# 協調バックグラウンドタスクスペースに関する検討

山口 実靖 * 相田 仁 ** 齊藤 忠夫 *

* 東京大学大学院工学系研究科

** 東京大学大学院新領域創成科学研究科

〒 113-8656 東京都文京区本郷 7-3-1

{sane,aida,saito}@sail.t.u-tokyo.ac.jp

## 概要

近年, 個人用計算機も含め非常に多くの計算機がネットワークに接続されるようになった. さらに, 個人用計算機はユーザが使用していない時間は負荷がほとんどなく非常に多くの計算機資源がアイドル状況にあることになる. 本稿ではこれらのネットワークにつながれたアイドル状況にある計算資源を協調させてバックグラウンドタスクとしてバッチ型の処理の実行をさせ活用する手法について述べる. 特に, (1) バックグラウンドで動かすタスクがローカルのタスクの作業を可能な限り妨げない, (2) 協調システムの一部に障害が発生してもシステム全体がダウンしない高い耐障害性がある, の 2 目標を重要と考えこれらに特化した手法を提案する. 提案手法により, ユーザがローカルの処理のために計算機を使う要求が発生したときにバックグラウンドのタスクを他の計算機に移送しローカルのタスクに影響を与えないことや, 協調システムに参加しているユーザが突然システムから切断したりするなの障害が発生しても正しく処理を続けることが可能なフォルトトレラントな環境が実現される. 提案システムはバックグラウンドのタスクに Watch Dog タイマーを設定する, 多重化するなどをして疎なつながりの計算資源群の信頼性を上げ, トークンパッシングを行い障害の検出をしている. また, 簡単な試作システムを実装し動作させ検証したところ, その有効性が確かめられた.

## 1 はじめに

計算機のネットワーク接続は進み多くの計算機が LAN やインターネットに接続されている. さらに, 以前とは異なり近年はサーバ用の計算機に限らず個人用計算機も常時あるいは長時間ネットワークに接続されている. これらの個人用計算機の多くはサービスを受けることのみを目的にネットワークに接続されており電子メールや WEB を使う以外はワードプロセッサや表計算などのスタンドアローンの計算機と同じ使われ方をしている. この場合これらの個人用計算機は常に高い負荷がかかっているわけではなく, 低い負荷状態にある時間も多いと思われる. 本稿ではこれらの状況を想定し "ローカルの作業に弊害を出さずに, アイドル状態にある計算資源を有効的に活用する" 手法を提案する. ローカルの作業とはワードプロセッサや表計算などを用いた対話的な作業 (以後このフォアグラウンドのタスクを "FG タスク" と呼ぶ) であり, 有効的な活用とはアイドル状態の資源を用いてバッチ的な処理 (以後このバックグラウンドのタスクを "BG タスク" と呼ぶ) を行うことを目指す. そしてこれらの計算機群で連携を行いアイドル状態の計算資源を提供しあうことにより全体でバッチ的な処理を行うことが可能となる. これはユーザにとっては自分の重要な "FG タスク" に弊害を出すこと無しに他人の計算機のアイドル状況の資源を用いて "BG タスク" を行うことが可能となることを意味している. また, 今後はユービキタスコンピューティングが実現されていき家電等の資源を利用することも可能となり, これらを有効的に活用するのにも役に立つ思われる.

本稿では第 2章において関連研究と本研究の新規性を示し, 第 3章において提案手法を示し, 第 4章において提案手法を評価し, 第 5章においてまとめと今後の課題を示す.

## 2 関連研究と本研究の新規性

本章にて, 本研究の想定している環境, 既存の研究との違い, 既存の研究として High Throughput Computing の紹介 (第 2.1節) を述べる.

