Vol.2012-HPC-135 No.4
2012/8/1

スケジューリング方針をカスタマイズ可能な軽量スレッド処理系の提案

中島潤[1,a)]　中谷 翔[1,b)]　田浦健次朗[1,c)]

概要：分散メモリ環境でのタスク並列処理におけるタスクのスケジューリングについては，共有メモリ環境における Lazy Task Creation のような，多くの場合に効率的な戦略が明らかになっていない．また，特性が異なる共有メモリ環境の手法をそのまま用いるのでは不十分であることが考えられる．本論文では，共有メモリ環境を用いて分散メモリ環境に近い条件を再現してタスク並列処理による密行列積を実行し，その性能解析の結果から，共有メモリ環境で一般的に用いられるスケジューリング手法では粒度の小さいタスクの移動が繰り返されることにより性能が悪化しうることを示す．さらに，アプリケーションごとにスケジューリング方針をカスタマイズ可能な API と，それを用いた，タスクの再帰の深さに着目して粒度の小さいタスクの移動を抑制するスケジューリングの実装と評価について述べる．

キーワード：タスク並列処理，スレッド，スケジューリング，密行列積

Lightweight Thread Library that Can Customize the Scheduling Policy

Jun Nakashima[1,a)]　Sho Nakatani[1,b)]　Kenjiro Taura[1,c)]

Abstract: On distributed memory machines, efficient task scheduling strategy for task parallelism is not known. It may be insufficient to simply adopt the strategies on shared memory and it may occur performance degradation. In this paper, first we show Lazy Task Creation - generally efficient scheduling strategy on shared memory - may degrade performance on distributed memory due to too frequent movement of fine-grained tasks. Then, we describe APIs to customize scheduling stragegy in order to fit scheduling for applications, and show our experimental implementation and evaluation using this API, which intend to steal coarser-grained tasks by prioritizing tasks which have shallower recusion depths.

Keywords: Task Parallelism, Thread, Scheduling, Dense Matrix Multiply

1. はじめに

処理全体を細粒度な並列処理の単位であるタスクに分割し，それらを実行時システムがハードウェアに割り当てて並列実行するタスク並列処理は，再帰や多重ループなどの動的に並列度が変化する処理や，領域ごとの計算負荷が不均衡な処理の並列化を容易に行なうことができ，生産性と性能を両立できるプログラミングモデルとして多くのプログラミング言語や処理系に用いられている．

タスク並列処理は共有メモリ環境において用いられることが多い．Cilk[1] や Intel Threading Building Blocks[2] をはじめとする，現在普及しているタスク並列処理系のほとんどは共有メモリ環境を対象としたものである．また，近年活発に開発が進められている Chapel[3] や X10[4] といったいわゆる高生産プログラミング言語は，分散メモリ環境を対象とし，なおかつタスク並列処理をサポートしているが，これらも動的負荷分散の対象が計算ノード内に限局されており，実質的には共有メモリ環境のみを対象とし

[1] 東京大学
University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo 113–0033, Japan
[a)] nakashima@eidos.ic.i.u-tokyo.ac.jp
[b)] nakatani@eidos.ic.i.u-tokyo.ac.jp
[c)] tau@eidos.ic.i.u-tokyo.ac.jp

ているのと大差ない．

しかし，分散メモリ環境においても，タスク並列処理を用いることで高性能なアプリケーションを平易に記述することが可能になるため，その有用性は高いと考えられる．例えば，ノード間通信のレイテンシを他の処理とオーバーラップさせて隠蔽することは分散メモリ環境において台数効果を確保するために重要な実装手法であるが，タスク並列処理においては，十分に多くのタスクが生成されていれば，単なるタスクの切り替えによって暗黙的に達成される．また，ノードの構成がヘテロジニアスな環境では，ノード間に適切に負荷を分散することが重要であるが，タスク並列処理においては，動的負荷分散によってこれを自然に達成することができる．

タスク並列処理においては，タスクのスケジューリングをどのように行うかが処理系に一任されるため，高い実行性能を得るためには，動的な負荷分散や各タスクの参照局所性を活用することなどに留意してタスクのスケジューリングを行うことが処理系に求められる．共有メモリ環境においては，新しいタスクを作成する際に，作られた子タスクを優先して実行し，アイドル状態のスレッドは他のスレッドをランダムに選択し，そのスレッド内の最も再帰の深さが浅いタスクを奪う Lazy Task Creation[5] に基づくスケジューリングによって，これらをバランスよく達成できることがよく知られており，共有メモリ環境を対象とした実装の多くはこれをスケジューリング手法として用いている．

