

高効率な I/O 処理が可能な細粒度マルチスレッド 処理系の Chapel による評価

中 島 潤^{†1} 田 浦 健次朗^{†1}

高生産並列プログラミング処理系のランタイムには、抽象度の高いコードを高性能に実行できることが求められる。我々はその実装に用いることを想定した基盤ソフトウェアとして、I/O 多重化による高効率な I/O 処理と、細粒度なスレッドによる動的負荷分散を両立するマルチスレッド処理系、MassiveThreads を提案している。

MassiveThreads の基盤ソフトウェアとしての有効性を確認するため、高生産並列プログラミング言語である Chapel を題材とし、そのタスク管理を行うモジュールを MassiveThreads によって実装し、タスク作成のオーバーヘッドとスケールビリティ、およびノード間通信の性能評価を行った。

Evaluating I/O-efficient and Fine-grained Multithread Library by Chapel

JUN NAKASHIMA^{†1} and KENJIRO TAURA^{†1}

It is essential for the runtime of high-productivity parallel programming frameworks to execute highly abstract code as fast as possible. As a middleware for implementing them with high performance, we have proposed a multithread library called MassiveThreads. It can create many threads with little overhead, can balance computational loads dynamically, and can execute I/O operation efficiently by I/O multiplexing.

In order to confirm that MassiveThreads is useful for implementing the runtime, we implemented task module of Chapel by MassiveThreads. And we evaluated task creation overhead and inter-node communication throughput.

1. はじめに

MPI をはじめとする、現在普及している並列プログラミング処理系の多くは、並列計算機のハードウェアをそのまま抽象化し、並列計算機の各構成要素が行う断片的な処理を記述させるようなプログラミングモデルをとっている。このようなモデルは並列計算機のハードウェアについての知識を利用することで高性能なアプリケーションを記述することができる一方、断片的な処理を記述する必要があるためその記述は困難で、ソースコードの可読性も低い。また、今後並列計算機は GPU やアクセラレータによるノードのヘテロ化、NUMA アーキテクチャによるメモリの非均一化など、複雑性を増す方向に発展していくとみられており、ハードウェアをそのまま抽象化するようなプログラミングモデルでアプリケーションを記述することはますます困難になると考えられる。

そのため、より高水準にハードウェアを抽象化し、大域的な記述を用いることでより簡単に並列アプリケーションを記述できる、高生産な並列プログラミング処理系への要請が高まっており、それを受けて近年は多くの高生産並列プログラミング処理系が提案されている。

これまでの並列プログラミング処理系においては、アプリケーションの記述をハードウェアに関連付けるのは開発者の役割であった。しかしこれらの高生産並列プログラミング処理系においては、その高度に抽象化された記述を実際のハードウェアに関連付けて実行するのはランタイムの役割である。並列プログラミングを行うそもそもの動機は、ほとんどの場合並列実行によって高い実行性能を得ることにあるため、ランタイムの実行性能は処理系の実用性を左右する、極めて重要な要素であるといえる。

このような状況を踏まえ、我々は高生産並列プログラミング処理系のための基盤ソフトウェアとして、効率的な I/O 処理と軽量性を両立するスレッドライブラリ MassiveThreads を提案し、既存のスレッド処理系において用いられている手法を拡張して組み合わせた実装により先に挙げた要件を達成し¹⁾、さらに、適合格子細分化法による移流計算によって実アプリケーションによる評価を行なった²⁾。

我々は現在、高生産並列プログラミング処理系の基盤ソフトウェアとしての MassiveThreads の有効性を評価するために、Chapel のランタイムのタスク管理を担うモジュールを MassiveThreads によって実装し、実行性能を改善することに取り組んでいる。本稿ではその実装と、性能評価の結果について述べる。

