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

2007－ARC－172 (36)
2007－HPC－109 (36)
2007／3／2

相互影響のあるタスク向けグリッド環境の提案

田中義久[†] 村田忠彦[†,††] 蟻川 浩[††]

パラメータスイープアプリケーションやデータ並列アプリケーションに代表されるような並列アプリケーションはヘテロジニアスなグリッド環境に適しているが，メッセージパッシングを必要とするアプリケーションはヘテロジニアスなグリッド環境に適していない．このことに着目し，我々はヘテロジニアスなグリッド環境に適したライブラリの開発を行っている．本稿では，相互影響のあるタスク向けグリッド環境を提案する．

A Grid Environment That Considers Influence of Each Tasks

YOSHIHISA TANAKA ,[†] TADAHIKO MURATA [†,††]
and HIROSHI ARIKAWA [††]

Generally, the embarrassingly parallel applicaiton, such as parameter-sweep and data parallel application, is suitable for the Grid environment. On the other hand, the application with message passing is especially unsuitable for the heterogeneous Grid environment. We have been developing some grid-enabled libraries for the heterogeneous Grid environment. In this paper, we propose a new grid environment that considers influence of each tasks. Our proposed environment can be computed the distributed application without stopping the all tasks by using an traditional grid environment.

1. はじめに

ネットワーク上に広域に分散した複数の計算資源を統合し，仮想的に1つの計算機として利用するグリッドが普及しつつある．グリッド環境で実行されるアプリケーションの多くは，独立に計算できる箇所を抽出し，それをタスクとして複数の計算資源で分散処理することで高速処理を実現している．タスク間の独立性が高い代表的な例としてはパラメータスイープアプリケーション（PSA）があり，グリッド環境の多くはPSAを解くために利用されている．特にPSAの代表的事例であるSETI@home[1] は企業や大学内にある一般のPCの遊休状態を利用することで大規模グリッドを安価に実現しており，費用対効果の高いグリッド環境で高速処理を実現している．このように，計算資源の遊休時間を利用するグリッドのことを一般的にデスクトップグリッドと呼び，近年注目されている．

ところで，デスクトップグリッドはPCクラスタ等で構築されたグリッドと異なり，複雑なヘテロジニアス環境である．一般的にグリッド環境には各計算機の潜在的な処理能力の差が存在するが，デスクトップグリッドの場合，本来の利用者が発生させたハードウェアやOSなどへの割り込みが各計算機の単位時間あたりの処理能力に影響を与えるため，同一性能の計算機で構成した環境であってもヘテロジニアスな環境になる．

各計算機の潜在的な処理能力の差を考慮に入れた研究は既にいくつかある[2)~5)]．これらの研究では，前提として各計算機の遊休時間を見積ることが可能であること，見積もられた遊休時間をもとにタスクスケジューリングが行われていることが挙げられる．

これらの研究で提案されている手法をデスクトップグリッドに適用する場合，各計算機の利用者の振る舞いを特定することが困難なため，タスクスケジューリングによる処理能力の向上を見込めない．また，デスクトップグリッドの場合，利用者の割り込みが長期化する可能性を排除することはできないため，割り込みの長期化によりシミュレーション全体の単位時間あたりの処理量が低下する可能性がある．特にタスク間で情報を共有しながら並列処理するシミュレーションの場合，タスクが停止状態となるが，デスクトップグリッドにおいてこれを回避する方法について研究が進んでいない．

我々は遊休時間の見積りが困難であることを前提に，既存グリッドミドルウェアとの親和性を意識したグリッド環境の構築について研究を進めている．本稿では，既存のタスクスケジューラを用いたデスクトップグリッド環境を対象として，相互影響のあるタスク

† 関西大学 総合情報学部
Faculty of Informatics, Kansai University
†† 関西大学 政策グリッドコンピューティング実験センター
Policy Grid Computing Laboratory, Kansai University

をもつシミュレーションにオーバーラップ機能を付加することで，タスクの停止状態を回避する手法を提案する．また，プロトタイプシステムによる実験を行い，提案手法の有効性について議論する．

2. 用語の定義

本稿では，相互影響のあるタスクを“タスク間に特定の依存関係があり，依存関係に基づいて任意のタスクと情報を交換し，他タスクの処理状況に影響を受けつつ計算を行うタスク集合”と定義し，タスク間の依存関係により次の3つに大別する．