しかし，分散メモリ環境においては，ノード間でタスクを移動するオーバーヘッドが共有メモリ環境と比較して大きく，さらに，分散配置されたデータへのアクセスコストがアクセス元のノードによって大きく異なるという，共有メモリ環境と大きく異なる特徴が存在しており，これらの要素に配慮したタスクのスケジューリングが必要であると考えられる．

そこで本稿ではまず，分散メモリ環境に比較的近い条件でタスク並列処理による密行列積の性能評価を行なった結果を通して，共有メモリ上で行われるような，Lazy Task Creation に基づくスケジューリングではアプリケーションの参照局所性が悪化し，実行性能が失われる可能性があることを示す．次に，スケジューリング方針をカスタマイズするために追加した API と，それを用いて，タスクの再帰の深さに着目し，できるだけ粒度の高いタスクをワークスチーリングで奪うようにスケジューリングをカスタマイズした場合の，同様のアプリケーションの性能評価について述べる．

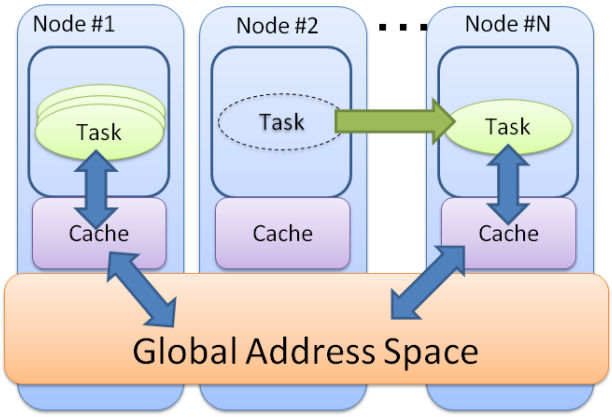

図 1 想定するプログラミングモデル
Fig. 1 Programming Model

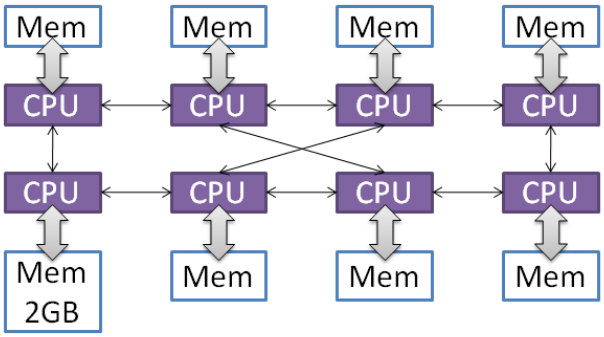

図 2 実験に使った計算ノードのメモリ構成
Fig. 2 Inter-socket Connections

2. 分散メモリ環境を想定した環境での密行列積の性能評価

2.1 前提とするプログラミングモデル

これは NUMA 構成の単一計算ノードにおけるタスク並列処理と類似した，以下のような特徴をもつプログラミングモデルを仮定する (**図 1**)．

(1) 各計算ノードは大域アドレス空間をもち，ローカルなキャッシュによって参照局所性を利用した通信オーバーヘッドの低減がなされる．
(2) 動作中のスレッドを停止させ，ノード間で移動させて再開することでノード間の動的負荷分散をとる．

また，このような分散メモリ環境を模倣する環境として，本実験では Quad-Core Opteron 8 ソケットからなる NUMA 構成の計算ノードを利用した．ソケット間のインターコネクトの構成は**図 2** のようになっている．また，ソフトウェア等の評価環境については**表 1** に示している．