本稿の構成

2 節では、MassiveThreads の概要と特徴について述べる。3 節では、Chapel とそのラン

^{†1} 東京大学
The University of Tokyo

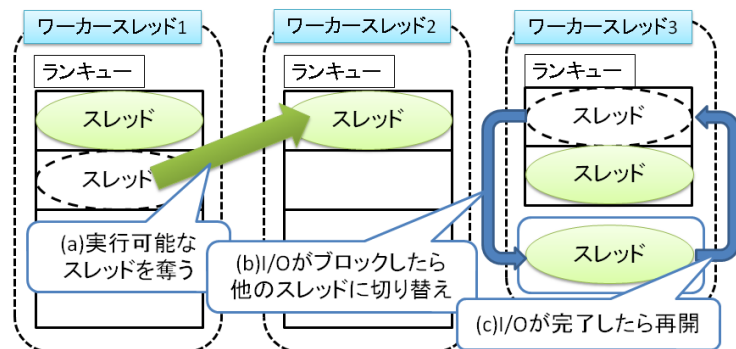


図 1 MassiveThreads の実装:ワークスチーリングと I/O 多重化
Fig. 1 MassiveThreads Implementation: Work Stealing and I/O Multiplexing

タイムについて概要を説明し、標準のタスク管理モジュールの実装とその問題点について述べる。4 節では、既存の軽量スレッドを実現した処理系についてふれ、MassiveThreads とそれらの差異について説明する。5 節では、MassiveThreads によるタスク管理モジュールの実装について述べる。6 節では、MassiveThreads によって実装したタスク管理モジュールのタスク作成の軽量性と I/O 性能を確かめるために行ったベンチマークと、その結果についての考察を述べる。最後に 7 節で、本稿の結論および今後の課題について述べる。

2. MassiveThreads の概要

MassiveThreads は高生産並列プログラミング処理系のランタイムに用いられることを主な用途として想定して設計されている軽量スレッドライブラリである。特に、タスク並列処理とタスクからの大域アドレス空間へのアクセスを高性能に実行することに着目し、スレッドの軽量性や動的負荷分散の性能と、I/O 多重化^{*1}による効率的な I/O 処理とを両立することを主な特徴としている。

MassiveThreads は各 CPU コアに OS レベルのスレッド (ワーカースレッド) を割り付け、各ワーカースレッドが実行可能なユーザースレッドを格納するランキューの中のスレッドを順次実行する、ユーザーレベルのマルチスレッド処理系として実装されている。LIFO

なスケジューリングと、アイドル状態のワーカースレッドが他のワーカースレッドをランダムに選択して実行可能なスレッドを奪うワークスチーリングを組み合わせることで、小さなオーバーヘッドでのスレッド作成・削除と計算ノード内のスケラブルな動的負荷分散を達成している (図 1(a))。このスケジューリング手法はスレッド間の公平性を保証しないが、特に再帰的にスレッドを生成する際に局所性や負荷分散の面で有利であることが知られている。それに加えて、I/O 呼び出しをフック、ブロックする可能性がある際には他のスレッドに制御を切り替え、定期的に I/O 可能になったかどうかをチェックして再実行することで I/O 多重化を実現している¹⁾(図 1(b,c))。

さらに、MassiveThreads 独自の API とは別に pthread と互換性のある API をあわせて、それを用いることで pthread を利用している既存の処理系やアプリケーションに対して、再コンパイルや移植の必要なしに組み込んで実行することが可能である。

3. Chapel の概要とタスクの実装

3.1 Chapel の概要

Chapel³⁾ は Cray によって開発が進められている高生産並列言語処理系で、並列計算機における高い生産性と、MPI に匹敵する実行性能の両立を目標としている。分散透明な大域アドレス空間を、計算ノードを抽象化した Locale という概念で分割し、それぞれの Locale でデータ並列・タスク並列の処理が行われるプログラミングモデルをとっている。また、高度に抽象化されたコードでプロトタイピングを行い、そこからインクリメンタルに性能最適化を行うことが可能なように設計されており、例えば配列の Locale へのマッピングなどの要素に関しては、定型的なものだけでなくユーザー定義のものを利用することができる。

Chapel のプログラムはトランスレータによって C 言語のコードに変換してコンパイルされ、ノード間通信、メモリ管理、タスク管理などを担当するモジュールとリンクして実行バイナリが生成される。各モジュールは独立性が高く、それぞれのモジュールとして何を用いるか (例えば、メモリ管理であれば標準 C ライブラリの malloc と dmalloc のどちらを使うか) の選択は処理系を構築する際に、動作環境にあわせて個別に行うことができる。

3.2 タスクとその実装

Chapel は並列性を表現するための多くの構文を定義しているが、これらはすべてコンパイル時にタスク並列処理の形に変換される。したがって、タスクの実行性能はアプリケーションの実行性能に大きく影響する。