並列同期タスク　分散実行中の各タスクがある一定の条件を満たした時，同時刻に実行されている任意のタスクと特定の情報を交換し，それにより得られた情報に基づき計算を行うタスク集合．本稿では，特にこの並列同期タスクの効率化について取り扱う．

連続タスク　タスク間に先行制約による実行順序が定められており，先行タスクの計算結果が後続するタスクの計算に影響を及ぼすタスク集合．

並列同期連続タスク　並列同期タスク間に先行制約による実行順序が定められており，先行タスクの計算結果が後続するタスクの計算に影響を及ぼすタスク集合．

3. 並列同期タスク分散化の問題

分散実行中の各タスクは同期を取るために通信を行うが，このためには実行中の全タスクが通信可能状態になるまで待機する必要がある．ヘテロジニアスな環境では，計算資源の性能差によりタスク実行時間は不均一になり，またホモジニアスな環境においても，各タスクの計算負荷の相違によりタスク実行時間は不均一なものとなる．

各タスクの実行時間は本質的に不均一であるが，一般に並列同期タスクの分散処理を考える際には各タスクの実行時間は本質的に均一なものと仮定されており，現実には最も処理速度の遅いタスクが全体の終了時間を支配する（図1，node A や node B）．また，図1の node D のように同期待ち状態にある計算資源は有効に利用されているとはいえず，タスク処理に用いる計算資源の利用効率は低下する．

同期回数が増えるとともに通信時間は線形に増大するため，計算資源の利用効率はさらに低下する．これらは並列化にともなう固定的なオーバーヘッドであり，特に密結合なタスクにより構成されるタスクで並列化の効果が得られにくい要因となる．

4. 相互影響のあるタスク向けグリッド環境の提案

相互影響のあるタスクを既存のグリッド環境で効率

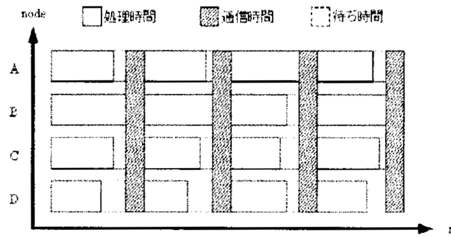

図1　同期実行時のタスクフロー

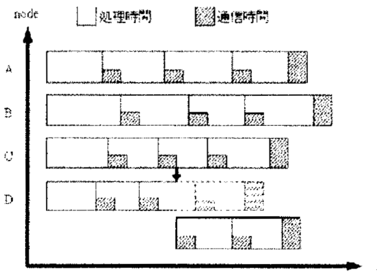

図2　非同期実行時のタスクフロー

的に分散処理する枠組みは現在のところ有効なものは存在しないので，これを実現する意義は大きい．そこで，タスク間の依存関係を隠蔽することにより，既存のグリッド環境を利用した相互影響のあるタスクの分散処理を実現するシステムを提案する．

4.1 概　要

図2に本提案手法におけるタスクの流れを示す．分散実行中のタスクが通信によりデータの同期を取る際，必要な情報を送信して受信を待たずに処理を継続する．この同期を取る箇所を同期ポイントと呼ぶ（図2縦線）．また，同期ポイントで処理に必要な情報のスナップショットを履歴として保持する．各タスクは常に自分宛にメッセージが届いていないかを確認し，送信元の同期ポイントと最後に処理した同期ポイントを比較し，必要であれば該当ポイントにロールバックした上で受信した情報を適用し，処理を継続する．以上を全てのタスクが終了するまで繰り返す．図中の node D の点線部分は本来の node D のタスクの流れを表している．node D は矢印の箇所で node C から2番目の同期ポイントの更新情報を受け取り，ロールバックを行った上で2番目の同期ポイントから再計算を行っている．

本提案手法は次の2つの要素で構成される．

監視ノード　全体の処理を統括する計算機である．全ての計算ノードの位置と，各計算ノードで実行中のタスクの依存関係を把握しており，任意の計算ノードから受け取ったメッセージを影響が及ぶ全ての計算ノードに伝達する．また，必要に応じて計算ノードに

適切な指示を出す役目を担っており，計算結果の回収も監視ノードが担当する．

計算ノード 計算資源を提供する計算機である．タスクを実行し，結果を監視ノードに通知する．また，他のタスクに影響を及ぼす情報に変化が生じた場合は，同期ポイントにて監視ノードに通知する．

監視ノードからのメッセージにより，計算の開始，終了，実行情報の更新，ロールバック，マイグレーションを行う．

4.2 バージョン管理

監視ノードはタスク全体のバージョンを管理し，全ての計算ノードはバージョン情報を共有する．各メッセージはバージョン情報と関連付けて管理されており，バージョンはロールバックを行うたびに更新される．

