタスクの親子関係を示している．各タスクは各自で計算を行いつつ，他のタスクを生成することで実行を並列化することが出来る．MassiveThreads/DM はタスクの負荷分散をタスクスチールによって行う．このタスクスチールは実行環境全体で起こるため，ノード内に限らずノード間でタスクが移動することもある．図の node 1 から node 2 へ移動しているのがその例である．

MassiveThreads/DM のタスク API を図 2.2 に，使用例を図 2.3 に示す．図 2.2 にあるように MassiveThreads/DM は pthread など通常のスレッドライブラリと同様のインターフェースでタスクを生成することが出来る．ただし，タスク間のデータのやり取りは GAS を通して行われるので，タスクへの引数やタスクの戻り値は GAS のポインタを利用してやり取りする必要がある．図 2.3 に示すように，MassiveThreads/DM は通常の C 言語の呼び出し規約に基づいたライブラリであり，

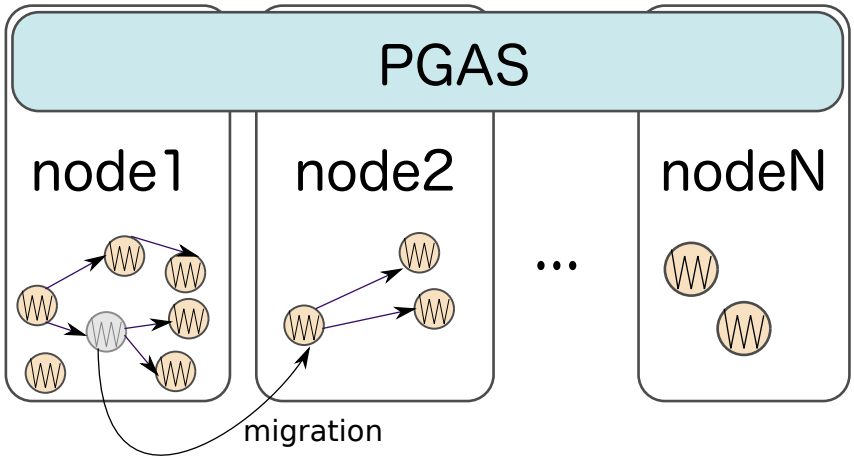

図 2.1. MassiveThreads/DM におけるタスクモデル

```
1 task_t task_create(task_func_t func, globalptr_t p);
2 globalptr_t task_join(task_t task);
```

図 2.2. MassiveThreads/DM のタスク API

```
1  void seq_sort(int * ptr, size_t size);
2  void qsort(globalptr_t arg) {
3      // get ptr, size from arg.
4      if(size < threshold) {
5          return qsort(ptr, size/sizeof(int), sizeof(int), cmp);
6      }
7      // split array with some pivot (in normal qsort way).
8      task_t t1 = task_create(qsort, arg1); // left of pivot
9      task_t t2 = task_create(qsort, arg2); // right of pivot
10     task_join(t1);
11     task_join(t2);
12 }
```

図 2.3. MassiveThreads/DM のタスク API の利用例

スタックも連続したメモリ領域として提供され[*1]，タスクの内部では通常の C 言語の呼び出し規約に従っているのでプログラマは通常の C 言語の記述一般が可能である．図 2.3 の例ではタスクの中で通常の C 言語の関数である `qsort` を呼び出している．

MassiveThreads/DM は負荷分散のためにプロセス内のワーカスレッド間でワークスチールするだけでなく，プロセス間でのタスクマイグレーションも行う．共有メモリ環境の場合タスクはスチールされた後も同じ仮想アドレス空間上に存在するため，今までアクセスしてきたデータに問題なくアクセスすることが出来るのに対して，分散環境上ではタスクがノード間でマイグレーション

[*1] ただし通常のスタックとは異なりアドレス空間の後方から一直線に伸びるという仮定は置くことができない

する際にそのタスクのスタック以外はマイグレーションの対象とならず，一般にマイグレーション前のプロセスのローカルメモリに対してアクセスすることはできなくなる．そのためノード間のマイグレーションがタスクの実行中の任意の地点で発生しうると仮定すると，ローカルメモリを確保しても次の命令で参照する時点でマイグレーションしている可能性があることになり，プログラミングが不可能になってしまう．タスク並列モデルではタスクがその実行を休止するのはタスクの生成待ち合わせなど他のタスクとの同期操作が発生する場合に限られていることが多い．そのため，MassiveThreads/DM でもタスクの実行が中断して他のタスクに移る地点を migration point と呼び，タスクのマイグレーションを行う地点を migration point に限定する．MassiveThreads/DM では work-first なスケジューリングポリシーを採用しているため，タスクの生成直後は必ずそのタスクは実行中断されるので migration point となる．また，子タスクの終了を待つ際も相手の終了するまで実行が中断されるため migration point である．

MassiveThreads/DM は分散環境で実行するにあたって複数のプロセスを利用する．ここではプロセスをハードウェア共有メモリを有する計算機資源としてみなす．MassiveThreads/DM ではユーザはタスクを用いてプログラムを記述するので，タスクの実行されるプロセスを明示的に意識することはない．ただし，後述する GAS のキャッシュがプロセスを単位とする．

2.4　従来の PGAS モデルとタスク並列モデルの不整合

上で述べたように PGAS モデルは分散環境におけるデータ配置の抽象化を実現し，かたやタスク並列モデルは並列実行モデルの抽象化を実現する．これらを分散環境上で統合して利用することで，分散並列プログラムの記述に際して高い生産性を得ることが可能となると予想されるが，従来の PGAS モデルとタスク並列モデルを同時に採用することには問題がある．

並列分割統治法に従ったタスク並列のプログラムを通常の共有メモリ環境下で実行した場合，タスク木内でのタスク間の距離が近い，即ち時間的に近接したタスクであるばあるほど，各タスク内で操作するメモリ領域には局所性が存在する．そのため，ある一定のタスク深さに達した時に一度操作する領域がキャッシュされてしまえば，その深さ以下で生成するタスク内でのメモリアクセスはキャッシュミスを起こさない．そのためタスク内のメモリアクセスは最適にキャッシュに対してアクセスすることになる．

一方で従来の PGAS モデル上でタスク並列プログラムを記述する場合，共有メモリ上では起こらない問題が生じる．PGAS モデルではノード間で分散したメモリ空間を仮想的な共有メモリ空間としてユーザに提供しているが，一方で他のノード上のメモリへのアクセスは自ノード上のメモリへのアクセスと比べて高遅延・低帯域であり，高い性能を得るためにはノード間通信を削減することに加えて他ノード上に実体が存在する領域へのアクセスをなるべく粗粒度にすることが求められる．

従来の PGAS モデルを採用したプログラミング言語ではこのようなユーザによる PGAS へのアクセスの最適化を行うことが可能であった．SPMD モデルを採用する UPC では MPI 同様に PGAS に対するアクセスを粗粒度に行い，次に取得したローカルなデータに対して計算を行い，最後に結果をまた集約して PGAS に書き込むようなプログラムを記述することができる．また，Chapel や

X10 においてもコードが実行されるノードはプログラマが明示的に指定する必要があるので，そのノードにおいて PGAS へのアクセスを粗粒度に集約することが可能である．

しかし，ノード間にまたがって負荷分散を行うタスク並列モデルを従来の PGAS モデル上で適用すると PGAS に対するアクセスの集約を行うことができない．なぜなら，各タスクがどのノードで実行されるかは処理系に委ねられているためユーザが明示的に指定することはできない．つまり，あるノード上でデータを先読みしておくという操作を記述すること自体が不可能である．そのため，従来の PGAS モデルを採用する場合各タスクで細粒度な PGAS へのアクセスが生じ，結果として通信のオーバーヘッドから高い性能を得ることができなくなってしまう．

2.5　MassiveThreads/DM の PGAS モデル

以上から，タスク並列モデルで性能と記述性を両立するには PGAS モデルのセマンティクスに対して修正が必要であることがわかる．従来の PGAS モデルでは各タスク間で行う PGAS へのアクセスが独立に行われることで，結果としてアクセスが細粒度になってしまうことが問題だった．しかし実際には分割統治法を利用する場合近隣タスクではアクセスする領域も近接しているものと期待される．そこで，PGAS に対するアクセスをユーザではなく処理系の側でキャッシュすることで実行が同じノード上で行われた場合の PGAS へのアクセスコストを最小限にすることができると考えられる．一方で，どの領域にアクセスするかをユーザが明示的に指定することで，キャッシュすべき領域を処理系が判断する必要がなくなり，またユーザもデータのコンシステンシを管理しやすくなることが期待される．本項ではこのようにキャッシュ機能を提供しタスク並列処理系に適した MassiveThreads/DM のメモリモデルについての提案を行う．