本研究で下記の分散環境を想定している. (1) 使用する計算資源群はヘテロジーニアスである, (2) 計算機群そのものを管理しておらず, 計算機にはプロセスを立ち上げるのみである, (3) システムの環境が動的である, (4) 構成する資源群は信頼性が低い, ここで (3), (4) はユーザのシステムへの参加, 脱退が頻繁に起こるからである. 上記の理由から整った環境である並列計算機, PC/ワークステーションクラスタ, 分散 OS とは想定している環境が大きく異なる. また, 連携処理専用のサーバを想定していないことや超大規模ネットワーク環境を想定しておらず Grid[1] , Ninf[2] などのグローバルコンピューティングとも想定している環境が異なる. 本研究では計算機数台から十数台の規模を想定している. さらに, Ninf などは RPC[1] モデルをしており, 主目的は処理内容 (実装) と資源の共有にあり目的も大きく異なる. 下記の Condor[3, 4, 5, 6] は本稿と同じ目標であり安定した環境では有用であると考えるが, 主に Unix などの常時接続された安定した環境を想定しており個人用の計算機及びそれによる不安定さなどについては深く考慮されていない.

本稿では特に以下の 2 目標を重要と考え, これらに特化したシステムを提案する. また, これらが本研究の新規性である.

[1] Remtoe Procedure Call

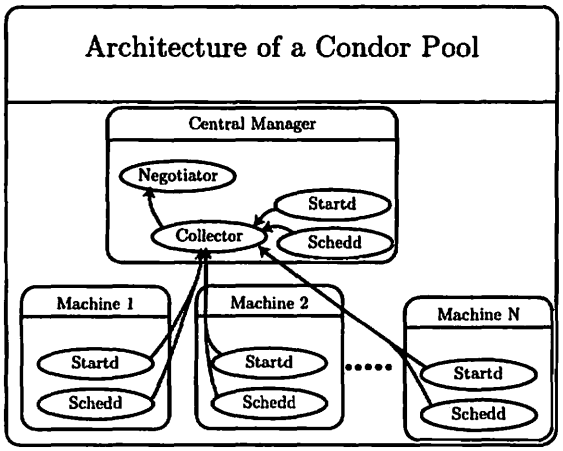


図 1: Condor Architecture

(1) "BG タスク"が "FG タスク"の作業を可能な限り妨げない

(2) 協調システムの一部に障害が発生してもシステム全体がダウンしない高い耐障害性がある

まず, (1) は "アイドル状況の資源も有効利用" することを目指し本来の作業に支障を来しては無意味と考え本稿の重要課題とする. 次に (2) は協調システムの参加者は "FG タスク"のために "BG タスク"を容易に停止すること, 参加計算機はサーバ専用計算機としての参加ではなくアイドル状態の資源の供給のために参加しているに過ぎないことなどから低品質の計算資源になることは避けられないと考え重要課題とした.

さらに, (3) 協調システムにユーザ認証機構があり認証ユーザレベルでの資源割り当てが可能である, (4) 動的な環境における適切な資源割当が可能である, も実現が推奨されると考える.

## 2.1 High Throughput Computing

**Condor** Wisconsin 大学のコンピュータ科学学部によって開発された Condor は不使用状態にあり有意義に使用されていない CPU 資源を有効利用するためのシステムであり, 基本的にワークステーションクラスタ上で動作する. Condor で述べる High Throughput Computing とは, 長期的期間で見た場合での高い計算能力のことである [6].

コンドルシステムはコンドルプールで管理され, 図 1の構造をしている. クラスタ上の個人用計算機の状況を監視するために "master", "schedd", "startd" の 3 種類のコンドルデーモンが常に稼働している. "master" は他のコンドルデーモンが稼働しているかを確認し, 障害が発見されたらデーモンを再起動する. "schedd" は要求されたジョブを管理し, "startd" はジョブを投入可能なアイドル状況の計算機の発見やユーザが戻り計算機がアイドル状況でなくなったことの検知を行う.

システムには Central Manager(以下 CM) 計算機が存在し, 全資源と全ジョブを管理している. システム内の全 "startd" および全 "schedd" は CM 上の "Collector" というデーモンにプール内の情報 ("startd" は計算機のアイドル状況, "schedd" は投入されたジョブの状況) を伝える. そして "Nagotiator" が定期的に "Collector" から情報を集めアイドル計算機を発見し, その計算機とマッチするジョブを見つける.