任意の計算ノードがロールバックを行った場合，この計算ノードがロールバック以前に送信した全てのメッセージは無効にしなければならない．新しい情報に基づいて行った計算結果は以前のように影響を及ぼさなくなる可能性があり，他の計算ノードが誤った情報に基づいて処理を行う可能性があるからである．これに対応し，受信したメッセージが適用すべき情報であるかを判断するためにバージョン情報を参照する．

ネットワーク上ではパケットの延着，順序の変更が起こりうるため，バージョン管理を行うことでこれらに起因するトラブルを防ぐ．バージョン情報に基づいて受信した情報の中から不要な情報を排除することで，全体の整合性を維持することができる．

4.3 実行情報の更新

タスクを処理するために必要な情報が更新された場合，監視ノードが該当する計算ノードに通知を行う．通知を受けた計算ノードはメッセージが作成されたバージョンを確認し，対象ステップのバージョンと一致したものだけを適用し，実行情報の更新を行う．バージョンが古い情報は破棄し，新しい情報は同じバージョンを計算するときまで保持する．

更新の手順はメッセージの送信元タスクの同期ポイントと通知を受けた計算ノードで実行中のタスクの最後に通過した同期ポイントとの関係により，次の3つの処理から1つを選択する．実行中のタスクが保持している最新の同期ポイントの番号を SP_{exec}，受信した実行情報の同期ポイントの番号を $SP_{receive}$ とする．

$SP_{exec} < SP_{receive}$ 実行中のタスクは実行情報を更新する必要のある同期ポイントに到達していない．受信した実行情報は該当する同期ポイントを通過する際に必要となるため，情報を保持して将来に備える．

$SP_{exec} = SP_{receive}$ 更新された実行情報は現在実行中の計算に影響を及ぼすため，即座に更新する．同期ポイント通過後にいくつかの計算を行っている場合は，$SP_{exec} > SP_{receive}$ と同じ処理を適用する．

$SP_{exec} > SP_{receive}$ 実行情報が更新された同期ポイントは既に計算済みであるため，オンデマンドロールバックを行い，更新を適用する．計算ノードがロールバックを行った場合，ロールバックした同期ポイント以降の更新情報は全て無効となる．この情報に基づいて行ったタスクの実行結果は整合性が保てなくなるため，全ての計算ノードは該当する同期ポイントまでロールバックを行い，保持している更新情報を全て破棄する．

4.4 オンデマンドロールバック

タスクを過去の任意の時点から再実行するためにはロールバックを行う必要があるが，ロールバックすることにより，ロールバック先以後の全ての計算結果が破棄されるとともに，全ての計算ノードが該当する同期ポイントまでロールバックする必要があり，そのコストは極めて大きい．そこで，過去の同期ポイントに対する情報更新であっても，単純に該当ポイントにロールバックするのではなく，その情報が影響を及ぼす同期ポイントを特定し，必要に応じて最小限のロールバックを行う．

ロールバックが必要な同期ポイントの探索方法は次の通りである．更新された実行情報と本来のロールバック先となる同期ポイントの計算結果を比較する．該当する同期ポイントの計算結果が受信した実行情報に影響を受ける場合，該当ポイントにロールバックする．受信した実行情報に影響を受けない場合は，最後に通過した同期ポイントまで順に同様の処理を繰り返す．最終的に，実行情報の更新が過去の計算結果に全く影響を及ばさなかった場合，ロールバックは行われない．

4.5 マイグレーション

オンデマンドロールバックによりロールバックの頻度を抑えることが可能であるが，ロールバックそのもののコストを抑えることはできない．提案システムでは全ての計算ノードが独立にタスクを実行するため，タスク間の同期ポイントの差が大きい．そのため，最も遅いタスクが他のタスクに影響を及ぼす．

提案手法ではマイグレーションを用いることでタスク間の同期ポイントの差を減らす．同期ポイントの差が規定以上になった時，最も遅いタスクと最も早いタスクを計算ノード間で交換する．

5. 性能評価

5.1 対象アプリケーション

本提案手法の有効性を検証するために，プロトタイプシステムを構築し，実験を行った．並列同期タスクとしてマルチエージェントシミュレーション（MAS）を想定し，分散処理を行った．同期ポイントはMASのステップ終了時に設定し，各ステップの処理結果と