2.5.1　アドレス空間

MassiveThreads/DM は PGAS モデルに基づいたメモリモデルを採用する．即ち，図 2.4 に示すように，ユーザの利用できるアドレス空間は各プロセスに限定されたローカルアドレス空間と，これとは完全に区分された全タスクから共有することができる大域アドレス空間 (GAS, Global Address Space) とに区別される．大域アドレス空間上にユーザがメモリ領域を確保する際に二つの種類から確保する方式を選択できる．一つは**大域オブジェクト**というもので，一プロセス上に確保される．もうひとつは**分散配列**というもので，大域アドレス空間上の複数の**ページ**の集合が一つの連続したメモリ領域としてユーザに提供される．これらの大域アドレス空間上のデータに対しては**大域ポインタ**を通してアクセスすることが可能である．また，大域オブジェクトや分散配列のページは配置されるプロセスをマイグレーションすることができ，また実体が移動しても大域ポインタは変化せず透過的に利用することができる．これらの GAS 領域の確保に利用するインターフェースを図 2.5 に示す．大域オブジェクトは `global_malloc`, `global_free` 関数を通して確保，解放される．`global_malloc` は確保するメモリ領域のサイズを引数に取り，確保された大域オブジェクトへのポインタを返す．`global_free` は解放する大域オブジェクトのポインタを引数に取る．また，分散配列の確保，解放

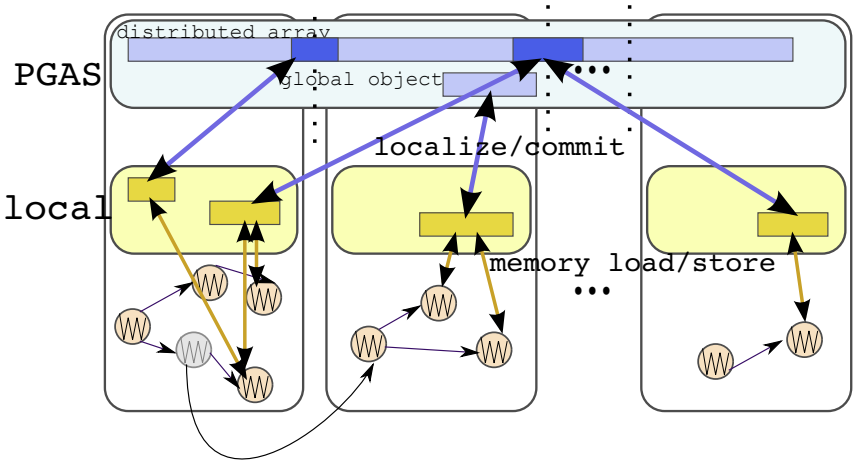

図 2.4. MassiveThreads/DM のメモリモデル

には `global_dmalloc`, `global_free` 関数を呼び出す．`global_dmalloc` は確保するメモリ領域のサイズだけではなく，1 ページあたりの大きさも引数で指定することができる．

2.5.2　GAS へのアクセス

セマンティクス

次に MassiveThreads/DM の GAS へアクセスする際のインターフェースを説明する．MassiveThreads/DM は細粒度タスク並列モデルを採用するため，各タスク内で行う GAS へのアクセスも細粒度になってしまう．しかし，性能上からは GAS へのアクセスは極力粗粒度にする必要がある．ここで，MassiveThreads/DM はプログラミングパラダイムとして分割統治法を念頭に置いている．そのためアプリケーションによるが，親タスクで子タスクのアクセスする領域が子タスクで操作する領域の和となり事前に分かることが多い．そのため，親のタスクで事前に通信して取得したデータを子供操作することで通信の集約を図ることができる．さらに各タスクがマイグレーションして親のタスクとは違うプロセスに移動した時でもなお正しく動作することが求められる．これらの要件を満たすのが以下で説明する *localize* インターフェースである．

localize インターフェースにおいても，当然 GAS 上のアドレスをそのまま参照することはできない．ユーザは大域アドレスに対して `localize` というインターフェースを通してアクセスする．`localize` は大域アドレスを受け取り，それに対応するプロセス内のローカルなキャッシュ領域を返すインターフェースである．このキャッシュ領域は通常のローカルメモリ領域であり，ユーザは一度この領域のポインタを得れば通常の C 言語プログラム同様にこのポインタを使用して良い．またこの領域は処理系が管理しユーザが自ら確保する必要はなく，また解放してはならない．このインターフェースは先に述べた性質を満たす．プロセス内のキャッシュ領域はプロセス内のタスクからは共有されている．そのため，プロセスのマイグレーションが起きなかった場合，一度親のタスクで localize した後に子のタスクで同じ領域を localize するとノード間通信を起こすことなく親のタスク

```
1  typedef struct {
2      globalptr_t p;
3      size_t size;
4  } global_memvec_t;
5  enum options {
6      REUSE = ...,
7      UPDATE = ...,
8      OWN = ...,
9  };
10 globalptr_t global_malloc(size_t size);
11 globalptr_t global_dmalloc(size_t size, size_t page_size);
12 void global_free(globalptr_t p);
13 void *localize(global_memvec_t read_vs[], size_t n_read_vs,
14                 global_memvec_t write_vs[], size_t n_write_vs,
15                 int options);
16 void commit(global_memvec_t write_vs[], size_t n_write_vs);
17 void invalidate(global_memvec_t vs[], size_t n_vs);
```

図 2.5. MassiveThreads/DM の GAS API

が localize した領域を返すことが出来る．一方で，マイグレーションが起きてかつて親が localize した領域が存在しないプロセスで子タスクが実行されている場合は，再びローカルなキャッシュメモリ領域を確保し GAS からデータを取得してからこの領域を返す．これによりユーザの記述上では**現在のサブタスクでアクセスする範囲を** *localize* **する**という単純なコードで現在のプロセスにキャッシュされていればそれを利用し，いなければ新たに取得するという動作を実現することが出来る．

localize API

localize の API も図 2.5 に示す．`localize` 関数は引数として `read_vs`, `n_read_vs`, `write_vs`, `n_write_vs` と `option` の 5 つを取る．`n_read_vs`, `n_write_vs` はそれぞれ配列長を与えるもので，`read_vs`, `write_vs` はそれぞれ GAS 中の読み込み，書き込みする範囲を指定するものである．`option` については後述する．`localize` を呼び出した時に，処理系は `read_vs` と `write_vs` を包含する領域に対応するローカルなメモリ領域を確保する．既にプロセス中に対応する領域が存在する場合はその領域を利用する．また必要ならばデータを GAS から取得しローカルな領域に書き込む．ここで，GAS の配列間のポインタのオフセットは `localize` の返すローカルなキャッシュ領域内でもそのまま利用できる．つまり，GAS 上の分散配列を `localize` した場合でも配列のインデックスをそのままローカルな領域に対して利用することができる．このことは配列を逐次的に読み込むようなアプリケーションでは自明ではあるが，アクセスパターンが複雑な際でも配列のインデックスが大域的なビューと同様に使用出来るというメリットになる．上で述べたように localize したメモリ領域は同じプロセス内で共通の領域を利用することがある．条件としては，同一プロセス内で A に対して localize した後その部分集合 B に対して localize した時に帰ってくるポインタは，A を localize した時の領域の中の B に対応する位置である可能性がある．このローカルなメモリ領域はそのプロセ

ス内で localize した全てのタスクが終了するか他のプロセスにマイグレーションした時点で解放される．

コヒーレンスの指定

`localize` はまた，`option` 引数を用いてデータのコヒーレンスに関する挙動を制御することが出来る．`option` は値として `REUSE`, `UPDATE`, `REUSE|OWN`, `UPDATE|OWN` を取ることができる．`OWN` に関しては後述する．`REUSE` が指定された場合，GAS に対応するローカルメモリ領域が存在する時 `localize` は実際の通信を行わずただその領域を返す．一方で `UPDATE` が指定された場合は必ず GAS からローカルメモリ領域へデータの読み込みを行う．これによりデータが途中で更新されるものか，途中で更新されず常に古いキャッシュを使っていいものかを指定できる．

