基幹バッチ処理向けグリッドスケジューラの耐障害性向上手法

細内昌明[†] 渡辺和彦[††] 石合秀喜[†††]

複数のジョブに渡って大量データを処理するワークフローがある基幹バッチ処理は，障害による遅延を最小化し終了時刻を厳守することが求められる．基幹バッチ処理にグリッドスケジューラを適用することでハードウェア障害への耐障害性は向上するが，他ノードで再実行しても回復できない自動回復不可能障害への耐障害性は向上せず，分割数が増えることで障害回復運用が複雑化し遅延を招きやすい．複雑化を回避するため障害発生時にワークフローを中断すると，再実行負荷を低減することができない．

このため，自動回復不可能障害の耐障害向上手法として，データ視点のワークフロー管理手法を提案する．大量データを分割したロットを単位にワークフロー内の進捗状態を管理する．障害が発生してもワークフローは継続し，個々のロットを指定しなくても，未実行または異常終了したタスクのロットのみを自動選択して再実行タスクに割り当てる．本手法を実装したグリッドスケジューラ uGPS では，分割数が増えても再実行のための運用時間も再実行時間も増加させず，障害による遅延を抑制し，耐障害性を高めることができることを示した．

Fault tolerance improve method of grid scheduler for mission critical batch job

Masaaki Hosouchi,[†] Kazuhiko Watanabe[††] and Hideki Ishiai[†††]

Minimization delay due to the failure and the punctuality of the job workflow ending time are required in the mission critical batch processing large scale data across multi jobs. The fault tolerance of the hardware failure is improved by applying a grid scheduler to the mission critical batch processing. But, the fault tolerance of the automatic recoverable failure cannot be improved even if rerun the workflow to other nodes, and the workflow is delayed easily because fault recovery use becomes complicated due to increasing number of partitions. If the workflow is stopped in case of failure to avoid complexity, rerun workflow 's load cannot be reduced.

Therefore, this paper propose a data-centric workflow management method for improve fault-tolerance of the automatic recoverable. In this method, the scheduler manages the progress state of the workflow by the lot of the large data. The workflow continues even if a failure occurs. The scheduler assigns only the lot of not yet executing or abnormal ended task to a resubmit job 's task without specifying individual lot. This paper described that the fault tolerance can be improved and control delay due to fault without increasing both the operative time for resubmits and the re-execute time even if number of partitions increased in the grid scheduler " uGPS " which implemented this method.

1. はじめに

近年，小口取引の拡大や，電子マネーなどの少額決済の普及，アルゴリズム取引など取引の機械化，などの影響によりトランザクション量は年々増加している．このため，トランザクションデータを入力とする，金融機関の口座振替処理・決済処理・公共料金計算処理・企業取引の明細書出力処理・売上受注データのチェック/加工/集計処理，といった数百万から数億規模の大量データレコードを扱うミッションクリティカルな基幹業務のバッチ処理[1](基幹バッチ処理) の負荷も増加傾向にある．一方，基幹バッチ処理とリソースやデータを共有するオンライン処理のサービス時間延長に

† (株) 日立製作所 横浜研究所
Hitachi Ltd. Yokohama Research Laboratory
†† (株) 日立製作所 ソフトウェア事業部
Hitachi Ltd. Software Division
††† (株) 日立製作所 金融システム事業部
Hitachi Ltd. Financial Information systems Division
☆ LSF は Platform Computing Corporation の米国およびその他の国・地域における登録商標です.

より，基幹バッチ処理に許容される時間は縮小傾向にある．

その結果，バッチ処理の時間的余裕が少なくなり，障害や予期せぬ大量のデータが発生すると，遅れを取り戻せずにバッチ処理サービス時間内にバッチ処理が終了せずオンラインシステム開始が遅延する突き抜けと呼ばれる現象が発生するリスクが増加している．このため，データ量が増加してもバッチ処理を高速・確実に実行可能な，スケーラブルかつ高信頼なバッチジョブスケジューラが求められている．

スケーラブルなジョブスケジューラとして，多数の計算機から構成される分散並列コンピューティングシステムを対象とするグリッドスケジューラがあり，代表的なものに Globus toolkit[2] や Condor[3] がある．グリッドスケジューラは，科学技術計算だけでなく，金融リスク計算や証券トレーデングシステムなどのエンタープライズ領域にも浸透しつつある[4]．