表 1 ノードの諸元

CPU	Pentium 4 3.0EGHz
Memory	2GByte
OS	Scientific Linux 4.1 (kernel 2.6.9-11.EL)
Network	1000Base-T (Gigabit Ethernet)
gcc	3.4.3
Condor	6.6.7

表 2 実験パラメータ

空間	100x100
分割数	4
重複領域幅	2
エージェント数	1000
エージェント初期体力	1 ～ 1000
エージェント視野	2
資源量	1 ～ 10
資源分布	Random
終了ステップ	1000

所要時間を比較した．

5.2 シミュレーションモデル

シミュレーションは Sugarscape モデル[6] に基づいて行った．格子状の空間にエージェントと資源が存在し，エージェントは資源を求めて空間を移動する．エージェントには体力が存在し，資源を回収することで回復し，行動することで減少する．体力が無くなったエージェントは消滅し，全てのエージェントが消滅するか規定ステップを終えるとシミュレーションは終了する．回収された資源は復活しない．

空間はブロック分割によりタスクとして抽出され，隣接タスクの持つ情報の中でシミュレーションに必要な資源情報を共有する重複領域[7] を持つ．ミドルウェアには Condor[8] を用いた．実験で用いた計算機単体の諸元を表 1 に示す．実験には監視ノードに 1 台，計算ノードに 4 台の計算機を用いた．実験のパラメータを表 2 に示す．実験は各 50 回行った．

シミュレーションの流れは次の通りである．

(1) エージェントは自分の視野内で最大の資源を探し，進行方向を資源の存在する方向へ設定する
(2) 視野内に資源が全く無かった場合は，上下左右へ進むかその場に止まる，の 5 つの行動からランダムに 1 つを選択する
(3) 選択された行動を実行する．移動した場合は，エージェントの体力が減少する
(4) 移動先に資源が存在する場合，回収する
(5) 体力が無くなったエージョントは消滅する
(6) 1 ～ 5 を 1 ステップとし，空間内の全エージェントが消滅するか規定ステップを終えるまで繰り返す

5.3 実　　装

5.3.1 監視ノード

監視ノードは計算ノードの接続要求を待ち，タスク実行に必要な台数の計算ノードを確保すると実行開始を指示する．計算ノードから送られる各ステップの処理結果を回収するとともに，計算ノードにまたがるエージェント移動，重複領域内の資源更新の情報を影響する計算ノードに転送する．全ての計算ノードから実行終了通知を受けると，全計算ノードに実行結果を要求し，全ての結果を回収して終了する．

5.3.2 計算ノード

計算ノードは監視ノードに接続し，計算開始の指示を受けるまで待機する．計算が開始されると各ステップの処理を行う前にメッセージの有無を確認し，4.2 節で述べた動作を行う．ステップが終了すると，担当領域外へ転出したエージェントが存在するか，重複領域内で更新された資源があるかを確認し，いずれかの情報が存在する場合，更新情報として監視ノードに通知する．また，そのステップの処理結果を監視ノードに通知するとともに，エージェント情報と資源情報を履歴として保持する．領域内にエージェントが存在しなくなるとタスク終了を通知し，待機する．待機中に実行情報更新を受けた場合も同様に対処し，必要であれば再度計算を開始し，その旨を監視ノードに通知する．タスクの実行が終了し結果回収要求を受けると，計算結果を返答して終了する．

5.3.3 履 歴 管 理

ロールバックを行うためには，計算に必要な全ての情報をステップごとに記録しておく必要がある．今回適用した Sugarscape モデルでは，エージェント自体の情報とエージェントが意思決定を行うために必要な資源情報がこれにあたる．このステップごとに記録される情報を各ステップの履歴とみなし，履歴の管理方法を説明する．

計算ノードは各ステップ終了時にこれらの情報のスナップショットに現在のバージョン情報を付与して履歴として保存する．再計算が必要になった場合，該当ステップの 1 つ前のステップの履歴に基づきロールバックを行う．ロールバックを行うことにより対象ステップ以降は全て再計算されるため，不要となった履歴は全て削除する．また，ロールバックしたステップ後の再計算結果はロールバック前のものとは異なるため，バージョン更新を監視ノードに要求し，返答を受けるまで待機する．このバージョン情報は全計算ノードで共有され，バージョンが更新されると該当ステップまでロールバックを行う．

5.3.4 実行情報更新

実行情報の更新手順を図 3 に示す．図 3 では簡略化のため計算ノード間を取り持つ監視ノードを省略している．実行情報の更新を受け取ると，計算ノードの実行情報を更新するとともに，1 つ前のステップの履歴を更新する必要がある．履歴を更新しなかった場合，