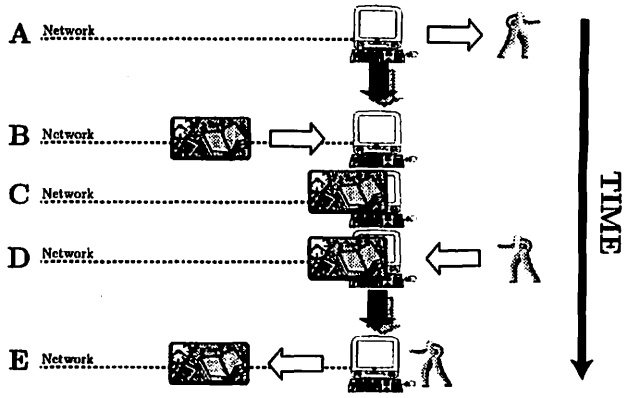


図 2: 提案システムの概要

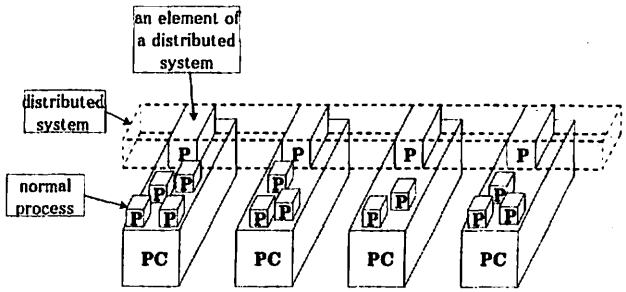


図 3: 提案システムの構造

ジョブが投入されると以下の手順により実行される. (1) ジョブ J が計算機に投入される, (2)CM がアイドル計算機を発見し, その計算機にマッチするジョブとしてジョブ J を採用する, (3) その計算機でジョブ J が実行される, プール内の計算機はアーキテクチャ, OS, メモリ量, CPU 能力などの情報を持ち, 投入されたジョブは実行アーキテクチャ, 必要メモリなどの情報を持つ. これらの情報をもとに計算機とジョブのマッチが決定される.

計算機ユーザが作業を再開し Condor に対する計算資源の解放を停止したときは, これらのプロセスは他の実行計算機に移動させられる [7, 8].

## 3 提案手法

提案するシステムの概要は図 2に示される. すなわち, ユーザが席を外し計算機がアイドル状態になる (図 2の A) と, ネットワークによりそのアイドル計算機にタスクが投入される (図 2の B). ユーザが席に戻るまでの間は他から投入されたタスクを処理し続ける (図 2の C). ユーザが席に戻ると (図 2の D) これらのタスクはユーザの作業の妨げとなるので他の計算機に移送する (図 2の E).

提案手法は図 3の構造をしている. すなわち, 各計算機には "FG タスク"(図 3の "normal process") と, 提案する協調システムのための "BG タスク"の実行環境 (図 3の "an element of a distributed system", 以後 "BG タスクスペース"と呼ぶ) が存在する. 前述のように各計算機の "BG タスクスペース"同士は協調動作し, 全体として一つのプラットフォームを実現する. ローカルの