しかし，基幹バッチ処理は，大量データレコードを扱うデータ並列処理に適したジョブはあるが，グリッドスケジューラ適用はまだ浸透していない．Grid Engine[5] ,LSF[6] などのグリッドスケジューラや，分散並列処理のフレームワークである MapReduce[7] や Hadoop[8] では，ノード障害に対する耐障害性は他ノードへの代替実行技術により確保されているが，代替実行だけでは基幹バッチ処理の必須要件である所定時間内終了厳守を保証するには充分でない．ジョブを構成する業務アプリケーションプログラム (以下 AP と略す）の不良やデータ不良など，代替実行が適用できないケースまではカバーできておらず，分散処理やデータ分割による複雑さの増加が障害時の運用を妨げる懸念がある．また，基幹バッチ処理では，複数ジョブにまたがったジョブ横断型の大量データ処理が一般的に利用されているが，ジョブ横断型の大量データ処理は考慮されていない．

このため，グリッド適用によりジョブ数やノード数が増加しても障害復旧運用を煩雑化させずに，障害が発生しても確実に所定時間内にバッチ処理を終了可能な耐障害性の高いグリッドスケジューラが求められている．

本論文の流れは，以下のとおりである．まず，2 章では，従来の基幹バッチ処理の概要・課題と，グリッド化の効果・課題を述べる．3 章では，基幹バッチ処理向けグリッドスケジューラ uGPS にて実装した自動回復不可能障害運用支援機能を提案する．4 章では評価結果を述べる．最後に関連研究との比較とまとめを示す．

2. 背　　景

2.1 基幹バッチ処理従来手法の概要

基幹バッチ処理は，科学技術計算・金融リスク計算・テキスト分析処理など他のバッチ処理に比較して，以下の特徴・要件を有する．

(1) **大量定型レコード処理**

数 100〜数 1000 バイトの固定または可変長のレコードの定型構造化データを処理する．レコード数は，最大で数億規模に達する．

(2) **終了時刻厳守**

オンラインシステムとのリソース共有により，規定時間内にバッチ処理を終了させる要件が厳しい．例えば，有限のハードウェアリソース制約のもとで昼間のオンライン処理応答性能を維持するために，不急の振込処理をオンラインシステム停止中の夜間に実行するバッチ処理が終了しないと，翌日のオンラインシステムを立ち上げることができない．終了時刻を厳守するためには，障害から迅速に復旧して障害による遅延を抑制することが必要であり，自動回復できない障害に対する障害運用支援機能が重要である．

(3) **ジョブ横断型大量データ処理**

基幹バッチ処理中間ファイルを介して受け渡しながら，大量レコードをワークフロー内の複数のジョブに渡って処理することが多い．例えば，図 1 のワークフロー例では，データベースから抽出またはオンラインシステムから受信した入力データ内の取引レコードを，商品コードや顧客 ID 順にソートする処理や重複をチェックする処理を実行し，商品コードや顧客 ID 順にソートされたマスタデータレコードと照合して入力データの正当性をチェックする処理を実行し，入力データを顧客データとマージして帳票出力するジョブと，集計するジョブとを並列に実行する．なお，ワークフロー上で先に実行するジョブを先行ジョブ，後に実行されるジョブを後続ジョブと定義する．

(4) **データ完全性**

基幹バッチ処理ではどのデータも重要である．データロストは許されない．

(5) **移行容易性**

基幹処理は，企業にとって必要不可欠な処理で，ハードウェア寿命以上の長期に渡って運用される．このため，新規構築よりも改変や移行が多い．また，高付加価値を生み出す処理ではないため，終了時刻厳守要件を損なわない範囲で，ハードウェア・運用・移行などのコスト低減が要件として求められる．

メインフレームなどの従来の基幹バッチスケジュー

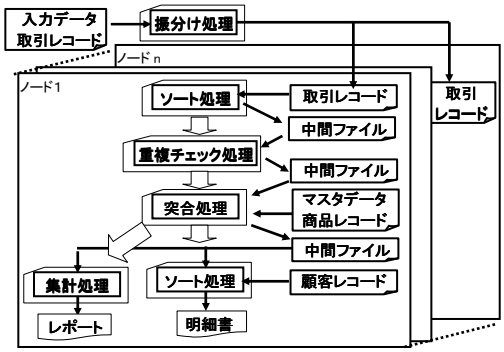

図 1 基幹バッチジョブワークフローの例
Fig. 1 Example of mission critical batch job workflow.