Chapel で標準的に用いられるタスク管理のモジュールは図 2 のような、複数のワーカー

*1 I/O 呼び出しがブロックする際に他のスレッドに制御を切り替え、オーバーヘッドを隠蔽する手法

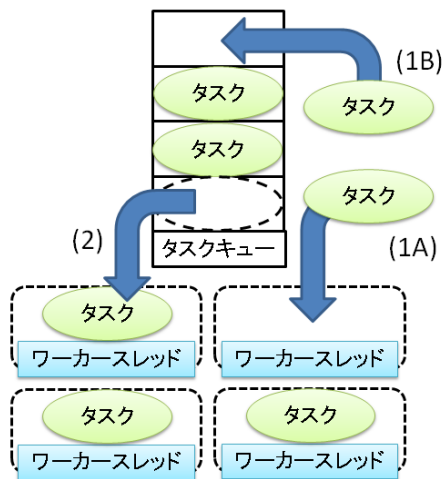


図 2 Chapel 標準のタスクの実装
Fig. 2 Chapel Default Task Implementation

スレッドと、全ワーカースレッドから共有されるタスクキューをもつ構成になっており、以下のようなスケジューリングポリシーで動作する。

- 新しいタスクが作成された際には、ワーカースレッド数が指定された制限に達するまではワーカースレッドを作成してそれに新しいタスクを割り当て、ワーカースレッド数が制限値に達しているならば、新しいタスクをタスクキューの末尾に格納する (図 2(1A,1B)).
- タスクの実行を終了したワーカースレッドはタスクキューの先頭からタスクを取り出してそれを実行する (図 2(2)).

問題点

この実装はループを CPU のコア数程度のタスクに分割するような、タスクの分割数が少なく、単純な並列処理ならば問題なく処理できる。しかし、分割統治法や木構造の探索などの、再帰的にタスクを作成するようなアプリケーションでは、以下のような性能上の問題が生じると考えられる。

- タスクキューをすべてのワーカースレッドが共有しており、操作に排他制御が必要なため、多くのワーカースレッドからのアクセスが集中し、タスクキューの操作がボトルネックとなる。

- FIFO なスケジューリングが行われるため、再帰的にタスクを作成すると最初にすべてのタスクが作成されてしまう。メモリなどの資源が圧迫されるだけでなく、一度確保した領域の再利用が困難であるため、タスク作成のオーバーヘッドそのものも大きくなる。

4. 関連研究

軽量なスレッドを用いたタスク並列処理系は古くから研究されており、Cilk⁴⁾をはじめとする多くの実装が存在している。しかし、ほとんどは 1 台の計算ノードですべての処理が完結することを前提として設計されており、それぞれの軽量スレッドが I/O を用いて他の計算ノードと協調し、分散処理を行うような動作は想定されていない。もし仮にそのような処理を行い、I/O がブロックした場合、I/O が完了するまでその CPU コアがブロックするため、CPU コアの利用効率が低下すると考えられる。

より軽量なタスクを提供するために Chapel のタスク管理モジュールを提供している既存の軽量スレッド処理系としては、Qthreads⁵⁾ と Nanos++⁶⁾ が存在しているが、これらも同様に複数の計算ノードを利用した処理を想定した実装はなされていない。また、これらを利用して複数の計算ノードを利用した実行を行うこともサポートされていない。

それに対して MassiveThreads は複数の計算ノードにまたがる並列計算全体の性能を高めることを前提に設計されており、I/O (=ノード間通信) がブロックした場合も他のスレッドに制御を移し、他の計算を行うことができる。

5. MassiveThreads によるタスク管理モジュールの実装

タスクの管理は MassiveThreads が行うため、タスク管理モジュールそのものの実装はほとんど MassiveThreads に対するラッパーである。しかし、実際はいくつか実装上の課題が存在しており、それらに対して以下のような対策を行っている。

5.1 GasNet 内部で行われる排他制御

ノード間通信モジュールが利用している GasNet⁷⁾ は、それを呼び出した pthread に対して排他制御を行う。したがって、MassiveThreads 固有の API を利用する場合は、GasNet の排他制御によってワーカースレッドがブロックしてしまい、CPU の利用効率が低下すると考えられる。