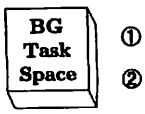


図 4: 最初の “BG タスクスペース”のオープン

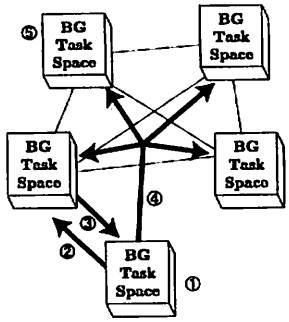


図 5: 最初でない “BG タスクスペース”のオープン

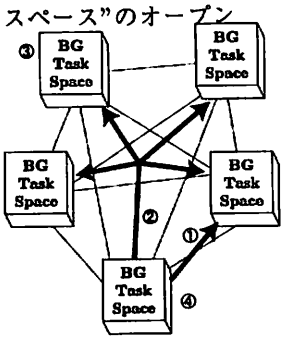


図 6: 最後でない “BG タスクスペース”のクローズ


図 7: 最後の “BG タスクスペース”のクローズ

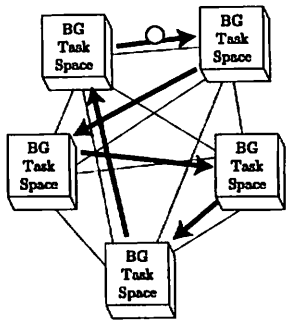


図 8: “BG タスクスペース”の調査

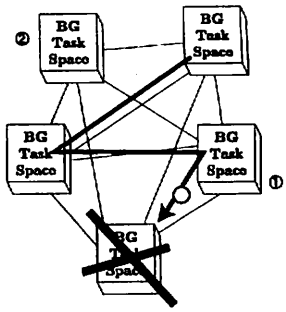


図 9: 障害検出

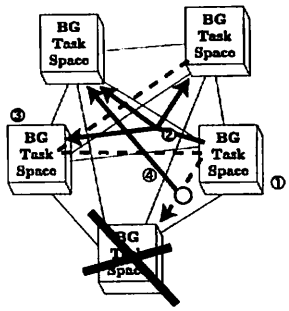


図 10: 障害スペースの隣接スペースが検出

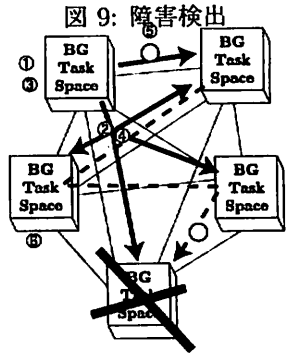


図 11: トークンのタイムアウトによる検出

OS は “FG タスク”と “BG タスクスペース”を同等に扱うため, “BG タスク”のために CPU やネットワークなどの資源を予約することはできず, 必然的に “BG タスクスペース”の環境は変化が多く不安定となる.

本手法の実行手順を以下の第 3.1節, 第 3.2節で述べる.

## 3.1 BG タスクスペース

“BG タスクスペース”は “BG タスク”の実行環境である. クライアントは “BG タスクスペース”に対してタスクを投入することが可能であり, 必要ならば処理結果を受けとることが可能である.

“BG タスクスペース”のオープンの手順を述べる. 協調システムで最初の “BG タスクスペース”をオープンするときは図 4の様に①“BG タスクスペース”を作成する, ②“BG タスクスペース”リストを作成し自分の登録する, となる. 新しい “BG タスクスペース”をオープンし既存の協調システムに参加する場合は図 5の様に①“BG タスクスペース”を作成する, ②システム内のいずれかの “BG タスクスペース”に現在の “BG タスクスペース”のリストを要求する, ③リストを獲得する, ④新規加入を全スペースにアナウンスする, ⑤各スペースは保持しているリストを更新する, となる.

次に, “BG タスクスペース”のクローズの手順を述べる. 最後でない “BG タスクスペース”をクローズするときは図 6の様に①処理中の “BG タスク”を他の “BG タスクスペース”に転送する, ②スペースを閉じることを他の全スペースにアナウンスする, ③各スペースは保持しているリストを更新する, ④スペースを閉じる, となる. 最後の “BG タスクスペース”をクローズするときは図 7の様に①処理中の “BG タスク”を不揮発メモリにストアするか破棄する, ②スペースを閉じる, となる.

また, “BG タスクスペース”に起きた障害を検出するために図 8の様に常にトークンパッシング方式でトークンを交換する. ある “BG タスクスペース”に障害が発生したときは図 9の様に①隣接 “BG タスクスペース”との通信に失敗した, あるいは②トークンがタイムアウト時間までに送られてこない, ことにより検出が可能となる. 前者 (隣接スペース) の場合は以下の手続きにより復旧する (図 10参照). ①障害を検出, ②その障害を全スペースにアナウンス, ③各スペースはリストを更新する, ④トークンを次の次のスペースに転送する, 後者 (トークンのタイムアウト) の場合は以下の手続きにより復旧する (図 11参照). ①タイムアウトにより障害を検出する, ②全スペースと交信を試みる, ③現在の障害状況が確認される, ④最新のスペースリストを全スペースにアナウンスする, ⑤新しい ID のトークンを作成しそれを隣接ノードに送る, 新しいトークンを作成する事によりトークンが複数存在してしまう可能性があるので最新トークン以外は破棄する必要がある. スペース間で同じトークン ID を割り振らないためにはトークン ID に BG タスクスペース ID を含ませればよい. また, 等しい新しさのトークンは BG タスクスペース ID で順序付けが可能となる.