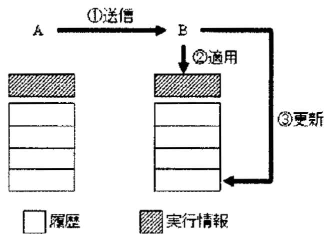

図 3 実行情報更新手順

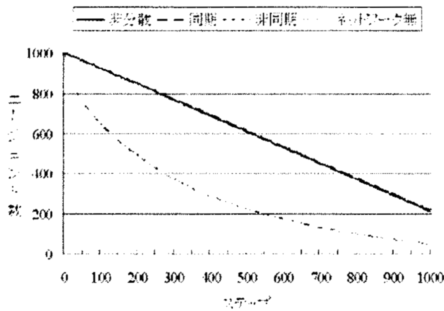

図 4 エージェント数の推移

同ステップに対してロールバックを行うと実行情報が変わってしまうためである．

5.4 整合性の評価

本稿で提案した手法による実験結果の整合性を評価するために，同じシミュレーションを分散化せずに1台の計算機で行った結果（非分散）と同期手法を用いた結果との比較を行った．同期手法とは，各同期ポイントで全ての計算ノードが情報交換を行うまで処理をブロックする手法である．図4にステップ終了時のエージェント数の推移のグラフを示す．グラフは縦軸が50回の試行におけるエージェント数の平均で横軸がステップである．グラフ中では分散化せずに1台で実行した非分散とステップ終了時に同期を取る同期，本提案手法である非同期が重なっており，エージェント数の推移はほぼ同じ軌跡を描いている．

さらに，通信することで正しい結果が得られていることを確認するために意図的にネットワークに障害を発生させ，計算ノード間の通信を妨害した実験を追加で行った．その結果，図4の一点鎖線に示すように，通信に失敗した場合はエージェントが急激に減少した．これにより，同期手法だけでなく提案手法である非同期手法においても，適切な通信を行うことで非分散化時と同等の結果が得られていることが示された．

5.5 計算時間の評価

同期手法と本提案手法のタスク終了までに要した時間の平均を図5に示す．同期手法に対して本研究で提案した非同期手法は25%程度の高速化が確認でき，本提案手法の有効性を示した．

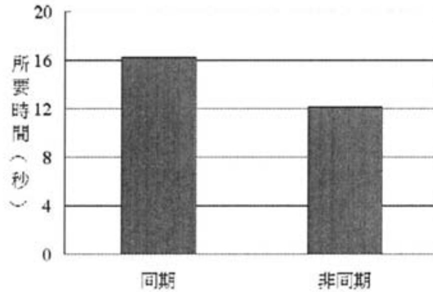

図 5 所要時間

6. 考 察

6.1 ホモジニアス環境での高速化

今回，実験はPCクラスタ上で行った．PCクラスタを構築する計算資源は一様の性能を持っており，一般利用者による割り込みは発生しないホモジニアスな環境となっている．ホモジニアス環境において本提案手法の有効性が示されたことには大きな意義があるが，本研究が本来対象とするヘテロジニアス環境で実験を行い，有効性を示す必要がある．

ホモジニアス環境でもタスク処理を高速化できたが，MASはエージェントの移動により各タスクの負荷が動的に変化し，同期待ちが生じやすい問題であり，同期待ちを解消し，有効利用することを目的とした本研究の対象と一致するためである．そのため，タスクの終了時間が不均一になり，動機待ちが生じるあらゆる問題に対して本研究は有効であると考えられる．

6.2 ロールバックコスト

本実験では，ロールバックが必要な同期ポイントを計算ノードが自ら判断し，最適なロールバックを行うオンデマンドロールバックが行われていない．1つのシミュレーションを完了するまでに平均して4800回程のロールバックが行われたが，同期手法と比較して高速なタスク処理が実現できた．

ロールバックを頻繁に行うタスクは処理速度の速いタスクであり，最も遅いタスクがロールバックを行うことは無い．また，ロールバックにより破棄される処理結果は従来同期待ちに浪費していた時間を利用して導かれたものである．よって，同期手法と比べればロールバックがタスク処理速度に与える影響は少ない．

本稿では，最大限のロールバックを行った状態でタスク処理速度を同期手法と比較するためオンデマンドロールバックを行わなかったが，これを行うことでロールバック回数を減らし，更なる高速化が可能である．