ラでは，終了時刻厳守要件に対して，以下のように高速化・高信頼化対応をしてきた．

(1) **高速化**

支店で業務が行われる基幹業務は，業務の入出力データも支店ごとの独立性が高く，支店単位に処理しても支障がない．このため，大量データの基幹バッチ処理では，従来から，支店コードなどレコードに含まれる特定項目の値の範囲ごとに，入力データファイル内のレコードを少数・固定数のファイルに振り分けて分割し，分割ファイル数と同数のジョブを事前定義して各分割ファイルを多重並列処理する高速化手法が広く用いられてきた[9]．分割数や値の範囲は変動しない．

(2) **高信頼化**

メインフレームなど高度に多重化された高信頼サーバノードでバッチ処理を実行することで，ハードウェア要因による障害発生率を抑えてきた．プログラム不良やデータ不良などの要因による異常終了に対しては，APによるデータ不良判断処理，スケジューラによるジョブ状態依存のワークフロー制御，OSのトラブルシュート機能（障害要因解析に必要なダンプやログなどの情報収集機能）などを強化することで，障害後の復旧時間を改善してきた．

2.2 従来手法の課題と基幹バッチ処理グリッド化の狙い

基幹バッチ処理にグリッドスケジューラと超多重度分割手法を適用することで，従来手法（少数固定分割手法）の課題を解決する．多重度とは，処理の同時並列実行数である．超多重度分割手法では，分割数を固定せず，多重度を超える多数のファイルに入力データファイルを分割し，同一ノードで複数のロットを並列処理する．入力データを分割したファイルを，以下ロットと呼ぶ．グリッドスケジューラは，各ロットを処理するタスクを，実行中のタスク数が多重度未満であるノードに投入する．タスクとは，分割前のジョブと実行するAPは同じであるが，入力データがロットの1つであるジョブである．従来手法では，分割数が少数に固定されていたため，データ並列性が低く，並列化による高速化に限界があったが，任意の数に分割可能にすることで，ノードを増やすことで，CPUや入出力スループット向上が見込める．

また，従来手法では，各ノードでタスクを1つしか実行しないため，特定分割キーのデータが増大した場合に特定ノードに負荷が集中して全体の処理時間が遅延する危険性があったが，ノード数を超えた多数のロットに入力データを分割し，同一ノードで複数のロットを処理することで，各ロットのデータ量にばらつきがあってもノードの負荷は平準化しやすい．機械加工ジョブにおいて，小ロットに分割して実行すると，負荷が平準化してジョブ終了時刻改善効果があるとの報告がある[10]．また，従来では高信頼サーバでの動作を前提とするため，ハードウェアの低コスト化が困難であったが，ハードウェア障害時に実行中のタスクを他ノードで代替実行することで，多数分割によって代替実行されるタスクの処理時間も減少するため，ハードウェアに対する耐障害性を改善し，高信頼サーバでなくても基幹バッチ処理運用が可能となる．

2.3 グリッドスケジューラの基幹バッチへの適用における課題

しかし，既存のグリッドスケジューラは，基幹バッチ処理のスケジューラとしては以下の課題があり，2.1節に示した要件を満足できていない．

(1) **自動回復不可能障害回復処理の煩雑化**

ジョブの障害要因と障害要因に対するグリッドスケジューラの耐障害機能を表1に示す．グリッドスケジューラは，スケジューラサーバの障害＃1〜＃2は多重化で，自動回復する可能性がある障害＃3〜＃6は他ノードへの代替実行で回復を試みる．多重化や代替実行は，Grid Engine[5]，LSF[6]，MapReduce[7]などで実装されている．