ネーミングサーバを用いて “BG タスクスペース”を管理しない理由はネーミングサーバは停止することが許されず常に稼働かつ接続されている計算機が必要であり, 自由に参加 / 脱退が可能であるという利点が失われるからである.

## 3.2 “BG タスク”の処理

本節では “BG タスク”の処理について述べる.

システムに対する “BG タスク”の要求は以下の様に

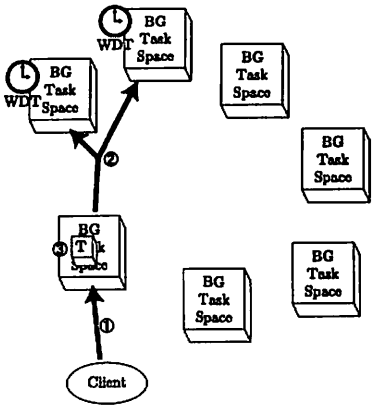


図 12: “BG タスク”の投入

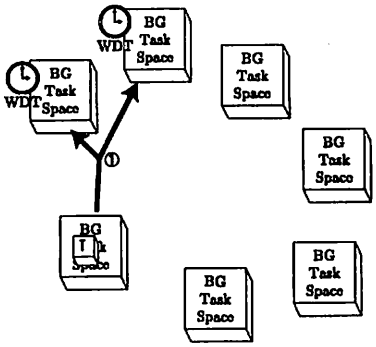


図 13: “BG タスク”の処理

行う (図 12参照). ①クライアントが “BG タスク”要求をいずれかの “BG タスクスペース”に送信する, ②その “BG タスク”のための Watch Dog タイマーを 1 つ以上起動する, ③その “BG タスク”を 1ヶ所以上の “BG タスクスペース”で開始する, Watch Dog タイマーはその “BG タスク”の処理に発生した障害を検出するためのものであり, 要求された “BG タスク”のコピーを保持している. “BG タスク”を起動するよりも先に Watch Dog タイマーを起動することにより, ②までの処理が成功すれば全 “BG タスク”の処理と全 Watch Dog タイマーが停止しない限り処理を正常に終わらせることが可能である. “BG タスク”処理中は定期的に Watch Dog タイマーをリセットする (図 13参照).

タスクの移送は以下の様に行う (図 14参照). ①“BG タスク”の移送要求が発生する, ②新しい Watch Dog タイマーを起動する, ③古い Watch Dog タイマーを停止

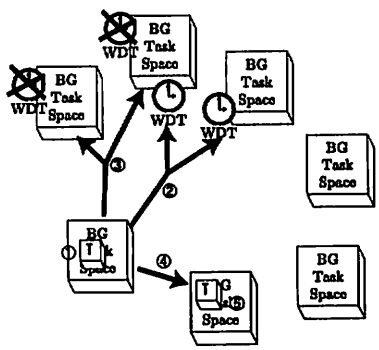


図 14: “BG タスク”の移送

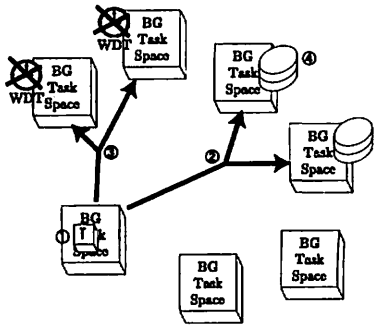


図 15: “BG タスク”の終了

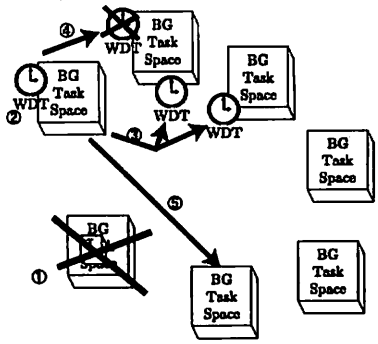


図 16: Watch Dog タイマーのタイムアウト

する, ④別の “BG タスクスペース”に “BG タスク”を移動する, 同様にして②までの処理が成功すれば全 “BG タスク”の処理と全 Watch Dog タイマーが停止しない限り処理を正常に終わらせることが可能である.