2.2 密行列積計算の実装

密行列積の計算は，分割統治法により，計算領域のサイズが十分小さくなるまで再帰的に分割を行うことにより，キャッシュの構成によらずキャッシュの機能を利用できる

表 1　評価環境
Table 1　Experimental Setup

CPU	Opteron 8354(2.2GHz)×8 = 32 コア
メモリ	2GB × 8 = 16GB
キャッシュ	L1D: 64KB/core L2: 512KB/core L3: 2MB/socket
ピーク性能	563.2 GFLOPS (単精度)
OS	Linux 2.6.32(Debian GNU/Linux)
C コンパイラ	GCC 4.6.0

```
1  const int M = 32, N = 32, K = 32;
2  static void rec_matmul(float *A,float *B, float *C,
3    int m, int n, int p, int ld, int add)
4  {
5    if (m == M && n == N && p == K) {
6      //32*32*32まで分割されたら実際に計算
7      small_block_gemm(A, B, C, ld, add);
8      return;
9    }
10   if (m >= n && m >= p) {
11     //要素数が 32の倍数になるように分割
12     int m1 = DIVIDE(m);
13     //i 軸で分割して並列実行
14     spawn_thread(rec_matmul(A, B, C, m1, n, p, ld,
           add));
15     spawn_thread(rec_matmul(A + m1 * ld, B, C +
           m1 * ld, m - m1, n, p, ld, add));
16     sync_threads();//起動したスレッドの終了待ち
17   } else if (n >= m && n >= k) {
18     int n1 = DIVIDE(n);
19     //j 軸で分割して並列実行
20     spawn_thread(rec_matmul(A, B, C, m, n1, p, ld,
           add));
21     spawn_thread(rec_matmul(A, B + n1, C + n1, m,
           n - n1, p, ld, add));
22     sync_threads();//起動したスレッドの終了待ち
23   } else{
24     int p1 = DIVIDE(p);
25     //k 軸で分割して逐次に実行
26     rec_matmul(A, B, C, m, n, p1, ld, add);
27     rec_matmul(A + p1, B + p1 * ld, C, m, n, p - p1
           , ld, 1);
28   }
29 }
```

ソースコード 1　密行列積の疑似コード

Cache-Oblivious なアルゴリズム [6] として実装した．行列は 32 × 32 のブロックまで再帰的に分割され，SSE を用いて実装されたブロックの計算関数によって計算される．計算の手順を擬似コードで表現したものをソースコード 1 に示す．計算に用いる行列としては，32 までの 2 のべき乗数個ののタスクに行列を均等分割分割でき，なおかつある程度小さいサイズのものとして，768 × 768 要素の単精度浮動小数点からなる行列を用いた．

並列化には MassiveThreads[7] を用いた．MassiveThreads は Lazy Task Creation に基づいてスケジューリングされる軽量スレッドの他に，ブロッキング I/O が発生した際のスレッド切り替えや，pthread と互換性のある API を有しているが，本論文では単なる軽量スレッド処理系として用いている．

2.3　実行結果と性能解析

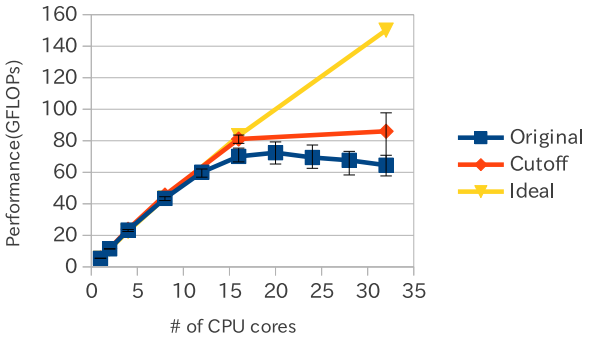

図 3　行列積の実行性能
Fig. 3　Matrix Multiply Performance

実行して得られた性能を**図 3** に示す．横軸は使用したコア数，縦軸は GFLOPs 単位の演算性能である．グラフ内の “Original” は実際に得られた演算性能を，“Cutoff” はタスクへの分割はコア数分にとどめ，それ以降は再帰関数によって逐次実行するようにした際の演算性能であり，理想的にタスクの動的負荷分散が行われた場合の性能とほぼ一致するといえる．“Ideal” は理想的に行列の分割が行われた場合に，各 CPU コアに割り当てられるのと同じ大きさの小行列による計算を 1 コアで実行して得られた演算性能に使用した CPU コア数を乗じたものである．

したがって，Original と Cutoff の性能差はタスクのスケジューリングが理想的に行われないことによる差異を，Cutoff と Ideal の性能差は複数のコア間で共有されているリソースなどの要因による性能低下を表しているといえる．Cutoff と Ideal は 16 コアまではほぼ一致しているが，32 コアすべてを利用する場合に大きな性能差がついてしまっている．また，Cutoff と Original を比較してみると，16 コア付近から性能向上が鈍くなり，20 コアの時点で性能が頭打ちになってそれ以上コア数を増やすと逆に実行性能が低下する．32 コア使用時には，Original は Cutoff の約 75%程度の性能しか達成できていない．特に 32 コアを利用した際に Cutoff と Ideal の間に大きな性能差が存在することは興味深い事実であるが，ここでは Cutoff と Original の間に存在する性能差について考察・検討を行う．