しかし，AP・データ・設定不良などを要因とした障害＃7〜＃10は，再実行しても正常終了せず代替実行では回復できない確率が高い．回復できない場合は，終了状態確認，障害要因解析，記述修正，再実行指示など一連のユーザアクションを必要とする．このため，グリッド化により分割数・ロット数・タスク数が増加することで，監視・操作が複雑化し，結果として障害回復処理が遅延する懸念がある．

なお，障害＃3〜＃6でもAP実行中にリソースを更新するジョブである非べき等ジョブは，状態が変っ

たまま代替実行すると事項結果が変ってしまう．障害＃ 11〜＃ 12 は，再実行することで正常終了する可能性があるが，再実行しても正常終了する確率の低い障害＃ 8〜＃ 10 と現象が同じタスク異常終了であり，メッセージを詳しく解析しないと，他の障害と区別できない．このため，これらの障害も代替実行では回復できない．障害＃ 7〜＃ 12 と非べき等ジョブの障害＃ 3〜＃ 6 を，自動回復不可能障害と定義し，本論文の障害運用支援機能の対象とする．

(2) **ジョブ横断型大量データ処理ワークフローへの対応**

基幹バッチに多いジョブ横断型大量データ処理のワークフローで自動回復不可能障害が発生すると，変動分割によりデータのロットとジョブのタスクとの対応関係把握が複雑化し，操作ミスを誘発する要因となる．

(3) **操作単純化と再実行負荷削減の両立**

終了時刻を厳守し，操作ミスを低減するには，再実行指示などのインタフェースを単純化しユーザアクションを削減する必要がある．しかし，単純化するためにタスクの異常終了を検出した時点でワークフローを中断させてしまうと，再実行の負荷が軽減されず，終了時刻を超えるリスクが高まる．

(4) **既存バッチ処理からの低コスト移行**

既存のグリッドスケジューラでは，新規構築を想定としているものが多く，既存基幹バッチ処理の資産の変更量削減が考慮されていない．

次章では，基幹バッチ処理のためのグリッドスケジューラの耐障害機能向上手法として，上記課題を解決し，ジョブ横断型大量データ処理ワークフローにおいて自動回復不可能障害が発生しても再実行時間・障害復旧時間（ジョブの障害発生から正常終了までの時間）を短縮して障害による遅延を抑制し，規定終了時刻までにバッチ処理を終了可能な確率を向上させる終了時刻厳守要件のための障害運用支援機能を提案する．

表 1 ジョブ障害要因とグリッドスケジューラの耐障害機能
Table 1 Cause of fault job and fault tolerance function of grid scheduler

#	障害部位および障害要因		耐障害機能
1	スケジューラサーバ	ハードウェア	スケジューラ多重化・ホットスタンバイ
2		OS, スケジューラ	
3		ネットワーク	gent 監視，再接続
4	ノード	ハードウェア	他ノード代替実行
5		OS	
6		Agent	
7		AP(無限ループ)	ジョブ異常終了・強制終了（ユーザが障害要因を解析し，回復後再実行）
8		AP(異常終了)	
9		データ設定	
10			
11		リソース不足	
12	ストレージ		

3. グリッドスケジューラの自動回復不可能障害運用支援機能の提案

3.1 基幹バッチ処理向けグリッドスケジューラ uGPS の構造

uGPS(uCosminexus Grid Processing Server)[11] は，超多重度分割手法によるジョブ実行に必要な，基幹バッチ処理向けに開発したグリッドスケジューラである．2.3 節の課題（1）〜（3）で示した課題を解決するため，従来のグリッドバッチと同等の耐障害機能のほかに，3.2 節にて示す自動回復不可能障害を対象とした障害運用支援機能を実装している．

uGPS では，課題 (4) の低コスト移行課題を解決するため，データ分割・ジョブ複製によるジョブレベルのデータ並列手法をとった．少数固定分割手法で構築されたジョブであれば，並列化はされているため，分割方法変更に伴って修正が必要なデータ分割処理や集計処理の変更程度のソースコード変更ですむ．データ並列用 API やライブラリによる関数レベルのデータ並列手法は，データ並列用 API を用いてソースコードを書き換える必要があり，採用しなかった．ジョブレベルの並列化は，関数レベルの並列化よりも高い粒度が必要となるが，基幹バッチが入出力するレコード数は数 100 万〜数億レコードに達するため，問題ないと考えた．