タスクの終了処理は以下の様に行う (図 15参照). ①“BG タスク”が終了する, ②“BG タスク”の処理結果を結果格納ストレージに格納するなどのその “BG タスク”に定められた終了処理を行う, ③Watch Dog タイマーを停止する, ④結果格納ストレージに納められている結果は有限時間で削除される, Watch Dog タイマーを停止する前にストレージへの格納を試みているため, (全 Watch Dog タイマーに障害が発生しない限り) 冗長に複数回実行され複数回ストレージに伝えられることがあっても, 処理の格納がされないまま処理が停止してしまうことはない.

障害が発生したときは Watch Dog タイマーにより障害が検出される (図 16参照). ①障害が発生しその “BG タスク”を処理している “BG タスクスペース”がなくなる, ②Watch Dog タイマーがタイムアウトし障害を検出する, ③Watch Dog タイマーが再実行のための Watch Dog タイマーを起動する, ④古い Watch Dog タイマーを停止する, ⑤別の “BG タスクスペース”に再度 “BG タスク”を要求する, 前述のように Watch Dog タイマーは “BG タスク”の要求のコピーを保持している. 古い Watch Dog タイマーを停止する前に新しい Watch Dog タイマーを起動しているため, 処理が重複実行されてしまう可能性があるが, 処理が障害により停止してしまう危険性は低くなる. Watch Dog タイマーのタイムアウト時間を異なる値にしておくことにより重複実行の可能性を軽減できる.

## 3.3 “FG タスク”への影響

第1章で述べたように, “BG タスク”が “FG タスク”の作業を妨げてはならない. 夜間やユーザの外出中には “BG タスク”に資源を割り当てるが, ユーザが “FG タスク”の作業を再開するときには速やかに “BG タスク”は資源を解放する必要がある. 資源の解放としては以下の4種類の “BG タスク”の処理方法が考えられる.

1. システム内の他のアイドル計算機に移送
2. サスペンド状態にする
3. ローカル OS での優先度を下げる
4. 破棄する

方法1は “BG タスク”にとっては理想的な方法であるが移送の処理があり速やかに資源を解放することができず, 厳密には “FG タスク”にとって理想的な方法ではない. 方法2, 方法3は速やかな移行が可能であることが予想されるが, 第4.1節で後述するようにデメリットも多い. また, CPU 資源は解放してもメモリなどの資源は解放していないことになる. “FG タスク”の処理を妨げることが全く許されない場合は方法4も必要であると考える. この場合, 破棄を行ってもその “BG タスク”の処理が正しく行われること, 破棄の被害を最小限に抑えることが必要となる (第3.4節参照).

## 3.4 耐障害性

前述のように個々の “BG タスクスペース”は信頼性が低い. これらの環境で高い信頼性を実現するために以下のことが考慮されている.

### 3.4.1 Watch Dog タイマー

Watch Dog タイマーは正しく終了されるか分からない処理を行う前に起動するタイマーであり, タイムアウト前に処理の終了に成功した場合はタイマーは止められるが, 逆にタイマーがタイムアウト時間までに止められなかった場合は処理の終了に失敗したことが検出される. 図6, 図7の様な処理が行われずに “BG タスクスペース”が終了したときはその “BG タスクスペース”上で行われていた “BG タスク”が処理要求ごと失われてしまう[2]. これを避けるために Watch Dog タイマーを導入した. さらに, 提案方式では Watch Dog タイマーに “BG タスク”の処理内容のコピーが登録されている. これにより, 障害の検出をすると同時に再実行が可能である. また, 障害を検出したが再実行処理を依頼する “BG タスクスペース”がダウンしており再実行が行えないことも避けることができる.