ローカルなキャッシュ領域の寿命

localize で得たキャッシュ領域はプロセス固有のローカルメモリ上に存在する．また，この領域はタスクマイグレーションの際に移動されるわけではないので，マイグレーション後には無効なポインタとなってしまう．そのため大域アドレスに対する localize は migration point を通過する度に行う必要がある．

commit API

`localize` は GAS に対応するキャッシュ領域を作成し，データを GAS から取得するインターフェースである．一方このキャッシュ領域の内容を GAS に反映するものとして `commit` API がある．`commit` に渡す引数は `localize` 同様に GAS のうち書き込む部分を指定して渡す．

2.5.3　GAS のデータ配置

PGAS が従来の分散共有メモリ手法と異なる点は，一つは通信を明示的に行いユーザが通信の起こる箇所を把握しやすくするものであった．MassiveThreads/DM では localize, commit インターフェースを呼び出した時にしか GAS との通信は発生しないのでこの特徴を備えている．PGAS モデルのもつもう一点の特徴として，データの配置をユーザが明示的に指定できる点で性能を最適化する余地をプログラマに提供している点が挙げられる．UPC や Chapel など既存の PGAS モデルに基づくプログラミング言語では PGAS 上のデータをブロック分割やサイクリック分割，あるいは言語によってはユーザ定義の分割によって実際のプロセスに割り当てることが出来る．

しかし，既存の PGAS はこのデータ配置の指定に関してもタスク並列モデルと親和性が低い．なぜならタスクが実際に実行されるノードは実行時に決定されるために，コンパイル時にどのように配列を分散すればよいのかを知ることができない．そのため MassiveThreads/DM ではデータ配置の指定のために PGAS 上のデータを移動することでデータ配置も動的に行い，データの局所性を高めることを可能にした．localize のインターフェースの `options` 引数として `OWN` を指定した場合，MassiveThreads/DM はアクセスする GAS 領域の実体を呼び出したタスクが実行されているプロセ

スへマイグレーションする．逆に OWN が指定されていなければ，実体の位置に関わらずローカルなキャッシュ領域を用意するだけの挙動を示す．この OWN に関しても，ある一定の粒度のタスクから常に OWN を指定しておけば親のタスクであらかじめデータを自プロセスに移動させておき，子のタスクでは実体にアクセスする際もローカルなメモリコピーで済むようになる．

2.5.4　注意すべき点

localize で得たキャッシュ領域は migration point を通過した後に利用することはできないが，これは migration point を通過した場合プログラムが実行されるプロセスが変化している可能性があり，データのコンシステンシーやキャッシュ領域自体の状態が migration point 以前と同じではないことを前提とする必要があるからである．つまり，プロセス自体が変更される可能性があり，またユーザからはタスクの実行されるプロセスは指定できないことから通常のグローバル変数に対して書き込みを行った場合結果は不定となる．また，migration point 以前に動的に確保したローカルなメモリを migration point 以後に引き続き利用することは，タスクが他のプロセスにマイグレーションされた場合ユーザの期待するメモリ領域にアクセスできないため明確に禁止される．これらの通常の共有メモリ環境とのセマンティクスの差異を正確に反映していることを検証するのが次章の Mgascheck である．

第 3 章

Mgascheck

本項では上で提案したタスク並列ライブラリ MassiveThreads/DM の支援ツールの設計実装を解説する．

3.1 概要

上で説明したように MassiveThreads/DM の提供するプログラミングモデルに基づいてプログラムを記述する際に，ユーザは共有メモリにおけるタスク並列による分割統治法のプログラムに対して localize, commit を追加するだけで分散環境で動作するように変更することが出来る．しかし localize, commit にはマイグレーションからくる制約が存在し，これを守ってプログラミングを行わなかった際は，マイグレーションしない時に発生せず，マイグレーションした時だけ発生する等，再現しづらいバグが生じてしまう．一方で，これは定式化されたプログラミングモデル上の制約であり，プログラムを実行した際にこの制約が満たされているかは機械的に判定することが出来る．そのため，本項で提案する Mgascheck はこのような MassiveThreads/DM の持つ GAS の制約が満たされているかどうかを検査し，ユーザの記述したコードのデバッグを支援することが出来る．

また，分散並列プログラムを動作させる環境は多岐にわたり，もっとも小規模なものでは通常のパーソナルコンピュータやサーバ一台でも動作させることができるが，中規模なものではコモディティサーバからなるクラスタ，更に大規模なものとしてはスーパーコンピュータなどでも動作させることが可能である．しかし，小規模な環境と異なり大規模並列環境は需要に対して供給が不足しているため利用できる機会は限られている．また，このような環境では一般に対話的にプログラムを実行することはできず，バッチジョブによってプログラムを実行するため，インクリメンタルなプログラムのデバッグや最適化を行うことは不可能である．そのため，大規模並列環境で実行する前にプログラムの性能を最適化しておくことが重要になる．そのため，並列プログラムにおいて並列度が増大した際にどのような挙動を示すかを事前に定量的に知ることが出来ることが有意義であると考えられる．そこで Mgascheck はプログラムの並列実行時におけるノード間の通信量を見積もることで並列度に応じた性能を予測する．

3.2　Valgrind

Mgascheck は MassiveThreads/DM を利用したプログラムに修正を加えることで所定の動作を実現するが，ここではそのフレームワークとして Valgrind を利用した．Valgrind [17, 23, 16] は memcheck というメモリデバッガが Valgrind 上で構築されたツールとして有名である[*1]が, Valgrind 自身は実行時にプログラムの挙動を解析するためのフレームワークであり，その対象としてメモリデバッガを主眼に置いている．そのため Valgrind ではあるメモリ上のオブジェクトに対して shadow というものを用意し，オブジェクトのメタ情報を shadow に格納することでメモリデバッグやキャッシュ解析などを汎用的に行える shadow memory tools を記述することを念頭に置いている．

一般に実行バイナリを解析するにはプログラムの挙動を解析するためにコード列を改変して解析用のコードを挿入する必要がある．このコード列の改変をコンパイル時に行うことも可能ではあるが, 既に存在するバイナリをそのまま再コンパイルなしで解析できる方が利便性は高いと言える．そのため実行バイナリに対して直接命令を改変可能なツールが求められている．従来のこのようなツールではコード断片を挿入する手法が用いられてきたが，Valgrind ではより汎用的な手法を用いている．Valgrind はプログラムを実行バイナリを一度中間表現に変換してこの中間表現に対して操作を行う．更に Valgrind はシステムコールや標準ライブラリ関数のフックなども提供し，開発者がバイナリ解析の本質的な部分だけをプログラムするだけでツールを作成できるようになっている．このフレームワークの提供する部分を core, 実際にユーザが記述する部分を tool plugin と呼び，Valgrind を利用してツールを作成する際はこの tool plugin を C 言語を用いて記述することになる．

次に Valgrind を用いた際のプログラムの実行の概要と実際の Valgrind におけるコーディングの関係を述べる．まずプログラムの起動時の処理を述べる．Valgrind のツールはそれぞれ tool plugin と core を合わせた形で静的リンクされている．プログラムは通常と異なるアドレス上にロードされる．通常では `0x38000000` 番地が利用される．Valgrind は起動直後に core の一部を初期化し，対象となる client の実行ファイルを読み込み展開する．この時点で core は `xx_pre_clo_init`, `xx_post_clo_init`API を介して各ツールの初期化を行い，またコマンドラインオプションの処理などを行う．最後に core を完全に初期化して初期化を終える．

Valgrind のツールが起動した後は，対象となるプログラムのコード列を変換して実行するフェーズに入る．Valgrind はネイティブのマシンコードを静的単一代入 (SSA) 形式の中間コードに逆アセンブルする．この変換は一気にすべてのコード列に対して行われるのではなく，動的に superblock ごとに行われる．ここでいう superblock は基本ブロック (basic block) と対比される概念であり，basic block が 1 つの開始点 (entry point) と 1 つの終了点 (exit point) を持つのに対して superblock は 1 つの開始点と複数の終了点を持つようなコードの集合である．