uGPS の構造を図 2 に示す．スケジューラサーバ内のワークフロー管理によって起動されたタスク管理プロセスが，タスクを生成して，各ノード内の agent プロセスにタスクの実行を指示する．タスクは，2.2 節記載の超多重度分割手法を用いて多重度未満のノードにスケジュールする．ノード内の実行中のタスク数を agent から収集し，タスク管理プロセス側で管理する．多重度と比較して，タスク実行数が多重度未満となったノードに未実行のロットの処理実行を依頼する．障害が発生したノードで実行していたタスクは，最後に各ノードに分散して再実行する．

ノード管理デーモンは，アクティブなノードを把握してタスク実行を指示するノードを指定する．ジョブ定義モジュールは，ジョブ内で実行する AP の名称，ロット数，ジョブの正常・異常判定条件などのジョブ動作設定を定義するインタフェースを提供する．タスク状態表示モジュールは，タスク単位の状態表示や操作を行う．

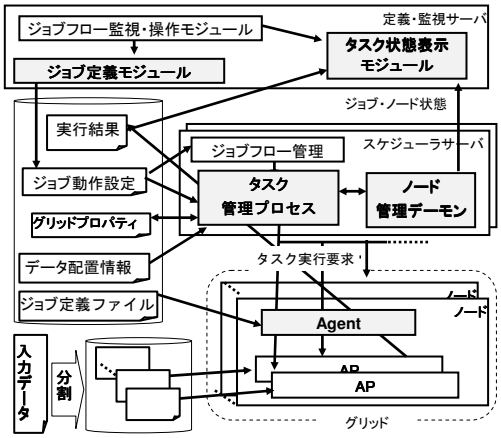

図 2 基幹バッチジョブワークフローの例
Fig. 2 Example of mission critical batch job workflow.

3.2 uGPS の自動回復不可能障害運用支援機能

文献[12] によると，2.3 節で示した自動回復不可能障害に属する AP 障害と設定ミスを合わせると，金融機関のシステム障害要因 (1000 件/年) の 50 %以上に達する．このため，バッチ処理の終了時刻を守るためには，代替実行などの自動回復可能障害回復技術だけでなく，自動回復不可能障害の復旧時間短縮支援技術が必要である．しかし，グリッド化により分割数が増加する結果，障害回復のための監視・操作インタフェースが煩雑化する．特に，ジョブ横断型大量データ処理ワークフローでは，同一ロットを複数ジョブに渡って処理するため，ロット別に加えてジョブ別でも監視・操作しなければならず，インタフェースが煩雑化する．

障害回復時間の縮減のためには，障害時運用容易化によるユーザの調査・操作時間の短縮と，障害に影響されない処理の継続による再実行負荷軽減が重要である．そこで，ロット数・ジョブ数増加による障害時運用悪化防止を目的として，以下の障害運用支援機能を提案する．

(1) **データ視点フロー管理**

大量データ処理のワークフローにおける進捗状態を，データを視点として管理する．データであるロットごとにロットを一意に識別するロット識別子を設定し，ロット識別子ごとに，そのロットを割り当てて実行したタスクの終了状態を管理する．タスクを異常終了後に再実行すると同一ロットを割り当てた複数のタスクが存在することになるが，この場合，最後に実行したタスクの終了状態をロットの状態として管理する．また，ロット識別子ごとにそのロットが最後に実行されたジョブの名称を記憶し，各ロットがワークフロー上でどこまで進んだかを把握し，データ完全性 (全ロットの正常終了) を確認する．ワークフローが異常終了により中断した場合，まずロット一覧からロットを特定し，次にロットを割り当てたタスクを特定してタスクの障害要因を調査する．

