社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004−HPC−99 (36)
2004／8／1

メタ情報に基づくタスク並列スクリプト言語のスケジューラ

大野 和彦[†] 森 英一郎[†] 佐々木 敬泰[†]
近藤 利夫[†] 中島 浩[††]

我々はメガスケールの並列処理を想定したタスク並列スクリプト言語 MegaScript を開発している。この言語においてタスクを効率よく並列実行するには、各ホストの負荷バランスやホスト間通信量の削減などを考慮したスケジューリングが必要になる。

本研究では、メタモデルを用いたスケジューリング手法を提案する。メタモデルはユーザがタスクの挙動を記述したメタプログラムを元に生成され、さらに実行時引数やタスク数など動的に決まる情報も反映される。これによって静的・動的情報を一元的に扱うことができ、効率的なスケジューリング戦略を決定できる。

A Task Scheduler for a Parallel Script Language Using Meta-information

Kazuhiko Ohno,[†] Eiichirou Mori,[†] Takahiro Sasaki,[†]
Toshio Kondo[†] and Hiroshi Nakashima[††]

We are developing a task-parallel script language named *MegaScript* for mega-scale parallel processing. For the efficient execution of this language, tasks must be scheduled considering load balancing and minimization of inter-host messages.

So we propose a scheduling scheme based on *meta-model*. A meta-model is generated from meta-programs, in which the user describes behavior of tasks. Dynamic information such as command-line arguments and number of tasks is also reflected to the meta-model. This model enables to handle both static and dynamic information for efficient scheduling.

1. はじめに

我々は、メガスケールの並列処理を想定した並列スクリプト言語 MegaScript を開発している[1)]。

本言語では並列処理の単位となるタスクを大量に生成し、並行・並列に実行する。このため、システムを構成する各ホストに対しどのようにタスクを割り当て、さらにどのような順序で実行するかという、スケジューリングを考える必要がある。メガスケールの環境では性能の異なるホストやネットワークが混在し、空間的な制約から広域分散型になると考えられる。したがって、負荷バランスや通信コストを考慮しながら適切なスケジューリング戦略を決定することが、高性能を得るために不可欠である。

そこで本研究では、このような目的に適したスケジューリング手法を提案する。本手法は、静的に得られた情報を元にプログラムの挙動を表すメタモデルを構築し、これを参照しながら動的にスケジューリングを決定する。これにより、スケジューリング戦略にプログラムの性質を反映させることが可能であり、挙動が動的に変化する場合にも柔軟に対処できる。

以下、2 章で MegaScript について説明し、3 章で関連研究に触れる。4 章で提案する手法の基本方針を示し、5, 6 章でメタモデルとスケジューリング方式についてそれぞれ述べる。最後に 8 章でまとめを行う。

† 三重大学
Mie University
†† 豊橋技術科学大学
Toyohashi University of Technology

2. タスク並列スクリプト言語 MegaScript

2.1 言語の概要

メガスケールの並列性を持つプログラムを一から記述するのは非常に困難である。そこで我々は、部分問題を対象とする逐次／小規模並列プログラムを組み合わせることにより大規模並列性を引き出す「多重並列モデル」を採用している。

MegaScript はこの多重並列性の上位層を記述するための言語である。ユーザは下位層のプログラムをタスクとして定義し、その間をストリームと呼ばれる通信路で繋いだ、タスクネットワーク構造を記述する。

この層の実行に必要な計算量は全体に対してわずか

であるため、実行効率より記述のしやすさを優先し、Ruby[2) をベースとするスクリプト言語としている。

2.2 タ ス ク

タスクの実体は独立した外部プログラムであり、内部の処理に MegaScript は関与しない。スクリプト上ではタスクは Task クラスのオブジェクトとして表され、対応する外部プログラムのファイル名や実行時引数などをメンバとして持つ ☆。タスクの生成やスケジューリングは、このオブジェクトを操作して行う。

2.3 ストリーム

外部プログラムであるタスクの入出力として MegaScript からアクセスできるのは、標準入出力とエラー出力、実行時引数、プロセスの返り値だけである。そこで、標準入出力をタスク間通信に用いる。

ストリームはあるタスクの標準出力の内容を、他のタスクの標準入力に流し込むための通信路である。現在のバージョンでは通信内容はテキストのみサポートしており、改行文字を区切りとする 1 行ずつが 1 メッセージとして送受信される。