それぞれの IR ブロックは複数の副作用のある文からなる．副作用とはレジスタやメモリへの書き込み，そして SSA で用いられる一時変数への書き込みである．さらにそれぞれの文は式からなり，

[*1] valgrind のデフォルトのツールが memcheck である．

式は副作用を持たず，具体的には定数やレジスタの読み込みからなる．式はネストすることもでき，また一時変数を利用してネストを解消することも可能である．IR は RISC に従った要素からなり，CISC 命令は複数の IR に分解される．IR の文としては以下のような種類がある．

- NoOp
- IMark （元のネイティブの命令を保存するコメント）
- AbiHint （ABI に関するヒントで現在はスタックの redzoning のみに使用）
- Put, PutI （レジスタへの書き込み）
- WrTmp （一時変数への代入）
- Store （メモリへの書き込み）
- CAS (Compare and Swap)
- LLSC (Load-Linked, Store-Conditional)
- Dirty （任意の C 言語の関数呼び出し）
- MBE （fence などのメモリバスの操作）
- Exit （条件付きのジャンプ命令）

また，式としては以下のような種類がある．

- Get, GetI （レジスタ読み込み）
- RdTmp （一時変数の読み込み）
- Qop （式への四項演算子の適用）
- Triop （式への三項演算子の適用）
- Binop （式への二項演算子の適用）
- Unop （式への単項演算子の適用）
- Load （メモリ読み出し）
- Const （定数）
- CCall （副作用のない関数呼び出し）
- Mux0X （条件比較）

これらからなる IR 命令列を core が元のプログラムから生成した後，各ツールの処理関数によりこの IR 命令列を自由に改変し，更にそのあと最適化を経て実際の CPU 上で実行される．

以上のように Valgrind のツールは IR 命令列を自由に操作できる点で任意のプログラムに対して強力な実行時解析を行うことが可能ではあるが，システムコールはカーネル空間で実行されるために IR 命令列に変換することができない．そのため，Valgrind は read/write などのメモリへの読み書きを伴うシステムコールの呼び出しの前後にフックをもうけてツールの開発者がシステムコールにおけるメモリアクセスも対象に出来るようにしている．またメモリ確保などの mmap などのシステムコールや，malloc, free などのライブラリ関数を任意のものに置換することも可能であり，ツールの中でメモリに関するクライアントプログラムの操作を調べることも容易である．

また Valgrind でプログラムを実行する際に，通常では実行されるプログラムから Valgrind の動作

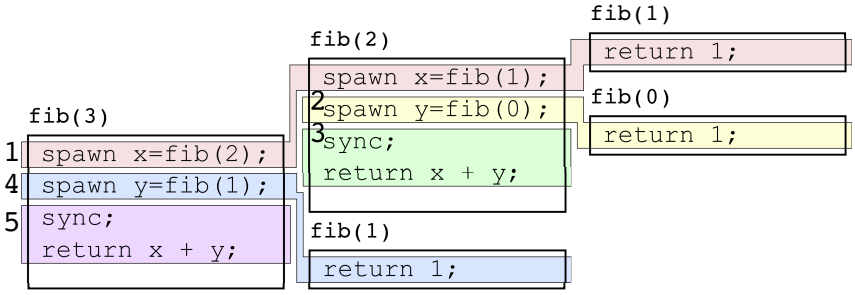

図 3.1. タスク関係の中のプロセス ID

を観測することはできないが，特殊なコード列を予め挿入しておくことで通常実行時は影響を与えず，Valgrind 上で実行した場合のみに影響を与えるようなコードを client 側に挿入することが可能である．これを client request と呼ぶ．

3.3 Mgascheck

本論文で提案する開発支援ツールの Mgascheck はこの Valgrind 上で記述されたツールである．Mgascheck は全てのメモリ確保・解放の際にその情報を記録し，メモリのアクセス時にその情報を参照・照合することでメモリのアクセスの妥当性を検証し，更に性能を推定するためにメモリアクセスのログを取得する．

3.3.1 Mgascheck におけるプロセスの仮定

先述したように，MassiveThreads/DM はそのタスクモデルとして migration point を指定している．この時，プログラムが migration point 以前と migration point 以後でタスクが同じプロセスで実行されると仮定したコードを記述している場合，タスクがマイグレーションした時に問題が生じてしまう．そこで，Mgascheck は migration point を通過する度に必ずプロセスが migration するものとしてプログラムの検証を行う．またこの仮定のもとでプログラムを実行した際の各プロセスを逐次化して id を振ることが出来る．以下ではこれをプロセス ID と呼ぶ．図 3.1 に単純なフィボナッチ数の計算を例にプロセス ID の変化を示す．

3.3.2 MassiveThreads/DM stub

Mgascheck では以下で説明する通り MassiveThreads/DM のインターフェース関数の実装の内部からも実行時に情報を受け取る必要がある．ここで MassiveThreads/DM 自体のランタイムに Mgascheck に対する client request を挿入することも可能ではあるが，現時点では MassiveThreads/DM の処理系は実装が完了しておらず，これに依存することができない．そのため，Mgascheck は Valgrind へ GAS の情報登録だけを行うスタブライブラリを同じインターフェースで用意し，こ

れにリンクをするようにコンパイルし直すことを現時点では要求している．

また，本来の MassiveThreads/DM では GAS 上のデータの実体と localize して得られるキャッシュメモリ領域は別の領域となっていてデータを書き戻すには明示的な `commit` が必要な実装となっているが，セマンティクス上は GAS 上のデータの実体のある領域が localize でそのまま渡されても 1 プロセスでは特に問題はない．そのため現時点では後者の実装をとっている．

また，Mgascheck は動作の正当性を確認するために MassiveThreads/DM のプログラミングモデルに従って動作するが，実行は逐次的に行われる．そのためタスクは実装としては通常の C 言語の関数呼び出しとして扱い，その前後にタスクの開始終了を伝える client request を追加している．

3.3.3　性能計測

Mgascheck はプログラムの正当性を検証するだけではなく，性能に関しての定量的な解析も提供することを目的としている．一般にプログラムの実行速度を制約するものとして CPU やメモリ，ノード間通信などが挙げられるが，現在のコンピュータアーキテクチャにおいては CPU に比べメモリさらにはノード間通信は十分とは言えない．そのため高い性能を得るために重要なのは高い計算通信比を実現することである．

MassiveThreads/DM を用いて実際にプログラムを実行した場合，各タスクはあるプロセスで実行され，更に負荷分散を取るために他のプロセスにマイグレーションする可能性がある．この時，MassiveThreads/DM では GAS 上のデータの配置を読み書きするタスクのところに移動するかそのままにするかという形で選択するため，GAS のデータの配置も実行時の負荷分散によって動的に決定される．そのため localize や commit の中で実際にノード間通信が発生するのか，それともノード内通信で完結するのかは負荷分散を行うタスクスケジューラの実装に依存する問題であり，更に実行時の状態に依存する．そのため並列度の小さい環境において大規模分散環境での性能を模擬的に評価するには困難な点が多い．

そのため，Mgascheck では図のようにタスクの階層構造の中である一定の深度以下のタスク全てが一つのプロセスでマイグレーションすることなく動作するという仮定をおいて，これらのプロセスで実行されるタスク内の通信量と命令数を模擬的に計量する．この際マイグレーションが無いという仮定からプロセス間での負荷分散は考慮されていない．しかし，負荷分散で行われるタスクの移動は全体から見れば小さいサイズのものであり，Mgascheck の仮定のもとで与えられる命令数，即ちタスク木の部分木の大きさと通信の量の関係には大きく影響しないと考えられ，ユーザが自身のプログラムが通信計算比の関係からどれくらいの規模までスケールするかを推定できる．

3.3.4　メモリアクセスの管理手法

メモリ管理

Mgascheck は Valgrind の機構を通して `malloc`, `realloc`, `free` などのメモリ管理の標準ライブラリ関数や `mmap`, `brk` などのシステムコール，スタックの伸縮やプログラムのロードなど，プログラム中でのすべてのメモリ確保・解放の際にアドレス空間中のどの領域が確保されたかを記録する．