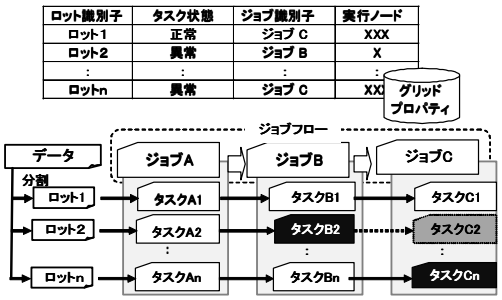

図 3 グリッドプロパティの構造
Fig. 3 Structure of grid property

ロットごとのタスク終了状態は，グリッドプロパティと呼ぶワークフローごとのファイルに保持する．グリッドプロパティの構造を図 3 に示す．グリッドプロパティのロットごとの各エントリには，ロット識別子と，そのロットを最後に実行したジョブの識別子と，そのロットを割り当てて実行したタスクの終了状態とを記録する．グリッドプロパティは，ワークフロー内でジョブを初めて実行するときに作成する．ジョブを再実行する場合は，グリッドプロパティは既存ファイルを更新する．実行されたタスクのロットのエントリのみを更新する．また，グリッドプロパティには，ロットを実行したノード識別子も記録する．後続ジョブを実行するときに，本情報を参照して記録したノードで実行させることで，メモリにジョブ間で受け渡すデータを格納して高速化したり，ノード間のデータ転送を削減したりすることができる．

ワークフローにおける進捗や状態を管理する目的に対しては，タスク単位に管理するアプローチがある．タスク単位の場合，再実行すると同一ロットを割り当てたタスクが複数生じるため，全ロットが正常終了したか否か，未実行のロットがないか確認することが，ロット数が多く再実行を繰り返すほど難しくなる．また，タスクに対するロットをメッセージから特定しなければならず，正常終了したタスクに割り当てたロットに対する後続ジョブを実行してしまうと，タスク未割当ロットを指定するのにロットを特定する必要があり，再実行指示インタフェースが複雑になってしまう．かといって，ひとつでもタスクが異常終了したジョブでワークフローを中断してしまうと，再実行時にチェックポイントジョブからすべてのデータの処理を再実行しなければならず，終了時刻を超えるリスクが増加し

てしまう．
このため，分割数が多いジョブ横断型大量データ処理のワークフローの進捗状態は，タスク単位ではなく，再実行が生じても管理しやすいロット単位で管理すべきである．

(2) **再実行範囲局所化**

異常終了したジョブを含むワークフローが終了しワークフローを再度実行する時の負荷を低減して実行時間を短縮するため，タスク異常終了が起こっても，ワークフローは中断させず，正常終了したタスクのロットのみを対象として後続ジョブを実行する．ワークフローを再実行するときは，グリッドプロパティに記録したロット単位のフロー状態情報を参照して，再実行対象のロットをスケジューラが自動判別し，タスクに割り当てて実行する．再実行指定がジョブを指定するだけでよいため，ユーザアクションに要する時間短縮と，操作ミス低減を図る．必要な処理だけを実行するため，再実行時間を短縮する．ジョブを実行するときに，グリッドプロパティ内のジョブ識別子が当該ジョブの識別子でデータ状態が「正常」でないロットと，ジョブ識別子が先行ジョブの識別子で状態が「正常」であるロットとを選択して実行する．ジョブ完了時にグリッドプロパティを更新して後続ジョブを続けて実行するため，先行ジョブでタスクが異常終了したために実行されなかったロットも実行する．このように，いくつかのタスクが異常終了して正常終了したロットのみを実行することで，ロットのフロー実行状態が不均一になっても，ユーザはジョブ実行を指示するだけで，実行が必要なロットのみが実行され，ロットの処理漏れを防止することができる．
後続ジョブの入力となる先行ジョブの出力データを，出力性能向上のため RAM ディスクなどの非共有または揮発性のストレージに格納すると，後続ジョブを異常終了後に再実行しようとしても，入力データにアクセスできず実行できなくなる．このような場合，入力データを参照可能なジョブから実行する必要がある．このため，異常終了したジョブを指定して再実行しても，入力データにアクセス可能な先行ジョブから再実行する機能を提案する．この先行ジョブ（入力データが不揮発・共有ストレージにあり再実行時もアクセス可能なジョブ）をチェックポイントジョブと定義する．チェックポイント以降の先行ジョブは正常終了しているが，異常終了したジョブの入力データを生成するために再度実行される．実行するジョブがすべてべき等ジョブであれば，正常終了したジョブを再度実行しても支障はない．データ分割によりタスクあたりの処理時間が短くなるため，適切な粒度に分割すれば，再実行による遅延を許容でき，通常実行時の性能を優先することが可能となる．
MapReduce[7] の Map タスクからの再実行機能に比較すると，再実行開始タスクが固定されていないため，さまざまなワークフロー構造に適用でき，通常時の実行性能と再実行時間とのバランスのとれた再実行開始ポイントを設定することができる．