1 つのストリームに複数のタスクを接続することで、一対多、多対多などの通信を簡潔に実現することができる。入力側に複数のタスクを繋いだ場合、メッセージは行単位で非決定的にマージされる。出力側に複数のタスクを繋いだ場合は、メッセージはそれらのタスクにマルチキャストされる。

タスクと同様、ストリームは Stream クラスのオブジェクトとして表され、これとタスクオブジェクトを操作することでタスクネットワークを構築する。

2.4 プログラムの流れ

MegaScript による並列処理は、以下の流れで行う。

(1) new メソッドにより必要な個数のオブジェクトインスタンスを生成し、実行時引数などを設定する。new_array メソッドを用いるとタスク／ストリームオブジェクトの配列も作成できる。

(2) connect メソッドを用いて、タスク・ストリーム間の接続関係を設定する。

(3) create メソッドを用いて、タスクやストリームの生成を要求する。対象となったオブジェクトはスケジューリング待ちキューに移動する。

(4) schedule メソッドを呼び、スケジューリング待ちキューにあるオブジェクトすべてをスケジューリングする。スケジューリングされたオブジェクトは割り当てられたホストが空き次第、対応するタスクやストリームの実体が生成される。

☆ 実際には、ファイル名の設定や 2.5 節で述べるメタプログラムを記述するため、Task の継承クラスを定義して用いる。これらのクラスをタスククラスと総称する。

```
t1 = T1.new()
t2 = T2.new_array(M)
t3 = T3.new_array(M*N)
s1 = Stream.new()
s2 = Stream.new_array(M)
s1.connect(t1, IN)
s1.connect(t2, OUT)
for i in 0..M
  s2.connect(t2[i], IN)
  s2.connect(t3[i*N..(i+1)*N], OUT)
end
```

図 1 タスクネットワークの定義例

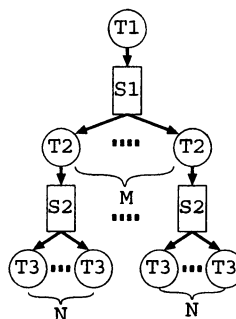

図 2 タスクネットワーク

実行の途中でタスクを追加するなど動的な生成を行いたい場合は、必要に応じて (1)～(4) を繰り返す。

MegaScript のプログラム例 (一部) を図 1 に挙げる。IN, OUT はストリームの入出力端のいずれに接続するかを指定する定数である。この記述により、図 2 のようなタスクネットワークが生成される。

2.5 メタプログラム

適切なタスクスケジューリングを行うには、実行時間などタスクの挙動をできるだけ正確に知る必要がある。ここでは、こうした情報をメタ情報と呼ぶ。

一般にメタ情報を得る手段として、タスクの静的解析や実行時トレースが用いられる。しかしタスクプログラムは任意の言語で書かれた外部プログラムなので、これらの手段が常に利用可能とは限らない。そこで、ユーザが直接メタ情報を記述できるようにしている。

スケジューリングに必要なのはタスクの計算や通信のコストであるが、これらの情報を導くにはプログラムの挙動を解析して評価する必要があり、ユーザの負担が大きい。そこで MegaScript では、タスクの振る舞いを抽象化したメタプログラムを記述する[3)。メタプログラムは元のプログラムに対し、ループや条件分岐といった制御構造を残してプログラムを抽象化し、部分コスト (の推定値) を記述したものである。

```
WHILE
  input(1)
  FOR @arg[0]
    compute(100)
  END
  output(10)
END
```

図 3 メタプログラムの記述例

例として図 3 に、図 1 の T2 のメタプログラムを挙げる。これは T2 に対応するプログラムが、標準入力からコスト 1 の入力が得られるたびに実行時引数で指定した回数だけコスト 100 の計算を行った後、コスト 10 の出力を行うことを表している。

メタプログラムはメタ情報の記述手段として高い表現力を持ち、元プログラムのソースが利用できる場合には作成も比較的容易である。一方で、ソースが存在しない場合やあまり労力を掛けたくない場合も、細部を省略してトップレベルの構造のみ記述するだけで、ある程度のメタ情報を得ることができる。このように状況に応じて手間と性能のトレードオフを取りやすい。

3. 関連研究

分散計算のタスクスケジューリングでは一般に、動的に状況が変化する非均質なホスト群に対し、タスクの計算量を考慮しながら動的負荷分散を行う。たとえば Ninf のメタサーバは収集したホストやネットワークの状況に基づいてタスクを割り当て、さらにタスク間の依存関係に基づいてスケジューリングする[4]。