提案方式では必ず処理を始める前に Watch Dog タイマーを起動している (第3.2節参照). この理由とその効果を以下に記す. (a)Watch Dog タイマー, “BG タスク”の順に開始する方法において Watch Dog タイマーのみ起動し障害が発生した場合と, (b)“BG タスク”, Watch Dog タイマーの順に開始する方法において “BG タスク”のみ起動し障害が発生した場合を比較する. 前者 (a) は Watch Dog タイマーがタイムアウトする前に Watch Dog タイマーの “BG タスクスペース”が停止しなければよい. 後者 (b) は “BG タスク”が次のチェックポイント時刻までに “BG タスク”を実行している “BG タスクスペース”が停止しなければよい. チェックポイント間隔を短くとることと比較してタイムアウト間隔を短くとることの方が容易である[3]ので Watch Dog タイマーを先に起動する方法がより信頼性が高いと言える. さらに, “BG タスク”の処理より Watch Dog タイマーの処理の方が必要とする資源が少なく, Watch Dog タイマーの方が多重度を高くすることが可能である. よって, 提案手法では Watch Dog タイマーを起動することまで正しく行われればその “BG タスク”は失われないと言える.

### 3.4.2 “BG タスク”の移送

ある “BG タスクスペース”でクローズ要求が発生したときに “BG タスク”を他の “BG タスクスペース”に移動することが可能であればそれまでの処理内容が失われない.

### 3.4.3 チェックポイント

“BG タスク”の処理が進んだ場合は障害発生時の再実行のコストを削減するためにチェックポイントをとっておくことが有効である. 提案方式では Watch Dog タイマーの再実行 “BG タスク”を更新することによりチェックポイントをとることが可能であり, この処理はプロセスの移送で行われる処理と同じである. よって, 他の “BG タスクスペース”あるいは同一 “BG タスクスペース”に “BG タスク”を移送することによりチェックポイントをとることが可能である.

## 3.5 結果の格納

“BG タスク”終了時の処理 (図15参照) としてはクライアントプロセスと交信し結果を伝えることが理想的であるがこれは “BG タスク”終了時にクライアントと交信できる状態にない場合は実現できない. そこで, 結果の通知の方法としてストレージに格納しておきクライアントがのちに読み込む方法を提案した.

また, FT[4] のための多重化, 前述の障害による Watch Dog タイマーの冗長動作によりストレージサーバは複数個の結果を受け取ることがあるが最初に得られた結果を採用すればよい.

# 4 評価

## 4.1 分散協調のトレードオフに関する考察

1台の計算機内でワードプロセッサや Web ブラウザを “FG タスク”としシミュレーションや数値計算を “BG タスク”とすると, “FG タスク”のワードプロセッサの作業に影響を与えずにアイドル時間の資源のみを用いて “BG タスク”のシミュレーションを進めることが可能となる. この1台の計算機内のプロセスを “FG タスク”と “BG タスク”にわけプロセスに優先度をつける[5]ことと提案する協調システムの違いは表1のようになる.

[2] 再度処理を行わなくてはならない “処理結果が失われる” と異なり, “要求が失われる” は要求が発生したこと自体が失われ再度処理することすらできない

[3] 一般にチェックポイントをとる処理の方が実現が困難であり, 処理量が多い.

[4] Fault Tolerant

[5] UNIX 系 OS での nice コマンドなど

| | 協調 | 非協調 |
|---|---|---|
| 並列実行 | 可能 | 不可能 |
| FT 化 | 可能 | 不可能 |
| オーバーヘッド | 大 | 小 |
| 規模可変性 | 高 | 低い |
| 資源利用効率 | 高い | bgタスクの有無に依存 |
| スループット | 安定 | fg タスクに依存 |

表 1: 協調型システムと非協調型システム

まず, 協調システムでは並列実行可能であるタスクのターンアラウンド時間の短縮が可能である. また, 単一計算機のみでの実行の場合はシステムの信頼性がその唯一の計算機に依存し FT 化は不可能であるが, 協調システムでは可能である. 次に, タスクの移送などの処理が増えオーバーヘッドは協調システムの方が大きいと考える. また, 単一計算機で行っている場合は常に "BG タスク"が存在するとは限らず, 資源の有効利用は困難であるが協調システムはより高い確率で "BG タスク"が存在する. つまり, ある時刻において "BG タスク"の要望がないユーザと "BG タスク"の要望が多数あるユーザを集めることにより資源の効率的な使用が期待できる. これらを集めることによる要求数の時間的, 計算機的偏りの軽減も期待できる.