7. 議　　論

提案手法はタスクの相互影響を隠蔽し，独立タスクとして実行することを可能にする．これを実現するために取り入れた手法を応用することで，次の3つの方向性が見出せる．

7.1 フォルトトレラント

提案手法はロールバックのメカニズムを内包するため，実行情報のバックアップを行うだけで容易にフォルトトレラントを実現できる．全ての計算ノードは独立に動作するため，任意の計算ノードが故障しても全体が停止することはなく，任意の計算資源に実行情報を転送することでタスクを再開できる．また，計算の途中経過を見て任意の時点からやり直すことも可能である．

7.2 タスク同時多重実行による高安定高速処理

一般システムを利用したグリッド環境は本質的にヘテロジニアスなため，予測した通りにタスクが終わる保証が無く，割り込みの発生により瞬間的に性能が低下し，それが継続する可能性がある．この問題に対して，1つのタスクを2台以上の計算ノードで平行に計算し，任意の時点で同期を取り最も進んでいるタスクの実行情報を共有し，その時点から再スタートすることで複雑なヘテロジニアス環境においても一定の性能を保証すること可能である．

この時の計算ノードは多いほど良く，大規模グリッドやデスクトップグリッドに適した手法である．割り当てるタスクの動的な変更により計算ノードの自由な参加，離脱が可能になるため，数ある計算資源を常に有効に活用することができる．

8. 関 連 研 究

Condor[8] はPSAのような単一タスクの実行を保証するためにマイグレーション機能が存在するが，我々が注目している相互影響のあるタスク実行は考慮されていない．Condor のチェックポイント機能はタスクの再計算を支援するが，チェックポイントを取る間隔は一定であり，直近のチェックポイントにしか戻ることができない．我々の手法ではシミュレーションに応じて同期ポイントを柔軟に設定することが可能であり，任意のタイミングで同期ポイントに戻ることができる．

菅ら[7] はロールバック手法に関して compared-rollback を提案しているが，実行情報が本来のロールバック先以降に及ぼす影響は考慮されていない．また，タスク処理速度のばらつきにより未処理箇所の情報を得る可能性があるが，このことについては言及されていない．分散化による誤差を許容した上で高速化を行っており，我々のように計算結果の高い整合性を前提としたものではない．我々は計算資源やタスク負荷の不均一性を不可避なものとみなし，これを前提としたタスクの効率処理や計算資源の有効利用を研究している．

9. お わ り に

本稿では，非同期通信とタスクのオーバーラップ実行を用いてタスク間の依存関係を隠蔽し，相互影響のあるタスクの効率的な処理を可能にするグリッド環境を提案した．プロトタイプシステムによる実験結果から，タスク終了時間に不均一性を持つタスクを効率的に処理できることを確認し，本研究の更なる応用を議論した．

今後は，オンデマンドロールバックによる高速化とヘテロジニアス環境での実験，ツール化を行う．

謝辞　本研究の一部は，文部科学省社会連携研究推進事業（平成17年度～平成21年度）による私学助成を得て行われた．

参 考 文 献

1) David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky and Dan Werthimer: SETI@home - An Experiment in Public-Resource Computing., Communication of the ACM, Vol. 45, No. 11, pp. 56-61, 2002.
2) 竹房　あつ子, 松岡　聡: グリッド計算環境でのデッドラインを考慮したスケジューリング手法の性能, 電子情報通信学会論文誌, Vol. J86-D1, No. 9, pp. 661-670, 2003.
3) 首藤一幸, 大西丈治, 田中良夫, 関口智嗣: 計算機資源の流通および集約のための P2P ミドルウェア, 情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 6, pp. 208-221, 2004.
4) 谷村勇輔, 廣安知之, 三木光範: グリッド計算環境でのマスターワーカシステムの構築, 情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG 6, pp. 197-207, 2004.
5) 武田和大, 小野智司, 中山　茂: 異機種混合並列計算ミドルウェア JSGrid の開発と評価, 日本計算工学会, No. 20060005, 11pages, 2006.
6) Joshua M. Epstein, Robert Axtell (服部正太, 木村　香代子訳), 人工社会 - 複雑系とマルチエージェント・シミュレーション -, 共立出版, 1999.
7) 菅　真樹, 合田憲人: 並列災害救助シミュレーションにおける精度と計算時間に関する評価, 情報処理学会シンポジウム論文集, Vol. 2003, No. 8, pp. 325-332, 2003.
8) M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In Proceedings of the 8th International Conference of Distributed Computing Systems, pp.104-111, 1998.