これに対し GFarm[5] では、ペタスケールのファイルを並列処理することを想定しており、ファイルアクセスコストを小さくするようにスケジューリングする。

オーガニックジョブコントローラ (OJC) [6] は、動的なジョブの生成や待ち合わせをスクリプトとして記述するという我々に似た手法を用いており、データ依存や計算機の空き状況を考慮してジョブ投入が行われる。

一方、静的スケジューリング指向のものとしては、笠原らの粗粒度タスク並列処理[7] が挙げられる。この手法ではプログラムを粗粒度タスクに分割し、依存関係や並列性を解析した上で静的にプロセッサを割り当てるか、動的割り当てのコードを生成する。

4. 手法の概要

4.1 前提条件

(1) タスクは実行中にストリームを介して通信する。
(2) メタ情報により、タスクの粒度や通信量は部分的に判明している。
(3) タスクは MegaScript の実行を通して、動的に生成される。つまり、タスクネットワークの形状は動的に変化していく。

ジョブの終了・起動時にデータを受け渡す OJC などと比べると、(1) は通信コストを小さくするようなタスク配置が重要であることを示している。一方で、通信相手とのストリーム接続がスクリプト中で明示的に書かれているため、通信関係の把握は容易である。

(2) は、これらの情報を利用すると笠原らのような静的スケジューリングが可能であることを示しているが、情報の不完全さが前提であることから、動的負荷分散の併用が必要である。

また、メタプログラムからはタスク単位のメタ情報しか得られないため、(3) を考慮すると実際に作られたタスクネットワークの形状を把握し、これに基づいてスケジューリングを決定する必要がある。

話を簡単にするため、以下ではホストおよびネットワークの性能がすべて等しい均質環境とする。非均質環境への対応については 7 章で述べる。

4.2 スケジューリングの流れ

最初にメタ情報を元に、タスクネットワークのモデルを動的に構築する。これをメタモデルと呼ぶ。

あるタスク群についてスケジューリングを要求されると、このメタモデルを元に各々の初期配置および実行順序を決定する☆。

一般にメタモデルの情報は不完全であるため、各ホストでの実行が進むにつれて残りの仕事量に違いが生じる。そこで、適当なタイミングで再スケジューリングを行い負荷の均等化を図る。

5. メタモデル

メタプログラムより得られるメタ情報はタスク単位のものであるため、通信相手のタスクに依存する情報や、ストリーム上の通信量は含まれていない。そこでタスクネットワークの構造を再現し、その上で情報を伝搬させることで、全体の詳細なメタ情報を得る。

5.1 タスク単位のメタ情報

処理系はメタプログラムを解析し、計算コスト C_c および入力コスト C_i, 出力コスト C_o を得る。しかし、ループ回数などコスト値計算に必要な式には、実行時引数など静的に決まらない値が含まれることがある。そこで、静的にコスト値を求めるのではなく、実行時に決まるパラメータを引数とするコスト関数を生成する。このコスト関数は単純な計算式ではなく、メタプ

☆ ストリームについても配置場所を決定する必要があるが、現実装のランタイムではタスクに付随して生成されるため、本稿では省略する。

ログラムを元にしたRubyの関数として実現する。これにより分岐によるコストの増減なども容易に表せる。

実行時引数`@arg[i]`をa_iで表すと、タスクTの各コスト関数は以下の形になる。

(1) 入力コスト $C_i(T) = f(a_0, \ldots, a_{n-1})$

(2) 計算コスト $C_c(T) = f(a_0, \ldots, a_{n-1}, C_i(T))$

(3) 出力コスト $C_o(T) = f(a_0, \ldots, a_{n-1}, C_i(T))$

実行時に決まるコスト関数パラメータとして、現在は実行時引数のみ扱う☆。タスク内で自身が読み込むデータ量を決めている場合は、入力コストは実行時引数の関数として求まる。ファイル末尾まで読むようなコードの場合、入力コストはこのタスク単体では決定できず、値は不明になる。

計算コストや出力コストも基本的には実行時引数の関数だが、入力された個々のデータを処理していく場合など、入力コストに依存して決まるケースも多い。そこで両者を引数に持つ関数として定義する。

図3のメタプログラムの場合、C_iについては不明であり、C_cはWHILEループの回数を決める入力コストとFORループの回数を決める実行時引数に、C_oは入力コストのみに依存した値になる。よって以下のコスト関数が得られる。