それを防止するため、環境変数 LD_PRELOAD *¹によってアプリケーション全体の pthread

*1 アプリケーションに対して、実行時に指定した共有ライブラリを読み込ませる環境変数

表 1 評価対象のタスク管理モジュール
Table 1 Task Modules Used for Evaluation

	スケジューリング方針	複数ノード対応
MassiveThreads	LIFO	○
Chapel 標準	FIFO	○
QThreads	FIFO	×

を置き換えることで MassiveThreads を組み込むものとした。この場合は GasNet によって行われる排他制御も MassiveThreads による軽量スレッドに対するものに置き換わり、ワーカースレッドがブロックする事態を回避できる。

ただし、これによってノード間通信モジュールがポーリングのために作成する pthread も MassiveThreads の軽量スレッドに置き換わってしまう。MassiveThreads のスケジューリングは LIFO をベースとしており公平性が保障されていないため、それが原因でデッドロックや実行性能の悪化が発生する可能性は否定できない。テストおよび性能評価ではそのような事態は発生しなかったが、ポーリング用のスレッドのスケジューリングに関しては検討の余地がある。

5.2 メモリ管理関数のスケーラビリティ

タスクの作成・破棄の際には、メモリ管理モジュールが提供する関数を利用してタスク間の同期をとるための変数が確保・解放される。しかし、現在 Chapel によって提供されているメモリ管理関数 (標準 C ライブラリの malloc と dlmalloc) はいずれも複数スレッドからの同時アクセスに対してあまりスケーラブルではなく、再帰的にタスクを作成する際に、メモリ確保がボトルネックになることが懸念される。

そこで、ワーカースレッド固有のフリーリストによって、スケーラブルにメモリを確保・解放できるような malloc 関数のラッパーを実装し、これによってメモリ管理モジュールの関数をフックしている。

6. 性能評価と考察

6.1 タスクの軽量性とスケーラビリティの評価

タスクの軽量性およびスケーラビリティを評価するために、以下のように再帰的にタスクを作成することによってフィボナッチ数を計算するベンチマークを、単一の計算ノードを用いて行った。このベンチマークにおいてはタスクあたりの計算量がごくわずかであるため、実行時間はタスクの作成・破棄に要する時間に支配される。

表 2 フィボナッチ数計算の評価条件
Table 2 Experimental Setup of Fibonacci Calculation

CPU	Opteron 8354(2.2GHz)×8 = 32 コア
OS	Linux 2.6.26(Debian GNU/Linux)
C コンパイラ	GCC 4.4.1
Chapel のバージョン	1.3.0
ノード間通信	利用しない
メモリ管理	C 標準ライブラリの malloc
計算する項	第 30 項

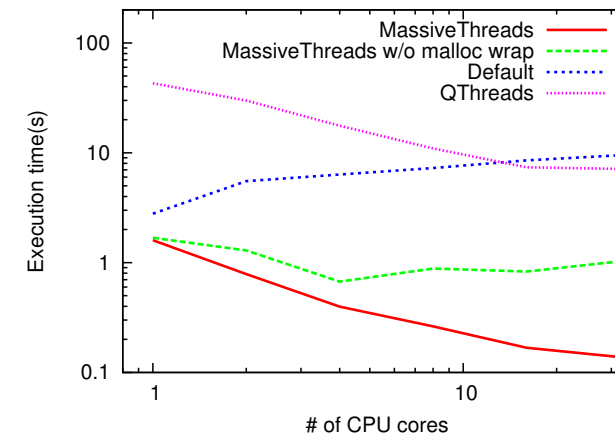


図 3 フィボナッチ数計算の実行時間
Fig. 3 Execution Time of Fibonacci Calculation

- (1) 各タスクは、開始時に引数として計算する項数 n を受け取る。 n が 0 か 1 なら n を返して即座に終了する。
- (2) 第 n 項を計算するタスクは第 $n-1$ 項と第 $n-2$ 項を計算するタスクを作成する。
- (3) さきほど作成した 2 つのタスクが終了するのを待ち、完了したらそれらの結果の和を戻り値として返す。

表 2 に示す評価条件において、MassiveThreads、MassiveThreads の malloc のフックを無効にしたもの、Chapel 標準のタスク管理モジュール、QThreads の 4 種類のタスク管理モジュールを比較した。その結果を図 3 に示す。横軸は使用した CPU コア数、縦軸は実行