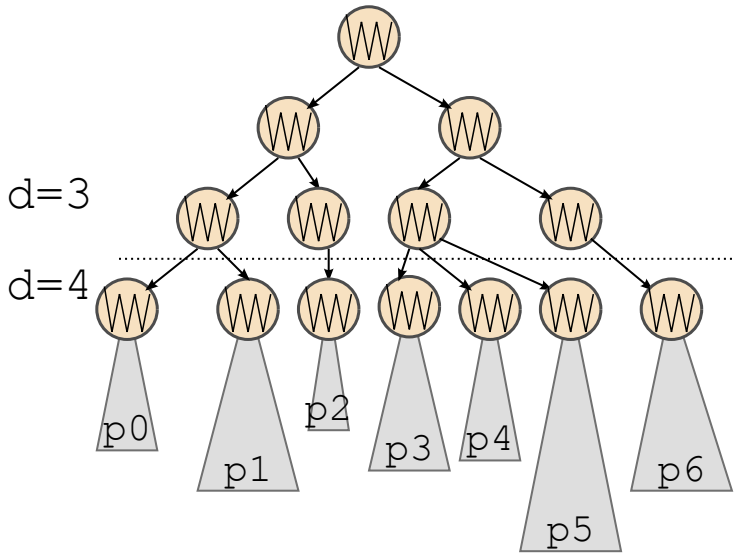

図 3.2. Mgascheck における通信推定時のプロセスモデルの仮定

同時にそのアドレス範囲がどのようにして確保されたのかを管理する．Mgascheck ではメモリ領域を以下のように分類する．

- 静的なメモリ
  Valgrind のプログラムローダによるコードの読み込み時やグローバル変数に対する分類である．
- スタック
- GAS
  GAS の実体の領域
- ローカルメモリ
  上で挙げたどれにも属さないメモリ領域．具体的には実行中に malloc や mmap を利用して確保されたローカルなメモリ領域が該当する．

この中で GAS の実体の確保は MassiveThreads/DM のスタブライブラリから通常のメモリ確保同様に `malloc` を用いて行われている．そのため Mgascheck からはローカルメモリとして確保した領域と区別がつかなくなる．そのためスタブライブラリでは GAS の領域を確保解放するときは client request を用いて GAS 上の領域を操作していることを Mgascheck 側に通知する．

3.3.5　バグの検出手法

メモリアクセス

Mgascheck は MassiveThreads/DM 上で記述されたプログラム中の全てのメモリアクセスを検査し，メモリに対するアクセスが守るべき制約を満たしているかを判断する必要がある．上に説明し

たように Valgrind がプログラムを変換して得られる IR 列を Mgascheck 内で利用可能な形で渡すので，Mgascheck はこの IR 列からメモリのアクセスがセマンティクスに合致しているかを検証する．IR 列にある文のうちデータに対して参照する文には Load 式が含まれメモリの load を発行する可能性があるので Load 式が含まれている場合チェックする．また，文のうちメモリに対して Store するものは Store 文, CAS 文, LLSC 文なのでこれらの文に対してはメモリの書き込みをチェックする．

次に load/store に対するチェックの手法を述べる．あるメモリアドレスに load/store する際のチェックはそのアドレスとサイズを用いて行う．まず，Mgascheck は対象となるメモリ領域がかつてのメモリ確保の際に記録されているかどうかを確認する．記録されていない場合そもそもその領域にアクセスすることができないはずのでエラーとして報告する．一方で記録済みの領域の場合はその領域の種類に応じて許されるセマンティクスが異なることになるため，以下でそれぞれの種類に応じて許可されるアクセスと許可されないアクセスの違いとその判別法を述べる．

- 静的なメモリ領域はプログラムのコードが記述された領域や，グローバル変数が存在する領域であるが，どちらも MassiveThreads/DM のメモリモデルでは読み込みは許可されるものの書き込みは許可されない．
- スタック領域は MassiveThreads/DM のプログラミングモデルでは各タスクの所有するスタック領域だけが使用が許されている．
  Mgascheck ではタスクが開始した際のスタックポインタを記録することでスタック領域へのアクセスが現在のタスクのスタックなのかどうかを判別することが出来る．
- GAS の実体の領域には MassiveThreads/DM のプログラミングモデルに基づくと localize や commit 内部以外でユーザがアクセスすることは禁止されている．ただし，Mgascheck で利用する MassiveThreads/DM stub の実装では localize が GAS の実体をそのままユーザに返すことにしている．そのため，Mgascheck では localize の際に localize した領域と，それを実行したプロセスの ID を記録している．そして，Mgascheck は GAS の実体へユーザがアクセスしようとしている際にこの localize の記録を参照して現在のプロセスで localize されている場合はアクセスを許可する．一方で過去に localize されていなかった場合はプログラム中で localize せず GAS のポインタを直接読み込んでいることが予想され，また localize の記録では現在とは異なるプロセス ID で localize されていた場合は migration point を通過する前に行った localize で得たポインタを利用していると考えられる．そのためこのような場合にはエラーとして報告する．
- ローカルなメモリ領域はヒープなど実行時に動的に確保するメモリ領域が挙げられる．これらのメモリ領域はメモリ確保時に確保した際のプロセス ID を記録しているので，アクセス時にはそのプロセス ID と現在のプロセス ID を比較する．一致している場合はアクセスを許可する．一致しない場合は migration point 以前のローカルメモリに対して参照しているのでエラーとして報告する．

```
1 int global_var;
2 void * x = malloc(...);
3 ...
4 // migration point
5 x[0] = ...; // <-- error!
6 global_var = ...; // <-- error
```

図 3.3. Detectable bug: heap and global variables

```
1 mgasptr_t task_foo(mgasptr_t g_arg) {
2     int x = 5; // stack variable
3     arg_t * l_arg = localize(g_arg, ...);
4     x += *(l_arg->x); // invalid access: parent stack.
5     l_arg->x = &x; // giving stack pointer to children tasks (errornous)
6     commit(g_arg, ...);
7     spawn task_foo(g_arg); // create children tasks
8     ...
9 }
```

図 3.4. Detectable bug: stack variables

ローカルメモリの不適切な使用

上で述べたようにマイグレーション前後を挟んでローカルメモリのアクセスを行うことはプログラムのセマンティクスに反している．そのため，Mgascheck は図 3.3 のようなマイグレーションを挟んだヒープ領域へのアクセスをバグとして検出する．

また，グローバル変数なども同様にローカル変数であり，読み込みは問題ないものの書き込みは実行されるノードに依存し結果が不定となるため禁止されるべきである．そのためこれらへの書き込みも禁止される．

スタックの不適切な使用

MassiveThreads/DM ではスタック領域に対しては通常のローカルメモリとは異なる例外的な扱いをする．タスクがマイグレーションするかどうかはユーザではなく処理系が決定する問題である．そのため，あるタスクのスタック上のデータはタスクと同時にマイグレーションしないといけない．そのためあるタスク内でそのタスクのスタックにアクセスすることは常に許可される．一方で，タスクがプロセス間でマイグレーションした場合に，先祖のタスクのスタックにアクセスすることは既にできないため図 3.4 のように他のタスクのスタックにアクセスすることは禁止されている．そこで，Mgascheck はスタック領域のうち現在のタスクの有するスタックのみにアクセスを許す事でスタックのチェックを行う．

```
1  globalptr_t gp;
2  int x = *gp; // cannot use global pointer as normal pointer
3  void * p = localize(...); // localize of gp
4  int y = *p; // OK
5  spawn foo();
6  y = *p; // cannot use localized pointer over migration point
7  p = localize(...); // you should re−localize
8  y = *p; // OK
9  unlocalize(gp);
10 x = *p; // cannot use localiezd pointer after unlocalize
```

図 3.5. 検出可能なバグ: GAS のセマンティクス違反

GAS のセマンティクス違反

MassiveThreads/DM GAS では *localize* を利用して大域アドレス空間へアクセスする必要があるが，任意の GAS のアドレスに対してアクセスする際に以下を満たす必要がある．

- ユーザは大域アドレス空間に対してユーザコードから読み書きすることはできない．
- ユーザは localize して得たローカルメモリ領域に対してマイグレーション地点を過ぎてからはアクセスしてはいけない．マイグレーション地点を過ぎた後は再度 localize する必要がある．
- ユーザは localize して得たローカルメモリ領域に対して unlocalize をした場合，再度 localize するまでその領域へアクセスしてはいけない．

図 3.5 にこれらに違反する例を示す．

3.4　実装

今回以上で説明したような手法を実装した．実装の基盤としては Valgrind の最新版[*2]をもとに実装を行なった．

3.4.1　兄弟タスク間の同期

現時点で Mgascheck は実行時にタスクを通常の関数呼び出しとして扱って実行している．そのため，タスクの生成と待ち合わせのみを利用して記述されている場合をエミュレーションすることはできるが，兄弟間で同期を利用したプログラムの記述はできない．

[*2] sha:5d0d52118210671d3eeff94fd3f5cc3807bd2a44

第 4 章

評価

4.1 正当性の検証

Mgascheck は正当性の検証が可能であることを検証するために先の章で述べた基本的な検出可能なバグを意図的に記述したプログラムを対象として実行した場合それぞれに対してエラーを正しく報告することを確認した.