$C_c(\mathrm{T2}) = 100 \times a_0(\mathrm{T2}) \times C_i(\mathrm{T2})$

$C_o(\mathrm{T2}) = 10 \times C_i(\mathrm{T2})$

5.2 メタタスクネットワークの構築

プログラム起動時に一度スクリプトを最後までスキャンし、記述されたネットワーク構造をモデル化する。具体的には、個々のタスク／ストリームオブジェクトに対しメタタスク／メタストリームオブジェクトを生成し、これらを連結したグラフ構造としてタスクネットワークのモデルを表現する。以下、このネットワーク構造をメタタスクネットワークと呼ぶ。

また、タスクを依存順にスケジューリングしやすいように、各メタタスクに対してデータフロー順位を決定しておく。

5.3 メタ情報の伝搬

ストリームSの通信コストは、S上を通過するメッセージの総量と考えることができる。送信側にタスク$T_0^s, \ldots, T_m^s$が、受信側にタスク$T_0^r, \ldots, T_n^r$が接続されているとき、通信コストは以下の式で表される。

$C_t(S) = C_o(T_0^s) + \ldots + C_o(T_m^s)$

また、受信側タスクT_i^rの入力コストが不明のときストリームから入力されるメッセージをすべて受け取

☆ 将来的にはタスクが参照するファイルの大きさや、ストリームを流れるデータの値も参照することを考えている。

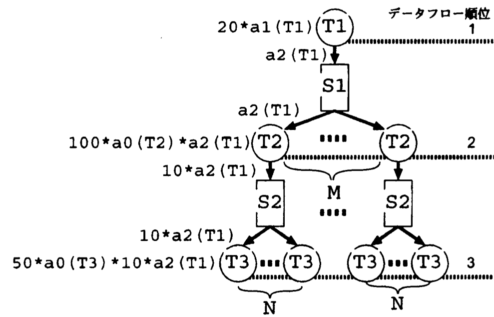

図4 メタモデルの例

るとみなすことにすると、$C_i(T_i^r) = C_t(S)$である。この関係を利用して、ストリーム通信の上流から下流に向かって入出力コストを伝搬させる。

たとえば図3のメタプログラムにおいて、5.1節に示したT2のコスト関数に加え、T1, T3のコスト関数が、以下のように得られたとする。

$C_c(\mathrm{T1}) = 20 \times a_1(\mathrm{T1})$

$C_o(\mathrm{T1}) = a_2(\mathrm{T1})$

$C_c(\mathrm{T3}) = 50 \times a_0(\mathrm{T3}) \times C_i(\mathrm{T3})$

このとき、T1からT3に向かって伝搬を行うと、以下のように決定できる(図4)。

$C_i(\mathrm{T2}) = a_2(\mathrm{T1})$

$C_c(\mathrm{T2}) = 100 \times a_0(\mathrm{T2}) \times a_2(\mathrm{T1})$

$C_o(\mathrm{T2}) = 10 \times a_2(\mathrm{T1})$

$C_i(\mathrm{T3}) = 10 \times a_2(\mathrm{T1})$

$C_c(\mathrm{T3}) = 50 \times a_0(\mathrm{T3}) \times 10 \times a_2(\mathrm{T1})$

5.4 メタモデルの動的補正

最初に作られたメタモデルは情報が不完全・不正確であるため、実行の過程で適時モデルの補正を行う。

コスト関数のパラメータである実行時引数のうちプログラム中で定数になっているものは、5.3節で述べた伝搬時に計算できる。一方、実行中に計算されて決まる引数については、最初の伝搬時には変数のまま残し、値が決定した時点で各コスト関数に反映させる。

また、生成するタスク数が動的に決まる場合や、条件分岐によりネットワーク構造の一部が変わる場合があるため、最初に作ったメタタスクネットワークは実際に生成されるタスクネットワークと一致するとは限らない。そこでスクリプトの実行開始後、タスク／ストリームの生成／接続指示のたびにモデルの構造と照合し、一致しない部分が生じたらモデルを修正する。

このように、本手法のメタモデルは動的に補正され、各々の時点では最新の予測モデルを表していると言える。これを用いることで、様々な静的・動的情報を一

元化してスケジューリングに役立てることができる。たとえばタスクの実行時トレースをメタモデルに反映させることで、メタプログラムが不完全なタスクの予測精度を高めることができる。