表 3 RandomAccess の評価条件
Table 3 Experimental Setup of RandomAccess

CPU	Xeon E5530(2.4GHz)×2 = 8 コア (16 スレッド)
ネットワーク	10GbE
OS	Linux 2.6.26(Debian GNU/Linux)
C コンパイラ	MassiveThreads:GCC 4.4.1 Chapel:GCC 4.3.2
Chapel のバージョン	1.3.0
ノード間通信	GasNet
通信路	UDP
メモリ管理	C 標準ライブラリの malloc
分散配列のサイズ	8 バイト × 2 ²⁸ 個 = 2.0GB
合計更新回数	2 ¹⁸ 回

時間である。

MassiveThreads はすべての条件においてもっともよい結果を示している。これは、比較対象のスケジューリングポリシーがすべて FIFO であるため、タスクやスレッドのためのメモリを確保するオーバーヘッドが大きいものに対して、MassiveThreads は LIFO なスケジューリングを行うためメモリを再利用を効率的に行えているためであると考えられる。ただし、malloc のフックを無効にし、C 標準ライブラリの malloc をそのまま利用した場合はそれがボトルネックとなり、性能向上が 2.5 倍程度で頭打ちになってしまっている。

また、Chapel 標準のタスク処理モジュールは逐次では QThreads より高速であるものの、コア数が増加するにしたがって逆に計算時間が増加してしまっている。これは単一のタスクキューがすべてのワーカースレッドから共有されることによるものであると考えられる。

6.2 通信性能の評価

通信性能の評価には HPC Challenge Benchmarks⁸⁾ の RandomAccess を利用した。これは分散配列の中のランダムな要素を排他的論理和により更新する速度を計測するベンチマークで、結果はノード間通信の性能に支配される。

Chapel による実装としては処理系に添付されているもの⁹⁾ を用いた。これは図 4 のような、配列の要素が存在する Locale でタスクを起動し、そのタスクが要素を更新する実装になっているため、タスクの軽量性による性能向上もみられることが期待される。

表 3 に示す評価条件において、MassiveThreads、MassiveThreads の I/O 多重化を無効にしたもの、Chapel 標準のタスク管理モジュールの、3 種類を比較した。

2 つの Locale(=2 ノード) を利用し、ループを実行する並列度を変化させた場合の実行性

```
//更新対象の要素の集合を各 Locale に分割し、
//各 Locale はそれらに対して以下の処理を並列に行う
forall ( , r) in (Updates, RASstream()) do
  //更新対象の要素が存在する Locale で以下のタスクを実行
  on TableDist.idxToLocale(r & indexMask) do {
    //分散配列の値をローカルに更新
    const myR = r;
    local {
      T(myR & indexMask) ^= myR;
    }
  }
}
```

図 4 RandomAccess のメインループ
Fig. 4 Main Loop of RandomAccess

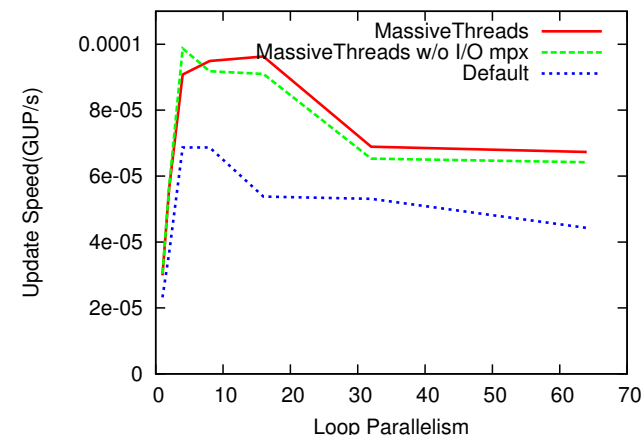


図 5 メインループの並列度を変化させた際の RandomAccess の結果
Fig. 5 RandomAccess Result v.s Loop Parallelism

能を図 5 に、ループの並列度を 8 に固定し、Locale 数を変化させた場合の実行性能を図 6 に示す。I/O 多重化を有効にするか否かにかかわらず、MassiveThreads を利用することによってスループットが向上している。しかし、I/O 多重化を有効にしても、ループの並列度が大きい(=多くのタスクが作成される)領域でごくわずかに性能が向上している程度で、並列度が CPU のコア数程度までは逆に性能が低下している。したがって、性能向上は主にタ