## 4.2 試作

第 1 目標である「"BG タスク"が "FG タスク"の処理を妨げない」を目標とするシステムの Java 言語による試作を行った. クライアントが "BG タスクスペース"に "BG タスク"を投入すると "BG タスクスペース"はその "BG タスク"にスレッドを割り当て実行を開始させる. "BG タスクスペース"の開始時は "BG タスク"を受け付けそれを処理するが, ユーザからの "BG タスクスペース"のクローズ要求があると処理中の "BG タスク"を全て他の "BG タスクスペース"に移送しクローズすることを可能とした. しかし, 現試作の "BG タスク"スレッドの移送の実装は Java 言語の直列化機能によりインスタンスを直列化し転送するにとどまっておりスタック上のデータやプログラムカウンタを引き継いで移送することはできない. よって, 任意の時刻に "BG タスク"を別の "BG タスクスペース"に移送し次の処理から再開することはまだ実現されておらず, 移送可能状態になったときはじめて移送される[6].

現状の試作はアイドル状態にある計算資源を利用してバッチ的な処理を行う目標はある程度達成している. 第 1 の目標である「"BG タスク"が "FG タスク"の処理を妨げない」はユーザからの要求により "BG タスク"を他の "BG タスクスペース"に移送できるため低いレベルでは達成されていると言え, 現試作でも十分に有効性は確認された. しかし, 任意の時刻に移送を行うことが実現されておらず理想的なレベルでの達成はなされておらずさらなる改良が望まれる.

[6] Mercury[9] など同程度の移送のみ実現しているモバイルエージェント実装も多く存在する.

## 5 おわりに

本稿では, ネットワークに接続された計算機群のアイドル状態の計算資源を利用してバッチ的な処理を行う手法として, 特に (1) バックグラウンドのタスクがフォアグラウンドのタスクを妨げないこと, (2) 高い耐障害性があること, に重点を置いた手法を提案した. 提案手法はある計算機のフォアグラウンドのタスクが開始されたときにはバックグラウンドのタスクは他のアイドル計算機に移送するか破棄するなどをしてフォアグラウンドのタスクへの影響を最小限にしている. また, 提案手法は Watch Dog タイマー, トークンパッシングを用いることにより信頼性の低い計算機群を用いても投入したタスクが正しく処理されることを実現している. そして最後に簡単な試作を行いその評価をし有効性を確認した.

今後は提案システムの実装, 評価を進めて行くとともに (1) ユーザ認証およびユーザ指向資源割り当て, (2) 動的な負荷分散機構, (3) Disconnected Operattion(オフライン作業および再接続時の同期), (4) 性能評価, について考察をして行く.

## 参考文献


[1] Ian Foster, Carl Kesselman, "The GRID Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1999.

[2] 中田秀基, 佐藤三久, 関口智嗣, "ネットワーク数値情報ライブラリ Ninf のための RPC システムの概要", 電子技術総合研究所 Tecnical Report TR-95-28.

[3] Jim Basney, Miron Livny, and Todd Tannenbaum, "High Throughput Computing with Condor", HPCU news, Volume 1(2), June 1997.

[4] Scott Fields, "Hunting for Wasted Computing Power", 1993 Research Sampler, University of Wisconsin-Madison

[5] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum, "Mechanisms for High Throughput Computing", SPEEDUP Journal, Vol. 11, No. 1, June 1997

[6] "Overview of the Condor High Throughput Computing System"

[7] Jim Pruyne and Miron Livny, "Managing Checkpoints for Parallel Programs" Workshop on Job Scheduling Strategies for Parallel Processing IPPS '96

[8] "Checkpointing and Migration of UNIX Processes in the Condor Distributed Processing System", Dr Dobbs Journal, Feb 1995

[9] 岩井 俊弥, "Java モバイル・エージェント", ソフト・リサーチ・センター, 1998