4.2 通信性能の推定

4.2.1 検証結果の妥当性について

Mgascheck はプログラム中の localize インターフェースを利用した通信の量を推定するが，この推定した通信量が Mgascheck の仮定するモデルに適合している必要がある．そのため今回はいくつかの単純なアプリケーションにおいて MassiveThreads/DM を用いてプログラムを記述し，また通信量の見積りが適正であることを確認した.

アプリケーションの例:フィボナッチ数

まずタスクの生成のみからなるような単純なアプリケーションにおいて通信量の見積りを行うために，図 4.1 のような単純なフィボナッチ数の計算を考える．このプログラムは単純にフィボナッチを再帰的に求めるが，この際このタスクがプロセス間でマイグレーションすることなく実行されたと仮定すると，localize インターフェースの呼び出しは下のように整理される.

- 2 行目の localize は引数を受け取る際のアクセスだが，親の時点で既にキャッシュされているので migration していない限り実体とのアクセスは発生しない.
- 22 行目の結果を書き戻す commit は GAS へデータを書き込む．commit は常に実体への書き込みを伴うので，この commit で int の大きさ S_{int} だけの GAS の領域の実体へのアクセスが生じる.
- 9 行目の子供のタスクへ渡す引数の localize は領域に対する初めの localize なので大きさ

$2S_{\text{int}}$ だけ実体へのアクセスが生じる．

- 19 行目の localize は子タスクで更新された値を正しく読み込むことが必要なので `UPDATE` モードで localize している．そのため実体との通信が大きさ $2S_{\text{int}}$ だけ発生する．
- 20 行目の localize はプロセスがマイグレーションしていた場合実体を取得する通信が発生するが，プロセスが同一であった場合 GAS の実体とアクセスする必要はない．

以上からマイグレーションがない仮定のもとでは `fib` タスクでの GAS へのアクセスは $n = 0, 1$ のリーフとなるタスクでは $3S_{\text{int}}$ の，それ以外の場合は $5S_{\text{int}}$ の大きさだけ発生することが分かる．またリーフでないタスクの場合，子タスクで生じる通信の量を考慮する必要がある．そのため，F_n を計算するタスク `fib(n)` のサブタスク全体で行う GAS との通信量を $T(F_n)$ と表すと，

$$T(F_n) = \begin{cases} 0 & n = 0, 1 \\ 5S_{\text{int}} + T(F_{n-1}) + T(F_{n-2}) & n \geq 2 \end{cases} \tag{4.2.1}$$

$$= 5S_{\text{int}}(F_{n+1} - 1) \tag{4.2.2}$$

であることが分かる．ただし，マイグレーション後最初のタスクで 2 行目の localize の通信コストを引き受ける必要があるため，F_n を計算するタスクとそのサブタスクだけの通信コストは $5S_{\text{int}}(F_{n+1} - 1) + S_{\text{int}}$ である．

アプリケーションの例:parallel mergesort

次に，フィボナッチ数の例よりも意味があり，さらに通信量を解析的に求めうるアプリケーションとしてマージソートを例示する．このマージソートは S. G. Akl らにより提案されたもので [2]，マージの操作を分割統治的により高い並列度を実現している．今回はこの実装として Cilk の examples として配布されているものを利用した．Cilk の example においてこのマージソートは `cilksort` という関数と `cilkmerge` という関数の二つをタスクとして呼び出すことで記述されている．これらの関数は大体次のように動作する．

まず，`cilkmerge` はマージソートにおける merge 操作を並列に行うものである．

- `cilkmerge` はマージすべき配列 [low1, high1], [low2, high2] と結果の格納先 [lowdest . . .] が与えられる．
  - [low2, high2] が [low1, high1] よりも小さい領域であると仮定しても一般性を失わない．
- しきい値より小さい領域を扱う場合は逐次的に merge を行う．
- そうでない場合 [low1, high1] の中点に最も近い値の要素を [low2, high2] から選び，それぞれの前半領域と後半領域を並列にマージする．

アクセスする対象の領域は単純にマージの対象となる配列の大きさと，マージ結果を格納する配列の大きさの和である．

一方で `cilksort` はマージソートのマージ側であり，まず配列を 4 分割して再帰的にソートし，これらを 2 つずつテンポラリな領域にマージする．更にそののちに元の領域にマージする．これもソートする対象の配列の二倍の大きさの領域に対してアクセスを行う．

```
1  globalptr_t fib(globalptr_t arg) {
2      int * arg_ = localize(arg, ...); // localize argument
3      if(*arg_ < 2) {
4          // if *arg_ = 0 or 1, you only have to return it as it is.
5          return arg;
6      }
7      task_t childtasks[2];
8      globalptr_t childargs = global_malloc(2*sizeof(int));
9      int * childargs_ = localize(childargs, ...);
10     childargs_[0] = *arg_ - 1;
11     childargs_[1] = *arg_ - 2;
12     commit(childargs, ...);
13     childtasks[0] = task_create(fib, childargs + 0*sizeof(int));
14     childtasks[1] = task_create(fib, childargs + 1*sizeof(int));
15     task_join(childtask[0]);
16     task_join(childtask[1]);
17     // you should localize childargs again
18     // since it runs over some migration points.
19     childargs_ = localize(childargs, ..., UPDATE);
20     arg_ = localize(arg, ...);
21     arg_[0] = childargs[0] + childargs[1];
22     commit(arg, ...);
23     return arg;
24 }
```

図 4.1. Naive Fibonacci Computation with Task Parallelism

これらに基づいて通信を推定した．アプリケーションとしてはフィボナッチ数, cilksort と行列積を計算した．このうち密行列積と cilksort の結果をそれぞれ図 4.2 と図 4.3 に示す．