6. スケジューリング

6.1 スケジューリングの方針

(1) 各ホストの仕事量が同程度になるようにタスクを分配する。
(2) ノード間にまたがる通信量がなるべく少なくなるように、タスクを配置する。
(3) 各ホストでデータフロー順にタスクを実行する。

(1) に従うには各タスクの計算コストをメタモデルより求め、コスト値が均等になるように各ホストに分配すればよい。(3) についても、5.2 節で述べたデータフロー順位に従って実行すればよい。

(2) については、通信コストの大きいストリームがホスト間にまたがらないように、タスクの配置を決めればよい。しかし、(1), (2) を同時に満たす最適解を短時間で求めるのは困難であり、近似解法が必要になる。そこで本手法では互いの通信量が多いタスク群をタスクグループとしてまとめ、同じホストに配置する。

6.2 スケジューリングの手順

スケジューリングの際には、以下のことを決定する必要がある。

(1) 各タスク/ストリームを配置するホスト
(2) 各ホストに割り当てられたタスクの実行順序

本手法では、各ホスト毎に生成待ちキューを用意し、スケジューラが各ホストへの割り当て分のタスクを、各キューに格納する。各ホストのローカルスケジューラはキューの先頭から順にタスクを生成するので、キューへの格納順によって実行順序を制御できる。

以下、スケジューリングの手順を説明する。

6.2.1 タスクのクラスタリング

まず、通信コストの大きい順にストリームオブジェクトをソートする。続いて、ソートされた各ストリームに対して、その入出力端に繋がれたタスクすべてを一つのタスクグループとしていく。この処理は、以下の条件が共に満たされた時点で終了する。

(1) タスクグループ数と未処理のタスク数の総和が、閾値 TG_MAX を下回る。
(2) 対象となるストリームの通信コストが、閾値 CT_MAX を下回る。

条件 (1) により、組み合わせによって負荷の均質化が図れる程度の個数のタスクグループが形成される。条件 (2) は、並列実行の効果より通信コストが上回るような密結合のタスク群を強制的にグループ化する。

このクラスタリングが終了した後、以下を行う。

(1) グループ化されずに残っているタスクは、各々を1タスクからなるタスクグループとして扱う。
(2) 各グループ内のタスクについて計算コストの総和を求め、そのグループの計算コストとする。
(3) 各グループ内のタスクについて、データフロー順位に従ってソートする。また、タスクグループのデータフロー順位は、グループ内タスクの順位で最小のものとする。

6.2.2 タスクの分配

クラスタリングされたタスクグループを計算コストの大きい順にソートする。

続いてこれらのタスクグループを、各ホスト用の生成待ちキューに割り当てる。ここで、ある時点で生成待ちキューに格納されているタスクの計算コストの総和を、そのキューの計算コストと定義する。これは、キューに対応するホストに割り当てられた負荷のうち、まだ実行されていない部分の量を表している。

以下の手順により、各キューの計算コストがなるべく均等になるように割り当てることができる。

(1) 計算コスト最小のキューを選択し、ソート済みのタスクグループの先頭から一つ割り当てる。
(2) 割り当てたタスクの計算コストをキューの計算コストに加える。
(3) スケジューリング待ちのタスクグループがなくなるまで (1) から繰り返す。

その後、各キュー毎に、新規割り当て分のタスクグループをデータフロー順位に従ってソートする。

6.2.3 再スケジューリング

すべての計算コスト・通信コストが正確に判明している場合は、最初に上記のようなスケジューリングを行えば効率よく並列実行できる。しかし実際には、メタ情報が不完全でコスト値に誤差が生じたり、ホストの負荷が変化したりするため、ホストによってキュー上のタスクの処理進度に違いが生じる。

そこであるホストのキューが空になった時点で、各キューに残っているタスクを一度集め、6.2.2 項の手順を適用し直す。

たとえば図 4 の例で、実行時に $a_1(T1) = 10, a_2(T1) = 5$, $T2, T3$ については図の左から順に $a_0(T2) = 1..M, a_0(T3) = 1..N * M$ と決まったとする。クラスタリングを行うと、S2 のコストは S1 より大きいので、$C_t(S1) < CT_MAX$ かつ $M+1 < TG_MAX < M \times N + M + 1$ なら、個々の