厳密にはこれらの性能差が MassiveThreads によるス

レッド作成のオーバーヘッドに由来する可能性も考えられる．しかし，本実験の評価のように再帰的にスレッドを生成し，なおかつ再帰の構造が均等である場合には，MassiveThreads はほぼ理想的なスケーラビリティをもつことが発表文献 [7] における性能評価で明らかになっている．そのため，使用する CPU コア数の増加に伴って MassiveThreads のオーバーヘッドが増加し，それがボトルネックになっている可能性は低いと考えられ，性能低下の主な理由は小さい粒度のタスクのワークスチーリングが多く行われたため，データの局所性を活用できていないことが原因だという仮説を立てることができる．

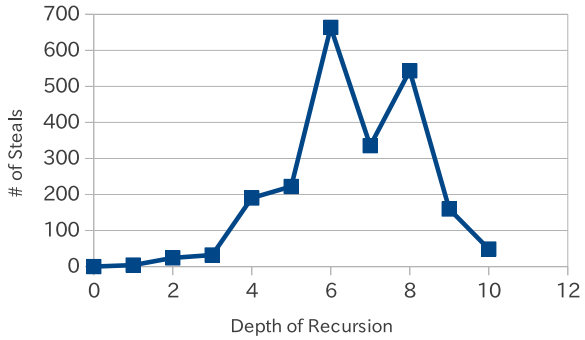

図 4 ワークスチールされたタスクの深さごとのヒストグラム

図 5 Histogram of Stolen Tasks by Depth

それを検証するため，32 コアを利用して行列積を計算した際に，ワークスチーリングが行われたタスクの粒度を測定し，そのヒストグラムを作成した．結果を**図 5** に示す．横軸はワークスチールが行われたタスクの粒度，縦軸はワークスチールが行われた回数である．理想的に負荷分散が行われた場合は，深さ 5 のタスクが全てのコアに行き渡ることになるため，ワークスチールが発生するのは深さ 5 のタスクを生成しうる，深さ 4 のタスクまでで，それ以降の深さではほとんどワークスチールは発生しないはずである．しかし，グラフを見ると明らかなように，実際には深さ 4 よりも細かい粒度のタスクに関しても高頻度にワークスチールが行われている．

3. スケジューラをカスタマイズする API

本章では MassiveThreads に追加した，スケジューラをカスタマイズする API の概要について述べる．この API は多様なスケジューリング方針を記述することのできる柔軟性を持たせることを意図して，ランキュー操作やスケジュール対象にならないスレッドの作成などの機能を提供するものである．

3.1 スレッド固有データの操作

スレッドをスケジューリングする際に利用可能なヒントとして，スレッド固有のデータを各スレッドに割り付けることが可能である．サイズは任意であるが，オーバーヘッドを小さくするため，スタック領域の末尾に領域が確保される．そのため，スタックサイズより大きな領域は確保できない．アクセスのための関数にスレッド作成の戻り値であるスレッド構造体を渡すと，この領域へのポインタが返され，任意に読み書きが可能である．ただし，複数ワーカースレッド間で共有して用いる場合，コンシステンシの確保は利用者が行う必要がある．

3.2 スケジュールされないスレッドの作成

MassiveThreads では，通常のスレッド作成関数によって作成されたスレッドは即座にスケジューラに投入され，Lazy Task Creation に基づいてスケジュールされる．しかし，動作する CPU コアを厳密に指定してい場合など，これを利用することが望ましくないスケジューリングが必要な場合も少なくないと考えられる．そのような場合に利用されることを想定した機能がスケジュールされないスレッドを作る関数である．例えばこれを利用して，作成したスレッドをワークスチール関数の戻り値として返すことにより，特定のワーカースレッドにスレッドを実行開始させることが可能である．

3.3 ワークスチール関数のオーバーライド

各ワーカースレッドがアイドル状態になっている際に実行される，ワークスチールを行う関数はワーカースレッドのインデックスを引数として受け取り，実行対象となるスレッドを返す機能をもつ任意のユーザー定義の関数で上書きすることが可能である．ランキュー操作関数と組み合わせて独自のワークスチール戦略を実装することを想定している．