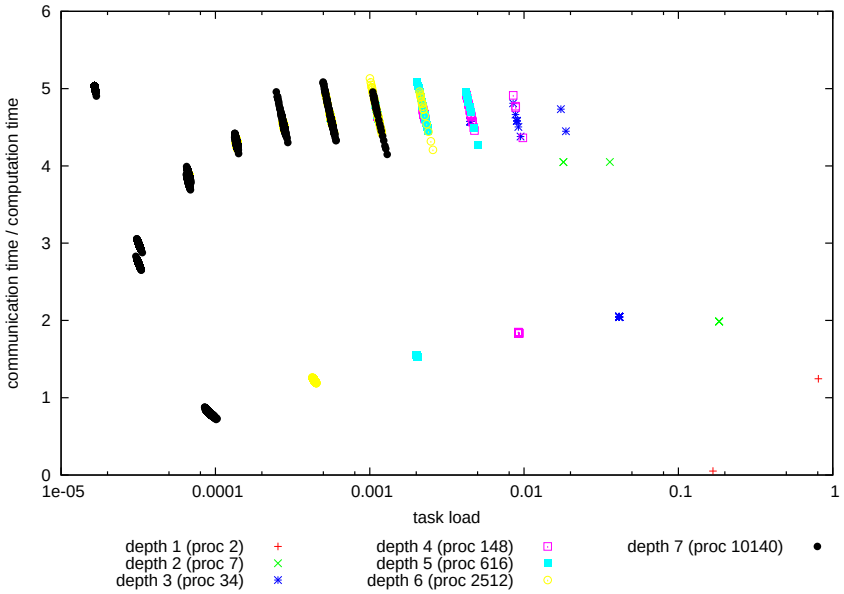

図 4.2. cilksort の通信計算比推定

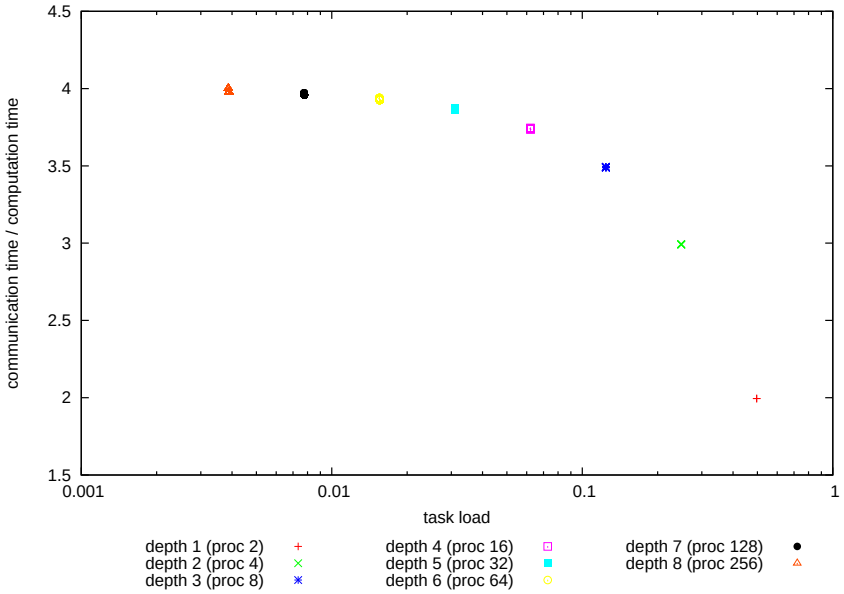

図 4.3. 密行列積の通信計算比推定

第 5 章

結論

5.1 まとめ

現在の分散並列プログラミングに比べてより高い生産性を持つプログラミングモデル MassiveThreads/DM の設計を提案し，またこれらに対してデバッガを提供することで生産性をより高くすることに成功した．また，通信量を見積もることで性能を予測するツールを開発した．

従来から分散並列プログラミングにおいては MPI という SPMD, 双方向通信という極めてプリミティブなプログラミングパラダイムが主流であり，計算機上でのプログラムの振る舞いを予測し最適化を行うのは容易ではあったが，一方でプログラムの記述そのものに通常の共有メモリ環境での逐次プログラミングとは大きく異なる配慮が必要であり，生産性が高いと言える状態ではなかった．

これに対して高い生産性を実現しようという試みはかねてからなされていたが，近年では Partitioned Global Address Space (PGAS) というプログラミングモデルがより平易なデータ共有と性能を両立するものとして注目されている．また，一方で共有メモリ環境における並列化手法として，動的な並列性を提供し負荷分散と低オーバーヘッドと記述性を両立した軽量タスク並列処理系とこれによる分割統治法がまた着目されている．

そこでこのタスク並列モデルと PGAS モデルを併用することで分散メモリ環境においても高い生産性と性能を両立することが可能であると考えられ，本研究ではこれを同時に実現した MassiveThreads/DM を設計した．MassiveThreads/DM では軽量タスク処理系に伴う細粒度な PGAS に対するアクセスの記述と，通信性能のための粗粒度な通信の両立を図るために localize インターフェースという新しいセマンティクスを PGAS に対して導入した．

また，これらのセマンティクスを採用したことで，通常の C 言語のライブラリで記述可能な範囲で，通信の集約と細粒度タスクを分散環境上で実現することに成功した MassiveThreads/DM だが，そのため通常の C 言語のプログラムにはないメモリモデルの制約が生じることになった．これらは再現性のないバグにつながる可能性があり開発中に確実に取り除けることが望ましい．また，大規模分散環境での実行を MassiveThreads/DM は視野においているが，このような環境での実行を行う機会は少なく，可能な限り事前に実行した場合の性能を予測し最適化を行うことが求められている．そのため筆者は MassiveThreads/DM 上で記述したプログラムがセマンティクスを順守しているか

を検証すると同時に，その通信の統計情報を実行時に抽出し，通常の共有メモリ環境上に於いて性能を模擬的に評価するツールを設計実装した．

このツールにより期待した PGAS のセマンティクスを違反したバグを検出可能であることが示された．また，基本的なアプリケーションを実行する際に通信量の見積もりが妥当な結果を得ていることを確認した．

5.2　今後の課題

今後の課題としてはまず MassiveThreads/DM 自体の処理系の完全な実装が急務である．Mgascheck は現時点で MassiveThreads/DM の処理系を利用することなくスタブライブラリを用意して MassiveThraeds/DM の代替となしているが，これを MassiveThreads/DM の実装と共通化することが可能であれば，より適切なプログラムの検証が可能であると考えられる．

一方で MassiveThreads/DM のプログラムの記述性を検証する必要がある．そのため MassiveThreads/DM でより大規模なアプリケーションを記述し，高い記述性を有することを示すことが必要である．

Mgascheck は現時点では PGAS に対する localize に特有なバグの検出に比重をおいており，通常の共有メモリプログラムを対象とした memcheck などと比較すると逐次部分のメモリバグの検出力が不足している．そのためこれらの共有メモリ環境用のメモリデバッガの手法を導入することが望ましい．

Mgascheck は性能に関して現時点ではスケジューラなどの実装に依存しない形での結果を出力しているが，もう一方で特定の実装が実際のプログラム上でどの程度の性能を発揮するかを評価する基盤が存在することが望ましい．そのため，スケジューラなどをモジュール化し，実際の並列実行と負荷分散をシミュレーションし実装間での性能の比較などを行う方法も有意義であると考えられる．

第 6 章

関連研究

6.1 タスク並列処理系

共有メモリ上では高い性能を得ることに成功しているタスク並列モデルであるが，分散環境に対しても多くの処理系が研究されている．Distributed Cilk [5] は Cilk を分散環境上に拡張した細粒度タスク並列処理系である．Cilk と同様にプロセス間にわたって負荷分散を行い，またタスク間の共有のためにソフトウェア分散共有メモリ (Software Distributed Shared Memory, SDSM) を提供しているが，この SDSM は通常のコンシステンシモデルとは異なり DAG コンシステンシと呼ばれるモデルを採用している．DAG コンシステンシでは兄弟タスク間でのデータの共有ができず，そのためタスクの同期も sync によってしか行うことができない．この制約に対して SilkRoad [19] や Cilk 自体の処理系 [15] では RC_dag consistency という lazy release consistency をあわせたモデルを採用している．そのため兄弟のタスク間でデータの共有やロックによる同期が可能になっている．

MassiveThreads/DM は PGAS モデルを採用するのに対し，これらの処理系は SDSM モデルを採用している．SDSM では PGAS と異なりユーザが大域アドレス空間へのポインタを通常のメモリ領域へのポインタと区別する必要はなく，どちらも通常の OS やハードウェアの提供する仮想アドレスの枠組みで処理される．そのため，ユーザからは透過的に DSM 上のデータにアクセスでき，データアクセスのためにライブラリ関数を呼び出したりする必要はなく生産性は PGAS モデルよりも高いと考えられる．

一方でその高すぎる透過性故にプログラマから DSM に対する通信の粒度を制御することは難しい．特に，メモリのページサイズが OS の提供する仮想メモリのページサイズと一致する必要があるが，一般的にはこのページサイズは 4KiB と小さな値であり，この程度細粒度にノード間通信を行うと十分な通信性能が得られず [3]，またユーザが自身で性能を最適化する余地が少なく高い性能を得られない．

Scioto[10] は PGAS 上のタスク並列プログラミングモデルを提供する言語である．Scioto は task collection というタスク実行モデルを持ち，SPMD モデルの中でそれぞれのプロセスが task collection にタスクを追加した後に collective にこの task collection を実行する関数を呼び出し，その関数中ではタスクの負荷分散などが Scioto のランタイムによって行われる．Scioto の PGAS

は Global Arrays [18] という PGAS ライブラリを用いている．Scioto と MassiveThreads/DM のプログラミングモデル上の違いは，Scioto では基本的な実行モデルが SPMD でその途中にタスクを実行するようになっているため，データの配置はユーザが明示的に行う必要がある．一方 MassiveThreads/DM ではデータの配置は localize インターフェースを利用してプログラムを記述することでコード上はどのプロセスに配置されるかを明記するのではなくどのタスク（の実行されるプロセス）に配置されるかを決めることになる．