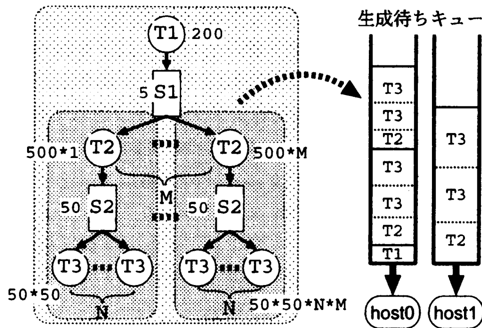

図5 スケジューリングの例

$S2$で接続されたT2とT3の集まりが、それぞれ1タスクグループになる。この結果、いずれかのホストで最初にT1を実行し、T2, T3のタスクグループは各ホストの計算量が均等になるように分配される (図5)。

7. 非均質な大規模分散環境への対応

7.1 非均質な実行環境への対応

タスクネットワークをモデル化したメタモデルと同様に、ホストネットワークをモデル化した実行環境モデルを導入する。各ホストやネットワークの性能に実測値を用い、実行中にも定期的に計測を行って実行環境モデルを補正することで、動的なホスト数やトラフィックの増減にも対応できる。スケジューリングの際には、この実行環境モデルを用いて計算コストや通信コストを補正する。

7.2 大規模なモデルの簡略化

現在のメタモデルは、実際に生成されるタスクネットワークを忠実に再現している。これは正確なスケジューリングが可能である反面、メガスケール環境ではメモリや操作コストの面で現実的でない。

実際には、メガスケールであってもモデルの大部分は同じ構造をしている。たとえばN要素のタスク配列を作成した場合、これらは同じタスクプログラムであり、多くの場合はNの大きさに関わりなく、マルチキャストや出力のマージなど、同じストリームを同じように接続する。このような場合は、メタタスクの配列をメタタスク1個に縮退して表すことができる。

7.3 マスターホストのボトルネック解消

現在の手法ではメタモデルやスケジューラがただ一つのマスターホスト上にあり、スレーブホスト数が増大するとマスターホストがボトルネックになる。

この問題を解決するには、サブマスターを設けてマスターホストの機能を部分的に委譲する。具体的には、マスターはサブマスターの台数に応じてタスクをクラスタリングし、粗粒度のタスクグループをスケジューリングする。サブマスターは自分に割り当てられたタスクグループをより細かく分解し、配下のホストに割り当てる。この場合、相互距離が短く通信コストが小さいと思われる下層のホスト群に対し、相互通信コストの大きいタスク群が割り当てられる。

8. おわりに

本稿では、並列スクリプト言語MegaScript用のタスクスケジューリング手法について述べた。本手法では、静的・動的に得られた情報を統合したメタモデルを用いてスケジューリング戦略を決定する。このため、タスクの挙動について十分な静的情報が得られる場合に効率的な実行が可能である一方、不完全な情報や実行環境の変化などに対しても、動的に対処できる。

現在、MegaScript処理系上にこの手法を実装中であり、完成後に実プログラムを用いた評価を行っていきたい。また、7章で述べた拡張を行って大規模な広域分散環境へ対応することも、今後の課題である。

謝辞 本研究は、科学技術振興事業団・戦略的基礎研究「低電力化とモデリング技術によるメガスケールコンピューティング」による。

参考文献

1) 大塚保紀, 深野佑公, 西里一史, 大野和彦, 中島浩: タスク並列スクリプト言語MegaScriptの構想, 先進的計算基盤システムシンポジウムSACSIS2003, pp. 73–76 (2003).
2) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語Ruby, ASCII (1999).
3) 大塚保紀, 大野和彦, 中島浩: タスク並列スクリプト言語MegaScriptによるタスク動作モデル記述, 情処研報 2003-HPC-95, pp. 113–118 (2003).
4) 中田秀基, 竹房あつ子, 松岡聡, 佐藤三久, 関口智嗣: グローバルコンピューティングのためのスケジューリングフレームワーク, 並列処理シンポジウムJSPP'99, pp. 277–284 (1999).
5) 建部修見, 森田洋平, 松岡聡, 関口智嗣, 曽田哲之: ペタバイトスケールデータインテンシブコンピューティングのためのGrid Datafarmアーキテクチャ, 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. 43, No. SIG 6(HPS 5), pp. 184–195 (2002).
6) 上田晴康, 吉田武俊, 安里彰: ジョブ投入と待ち合わせの出来るジョブ制御スクリプト: オーガニックジョブコントローラの試作, 情処研報 2003-OS-94, pp. 99–106 (2003).
7) 笠原博徳, 小幡元樹, 石坂和久: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理, 情報処理学会論文誌, Vol. 42, No. 4 (2001).