3.4 ランキューの操作

各ワーカースレッドがもっているランキューに対して操作を行うことが可能である．実行中のスレッドが所属するワーカースレッドがもつランキューの先頭に対するスレッドの追加および取り出しと，および任意のワーカースレッドのランキュー末尾に対するスレッド追加と取り出し，そして任意のワーカースレッドのランキュー末尾に存在するスレッドの情報を取得する関数，の 5 種類が提供されている．

最後に示したスレッドの情報を取得する関数は，スレッドを取り出す場合と同様にスレッド構造体へのポインタを返すが，このスレッド構造体に対して許可されている操作は固有データの取得のみで，その情報が正確であることも保障されないが，その代わり排他制御を行うことなく実行可能である．これは複数のランキューの中からある条件を満たす一つのランキューをワークスチールの対象として選

択するような用途に用いられることを想定している．

4. タスクのスケジューリングのカスタマイズ

2.3 で得られた考察から，小さな粒度のタスクのワークスチールが頻繁に発生することを抑制できれば “Cutoff” に近い実行性能を達成できると考えられる．そのために，ワークスチールの際に複数の候補を選択し，その中から最も再帰の深さが浅いタスクを選択してスチールを行うことで，小さな粒度のワークスチールの頻度を削減することをねらいとして，前述の API を利用してスケジューリングのカスタマイズを行なった．具体的な実装は以下のように行われている．

4.1 スレッド固有データの利用

各タスクのスケジュールにはタスクの深さを利用することになる．これはワークスチール関数内からも参照する必要があるため，タスクを作成する際に，スレッド固有データとしてそのタスクの再帰の深さを格納する．

4.2 最も再帰の深さが浅いタスクの選択

ワークスチールの際に，ランキューの末尾のスレッドをロックなしで参照できる API を利用して取得し，スレッド固有データからタスクの深さを取得する．次にその結果から最も再帰の段数が浅い候補を特定し，それに対して実際にワークスチールを試行する．

4.3 性能評価

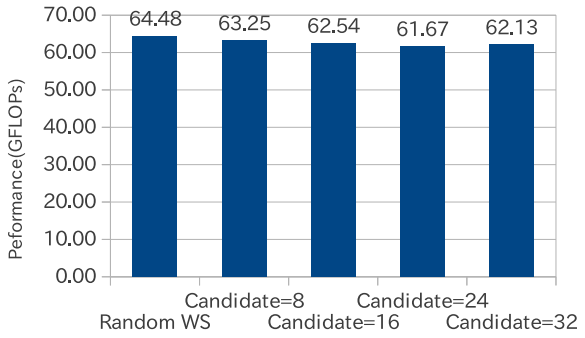

図 6 32 コア使用時の性能比較
Fig. 6 Performance Comparision on using 32 Cores

4 で示したスケジューリングのカスタマイズを適用した密行列積を 32 コアを利用して実行した際の性能を**図 6** に，その際にスチールされたタスクの深さのヒストグラムを**図 7** に示す．粒度の大きなタスクのスチールが増加することによる性能向上が期待されたが，実際にはランダムにワークスチールを行なった場合と比較してわずかに低下する結果となった．ワークスチーリングの回数についても，

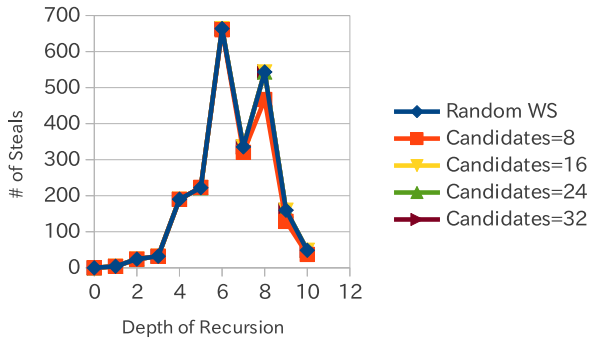

図 7 スチールされたタスクのヒストグラム
Fig. 7 Stolen Tasks Histogram

候補数が 8 の場合に深さ 8 のタスクのスチール回数が減少しているのが見て取れるが，それ以外についてはランダムにワークスチールを行うものと大きな差異はみられない．性能向上がみられない理由についての詳しい調査はまだ行えておらず，今後詳しい検討が必要である．