(3) **ロット一括操作インタフェース**

表示や操作インタフェースをジョブレベルとタスクレベルの 2 階層構造にし，スケジューラ内部で内部情報をもとにタスクレベルに分解する．通常はロットやタスクを意識せずに監視・操作を行い，障害時に特定のロットやタスクを意識したいときだけロットレベルで表示・操作を行う．これにより，障害時運用におけるユーザアクションを削減し，操作ミスリスクを低減する．
ジョブ全体の状態は，すべてのロットを処理したタスクが正常終了した場合は「正常」，異常終了のロットを処理したタスクがある場合は「異常」として表示する．すべてのタスクが正常終了しても，先行ジョブにてタスクが異常終了したために，後続ジョブにて実行することができないロットがあることがわかるように，未実行のロットがある場合は「警告」として表示する．
ジョブの詳細状態は，タスク単位ではなくロット単位に表示し，タスク実行順ではなくロット識別子順に表示する．再実行などで同一ロットのタスクが複数ある場合は，最新の結果のみを表示する (ただし，最新でないタスクも実行履歴は残すので，障害履歴は保存できる)．
ロットの状態表示画面であるロットビューの例を図 4 に示す．ロットビューでは，ロット識別子・ロットを割り当てて実行したタスクの名称・終了状態・実行ノードなどのロット単位の状態・実行履歴を行とした表形式で表示する．
ロットの表示順は，ロット識別子以外に，状態や実行したノード順に並び替えできるので，異常状態のロットを容易に判別することができる．ロット単位のメッセージ表示や再実行指示も，この画面でロットを指定することで操作する．
なお，ロットビューは情報量が多いため，障害監視・判断容易化のため，通常時は表示せず，障害時などロットごとに調べる必要があるときのみ表示する．通常時は，ジョブの状態を表示する画面であるジョブビューを表示する．ジョブビューでは，ジョブ内で異常終了

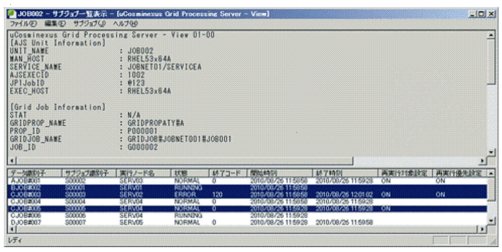

図 4 ロットビュー画面の表示例
Fig. 4 View example of lot view window

または先行ジョブ異常終了により未実行のロットがあれば異常または警告状態で表示する．

4. 評　　価

4.1 自動回復不可能障害運用支援機能の評価

自動回復不可能障害運用支援機能であるデータフロー視点フロー管理・再実行範囲局所化の効果は，自動回復不可能障害発生時のジョブ実行時間削減効果によって評価する．ただし，自動回復不可能障害発生時のジョブ実行時間は，障害要因・ジョブ実行時間・オペレータの習熟度などの終了時刻に影響する条件の個別変動幅が大きく，特定ケースでの評価は不適切と思われる．このため，以下のようにモデル化して評価する．

自動回復不可能障害発生時のワークフロー実行時間 Tx は，以下の式であらわすことができる．ワークフローは，先行ジョブ A と後続ジョブ B の 2 つから構成されると仮定し，ジョブ A の実行中に自動回復不可能障害が発生したと仮定する．

$$T_x = A_x + B_x + C_x + D_x + E_x + F_x + A_x^{'} + B_x^{'} \quad (1)$$

A_x =先行ジョブ A の実行時間（障害が発生し異常終了するまで
B_x =後続ジョブ B の実行時間（障害前）
C_x =障害要因調査時間
D_x =障害要因除去・修正時間
E_x =特定のタスクのみを再実行するための定義修正時間
F_x =再実行指示時間
$A_x^{'}$=先行ジョブ A の再実行時間
$B_x^{'}$=後続ジョブ B の実行時間
各時間の添え字 x は，方法を識別する番号である．障害が発生せずに終了した場合を x=0 とする．ワークフロー実行時間 T0 は，以下の式で求められる．

$$T_0 = A_0 + B_0 \quad (2)$$
$$(C_0 = D_0 = E_0 = F_0 = A_0^{'} = B_0^{'} = 0)$$