X10 や Chapel は HPC を対象にした並列分散プログラミング言語であり，両者とも PGAS モデルの大域アドレス空間を提供し，更にタスク並列モデルの機能を備えている．しかし，それぞれタスクの負荷分散はノード内でしか行われず，ノード間でタスクを移動したい場合はそれぞれ `at`, `on` という実行されるノードを明示的に指定する記述を行う必要があり，タスク並列モデルの記述だけでノード間の負荷分散を行うことはできない．

6.2　開発支援ツール

まず共有メモリ上のプログラムに対してデバッグを支援するフレームワークとしては Valgrind 以外にもいくつかを挙げることができる．このうち，Low Level Virtual Machine (LLVM) [12] プロジェクトの一環として開発されているコンパイラフロントエンドである Clang [1] の機能として Address Sanitizer [22] が開発されている．Address Sanitizer は Valgrind のように実行時にコードを変換するのではなくコンパイル時に検査用のコードを埋め込むことで実行時のオーバヘッドを抑えている．またメモリの shadow を得る操作を Valgrind (memcheck) ではテーブルの検索というメモリの負荷が掛かる手法で実現しているが，Address Sanitizer では単純なビット演算を利用して shadow を得ることで性能の劣化を小さく抑えている．一方でこの手法はメモリの実体に対して shadow が小さいことを前提にしているので shadow として記録できる情報に限界がある．また，コンパイル時にコードを挿入するためプログラムの再コンパイルを行わないとチェックができない．更に Address Sanitizer 自体は Clang の機能として実装されているが，汎用的なフレームワークとして提供されているわけではないため，派生ツールの開発にはコストを要すると考えられる．

また，分散並列プログラミング言語に対して開発支援ツールを提供する研究もなされている．UPC-CHECK [9] は SPMD 型の PGAS モデル分散並列言語である UPC を対象とした開発支援ツールである．UPC-CHECK は UPC のランタイムに渡された引数が誤ったものでないか，UPC のプログラムが deadlock していないかを検知する．引数に関してはスレッド数やスレッドの index などが値域から逸脱していないかを検出することができる．

Parallel Performance Wizard (PPW) [24] は Hung-Hsun Su らの開発している PGAS プログラミングモデルに対する性能解析ツールである．Su らは GASP! [25] という PGAS の通信パターンを一般化して記録し性能解析を提供するインターフェースを提案しており，PPW もこのインターフェース上で構築される．Su らは PPW を UPC のプログラムを対象に実装し，実際の通信の状況を可視化したり，ノード間の負荷のバランスを可視化する機能が備わっている．また，より高い生産性を実現するために開発統合環境 (IDE) の一つである Eclipse との連携 [4] も提案されている．これ

と Mgascheck の違いとしては，まず PPW は localize のような実行時に通信が発生するかしないかが決定するようなプログラミングモデルを対象にしていない．また，PPW においてはプログラムの性能を計測するために実際の環境でプログラムを動作させる必要がある．最終的にはプログラムの最適化のために実際の環境での実行は必須となるが，Mgascheck は仮想的にローカルなマシンでも分散環境での性能を予測できる点で，可用性の低い高並列環境での実行を最小限に抑えることができる．

謝辞

まず当研究をするにあたって修士課程に渡りご指導いただいた田浦健次朗准教授に多大な感謝をしたいと思います．研究の方針もプロジェクト全体の方針や私の研究の方針などに強い影響を与えていただき，先生の言葉がなければこの論文の完成を提出することはなかったと断言できるくらい研究を引っ張っていただきました．

また，この MassiveThreads/DM の設計は先生だけではなく実際の実装を行なっている秋山茂樹先輩や共有メモリ版の MassiveThreads の開発者である中島潤先輩と始め研究室内でも再三の議論を行なって形が定まったものであり，これら先輩方にも謝意を表したいと思います．また，研究室の同期の河野瑛君，堀内美希さん始め，研究室のメンバ全員には日頃のミーティングや議論で助けられたことを感謝します．また既に卒業された研究室の先輩にも原健太朗先輩や加辺友也先輩を始め研究のことのみならずコンピュータ環境にまつわる広い知識をいただき結果として Linux 環境に対しかなりの知識を得ることができました．

また末筆ながら研究生活を送るにあたって度々無茶を強いても傍らで助けてくれた妻に心から感謝を捧げたいと思います．

発表文献

[1] 池上克明, 田浦健次朗. 並列科学技術計算に対して自動的な通信の集約を行う高い生産性を持つ言語の設計と実装. 並列/分散/協調処理に関するサマーワークショップ (SWoPP2011), 鹿児島, 2011/8.

[2] IPSJ-TPRO0501004, Katsuaki Ikegami and Kenjiro Taura. Design and Implementation of a High Productivity Language with Communication Aggregation for Parallel Scientific Computation. 情報処理学会プログラミング（PRO）Vol.5 No.1, 2012/3.

[3] 池上克明, 田浦健次朗. 分散メモリ環境上におけるタスク並列処理系 MassiveThreads/DM に対する共有メモリ環境上での模擬評価. 並列/分散/協調処理に関するサマーワークショップ (SWoPP2012), 鳥取, 2012/8.

参考文献

[1] clang: A C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, Vol. 36, No. 11, pp. 1367–1369, November 1987.

[3] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, p. 10 pp., April 2006.

[4] Max T. Billingsley, III, Beth R. Tibbitts, and Alan D. George. Improving UPC productivity via integrated development tools. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pp. 8:1–8:9, New York, NY, USA, 2010. ACM.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP '95 Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, October 1996. ISBN:0-89791-700-6.

[6] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. *IDA Center for Computing Schences*, May 1999.

[7] B.L. BL Chamberlain, David Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, Vol. 21, No. 3, pp. 1–24, August 2007.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 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*. ACM, October 2005.

[9] James Coyle, Indranil Roy, Marina Kraeva, and GlennR. Luecke. UPC-CHECK: a scalable tool for detecting run-time errors in unified parallel c. *Computer Science - Research and Development*, pp. 1–7, 2012.

[10] James Dinan, Sriram Krishnamoorthy, D Brian Larkins, Jarek Nieplocha, and P Sadayap-

pan. Scioto: A framework for global-view task parallelism. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pp. 586 – 593, 2008.

[11] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pp. 285 – 297, 1999.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, No. c, pp. 75–86, 2004.

[13] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pp. 36–43, New York, NY, USA, 2000. ACM.

[14] D.B. Loveman. High performance fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, Vol. 1, pp. 25–42, February 1993.

[15] Rafael Mendes, Lauro Whately, Maria Clicia de Castro, Cristiana Bentes, and Claudio Luis Amorim. Runtime system support for running applications with dynamic and asynchronous task parallelism in software DSM systems. In *Computer Architecture and High Performance Computing, 2006. SBAC-PAD '06. 18TH International Symposium on*, pp. 159 – 166, October 2006.

[16] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, p. 65, 2007.

[17] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, 2007.

[18] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94 Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, 1994. ISBN:0-8186-6605-6.

[19] L Peng, W.F. Wong, M.D. Feng, and C.K. Yuen. SilkRoad : A multithreaded runtime system with software distributed shared memory for SMP clusters. In *Cluster Computing, 2000. Proceedings. IEEE International Conference on*, pp. 243–249. IEEE, 2000.

[20] John Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1 – 31, August 1998.

[21] James Reinders. *Intel Threading Building Blocks Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007.

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[23] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. *USENIX Annual Technical Conference*, 2005.

[24] Hung-Hsun Su, Max Billingsley III, and Alan D. George. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8. Ieee, April 2008.

[25] Hung-Hsun Su, Dan Bonachea, Adam Leko, Hans Sherburne, III Billingsley, Max, and AlanD. George. Gasp! a standardized performance analysis tool interface for global address space programming models. In Bo Kgstrm, Erik Elmroth, Jack Dongarra, and Jerzy Waniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, Vol. 4699 of *Lecture Notes in Computer Science*, pp. 450–459. Springer Berlin Heidelberg, 2007.

[26] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/mp: integrating futures into calling standards. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pp. 60–71, New York, NY, USA, 1999. ACM.

[27] 中島潤, 田浦健次朗. 高効率な I/O と軽量性を両立させるマルチスレッド処理系. 情報処理学会論文誌プログラミング（PRO）, Vol. 4, No. 1, pp. 13–26, 2011.