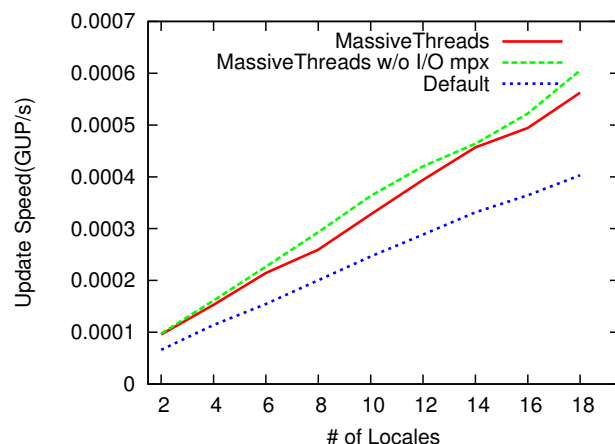


図 6 Locale の数を変化させた際の RandomAccess の結果
Fig.6 RandomAccess Result v.s Locales

スを作成する際のオーバーヘッドが小さくなったことに起因するものと考えられる。I/O 多重化の効果がみられない原因については現在調査を行っている。

7. おわりに

7.1 まとめ

本研究では、MassiveThreadsの有効性を検証するため、Chapelに対してMassiveThreadsを組み込み、再帰的な並列処理にも利用できるような、より軽量のタスクを実現した。

実装にあたっては、ノード間通信モジュールが利用しているGasNetが内部でpthreadに対して排他制御を行うため、それに配慮し、pthread互換APIを利用して全体で使われるスレッド処理系を置き換えることでMassiveThreadsを組み込み、さらにスケーラビリティに影響すると考えられるメモリ確保関数についてもフックして対策を行った。

性能評価の結果、再帰的にタスクを作成してフィボナッチ数列を計算するアプリケーションを他の既存の処理系と比較して高速に実行することができた。これにより、MassiveThreadsが、分割統治法のような再帰的なタスク作成を行うアプリケーションを効率的に並列実行できるランタイムを構築するのに有用であることが示されたと考えられる。

また、複数の計算ノードを用いたRandomAccessについても性能向上がみられたが、I/O

多重化のあるなしにかかわらず性能はほとんど変化しなかった。性能向上は、遠隔ノードでタスクを作成するオーバーヘッドが小さくなった効果に起因すると考えられるが、I/O 多重化による性能差がない原因については今後さらなる検証が必要である。

7.2 今後の課題

7.2.1 詳細な性能解析

MassiveThreadsを組み込むことによって、確かに性能向上がみられたが、特にRandomAccessについては、具体的にどのような要素が性能向上に寄与しているのか不明確であるため、さらに実験を行い、それを明らかにする必要がある。それと同時に、GasNetのポーリング用のスレッドなど、定期的に行われることが必要なスレッドがMassiveThreadsの公平性を保証しないスケジューリングにより悪影響を受けていないかについても検証する必要がある。

7.2.2 他のインターコネクトへの対応

現在MassiveThreadsがI/O 多重化を行えるのはソケットI/O に対してのみである。ここで、InfinibandやMyrinetなどのインターコネクトによるI/O 多重化にも対応することで、MassiveThreadsの実用性をさらに高められると考えられる。

参 考 文 献

- 1) 中島 潤, 田浦健次朗: 高効率なI/Oと軽量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).
- 2) 中島 潤, 田浦健次朗: 適合格子細分化法による細粒度マルチスレッドライブラリの評価 (2011). 先進的計算基盤システムシンポジウム SACSIS2011(ポスター発表).
- 3) Cray: The Chapel Parallel Programming Language. <http://chapel.cray.com>.
- 4) Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System, *SIGPLAN Not.*, Vol.30, No.8, pp.207-216 (1995).
- 5) Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *IPDPS*, IEEE, pp.1-8 (2008).
- 6) BSC: Nanos++. <http://pm.bsc.es/projects/nanos>.
- 7) Bonachea, D.: GASNet Specification, v1.1, Technical report, Berkeley, CA, USA (2002).
- 8) Dongarra, J. and Luszczek, P.: Introduction to the HPCChallenge Benchmark Suite, Technical report (2005).
- 9) Cray: HPC Challenge Benchmarks in Chapel. <http://chapel.cray.com/hpcc/hpcc09.pdf>.