5. 関連研究

5.1 分散メモリ環境を対象としたタスク並列処理系

Cilk-NOW[8] はアイドル状態にある計算機にタスクを振り分けることで計算資源を有効活用することを目的にしたタスク並列処理系で，耐故障性や動的な計算ノードの参加・脱退にも配慮した設計が行われている．Tascell[9] は新しいタスクを生成する際に状態を保存せず，ワークスチールの際に状態を巻き戻すことで逐次にタスクを生成する際のオーバーヘッドを小さく抑えているタスク並列処理系である．これらの処理系は大域アドレス空間をもたず，各タスクが使用するデータを全てタスクを起動する際の引数として受け取り，計算結果を戻り値，あるいは次のタスクの引数として渡すプログラミングモデルを用いているため，タスクのスケジューリングは Lazy Task Creation とノード内，およびノード間の階層関係のみを考慮したワークスチーリングを用いて行なっている．

Scioto[10] は PGAS 処理系と組み合わせて動作するタスク並列処理系である．Scioto ではタスクを作成する際，それぞれのタスクに対して「作成されたノードでそのまま実行されることが望ましいか否か」をパラメータとして指定させ，そのまま実行されることが望ましいタスクはすぐに実行されやすいランキューの先頭に，そうでないタスクはランキューの末尾に挿入することで，データの局所性に配慮している．動的負荷分散はランダムに相手先を指定したワークスチーリングによって行われる．また，著者らは [11] において，ワークスチールの際に奪うタスクの数をランキューの末尾一つだけでなく，ランキューに入っているうちの半分にすることで，特に再帰構造が不均衡なアプ

リケーションでより良好な負荷分散を行う手法を提案している．

5.2 スケジューラにヒントを与えることが可能なタスク並列処理系

単純なワークスチーリングは，以前に別のタスクによってキャッシュにロードされたデータを考慮しない．したがって反復法のようなタスクの起動と合流を繰り返すアプリケーションでは，直前のイテレーションでキャッシュにロードされたデータを有効に利用することができない．それを改善するために提案されたスケジューリング手法が Locality-Guided Work Stealing[12] である．この手法では，各ワーカースレッドにランキューとは別に親和性のあるタスクを格納するデータ構造を用意し，ワークスチーリングの直前にメールボックスを確認して，その中に存在するタスクを優先して実行することで，できるだけ親和性のあるワーカースレッドでタスクを実行するものである．

6. まとめと今後の課題

本論文ではまず，分散メモリ環境におけるタスク並列処理を想定した，相対的に計算の割合が小さい条件でのタスク並列処理による密行列積について評価を行い，共有メモリ環境で用いられる Lazy Task Creation に基づくスケジューリングでは小さなタスクの移動が多くなることによって，データの参照局所性が損なわれ，性能が低下しうることを示した．さらに，アプリケーションがスケジューラをカスタマイズ可能にする API を提供することで，これを実際に評価を行なった．

今後の課題としては，まず今回実装した密行列積，およびスケジューラのカスタマイズについてより詳細な性能解析を行い，理想的な結果と実行性能が乖離している理由，および性能向上の余地を明らかにすることが挙げられる．また，今回提案した API は記述性よりも機能の柔軟性を重視したものが多かったが，スレッドの集合を全ワーカースレッドに分配する，などのより抽象度が高く，記述しやすい API の導入についても検討の余地があると考えている．

謝辞 本研究は，JST CREST「高性能・高生産性アプリケーションフレームワークによるポストペタスケール高性能計算の実現」の助成を得て行われた．

参考文献

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, Vol. 30, No. 8, pp. 207–216, 1995.

[2] Chuck Pheatt. Intel®Threading Building Blocks. *J. Comput. Small Coll.*, Vol. 23, No. 4, pp. 298–298, 2008.

[3] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *in Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pp. 52–60, 2004.

[4] 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*, pp. 519–538, New York, NY, USA, 2005. ACM.

[5] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.*, Vol. 2, No. 3, pp. 264–280, 1991.

[6] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pp. 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[7] 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, March 2011.

[8] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '97, pp. 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[9] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pp. 55–64, New York, NY, USA, 2009. ACM.

[10] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *ICPP*, pp. 586–593. IEEE Computer Society, 2008.

[11] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scalable work stealing. In *Proc. 21st Intl. Conference on Supercomputing (SC).*, pp. 14–20, 2009.

[12] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The Data Locality of Work Stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pp. 1–12, New York, NY, USA, 2000. ACM.