タスクが異常終了したジョブで実行を中断し，ジョブ全体を再実行する従来方法 1 を x=1，定義を修正してジョブの一部を再実行する従来方法 2 を x=2 とする．本論文の提案方法を x=n とする．時間 A_x・$A_x^{'}$ には，以下の式が成り立つ．

$$A_0 > A_1, A_0 > A_2, A_0 > A_n \quad (3)$$
$$2 * A_0 > A_1 + A_1^{'} > A_2 + A_2^{'} = A_n + A^{'}n > A_0, \quad (4)$$
$$A_0 = A_1^{'} > A_2^{'} = A_n^{'}$$

異常終了した場合のジョブ実行時間 A_1，A_2，A_n は，正常終了したときのジョブ実行時間 A_0 より短いが，再実行時のジョブ実行時間 $A_1^{'}$，$A_2^{'}$，$A_n^{'}$ と合わせた時間は，全ロットを再実行する従来方法 1 より，一部のみを再実行する本提案方法と従来方法 2 のほうが短く，A_0 に比べて異常終了したロットの処理時間しか長くならない．時間 B_x・$B_x^{'}$ には，以下の式が成り立つ．

$$B_0 > B_n, B_1 = B_2 = 0, B_n + B_n^{'} = B_0 = B_1^{'} = B_2^{'} > B_n^{'} \quad (5)$$

本提案方法は，異常終了前に，正常終了したタスクが処理したロットに対する後続ジョブも実行するため，後続ジョブの実行時間 Bn は 0 ではないが，その分再実行時間 B'n が短くなるため，トータルの実行時間は他の方法と同等である．時間 Cx の障害要因調査は時間 Bx と同時にできるため，その分短縮することができる．時間 C_x および D_x は，各方法の差はない（C1　C2　Cn, D1　D1　Dn）．時間 E_x および F_x には，以下の式が成り立つ．

$$E_2 > E_1 = E_n = 0, F_2 > F_1 \approx F_n \quad (6)$$

従来方法 2 では，定義をユーザが修正するため，時間 E2 が必要である．ロットごとに操作が必要なため，時間 F_2 も一括操作できる本提案方法より長い．以上総合すると，以下の式となる．

$$\begin{aligned} T_n - T_1 = & \\ & (A_n + A_n^{'} + E_n + F_n + (B_n + B_n^{'} + C_n + D_n)) \\ & -(A_1 + A_1^{'} + E_1 + F_1 + (B_1 + B_1^{'} + C_1 + D_1)) \\ = & (A_n + A_n^{'} + E_n + F_n) - (A_1 + A_1^{'} + E_1 + F_1) \\ = & ((A_n + A_n^{'}) - (A_1 + A_1^{'})) \\ & +((E_n + F_n) - (E_1 + F_1)) \\ \approx & ((A_n + A_n^{'}) - (A_1 + A_1^{'})) < 0 \end{aligned} \quad (7)$$

$$
\begin{aligned}
T_n - T_2 = & \\
& (A_n + A_n^{'} + E_n + F_n + (B_n + B_n^{'} + C_n + D_n)) \\
& -(A_2 + A_2^{'} + E_2 + F_2 + (B_2 + B_2^{'} + C_2 + D_2)) \\
= & (A_n + A_n^{'} + E_n + F_n) \\
& -(A_2 + A_2^{'} + E_2 + F_2) \\
= & ((A_n + A_n^{'}) - (A_2 + A_2^{'})) \\
& +((E_n + F_n) - (E_2 + F_2)) \\
= & (E_n - E_2) + (F_n - F_2) < 0 \qquad (8)
\end{aligned}
$$

このように，本提案方法では，従来方法に比べて，再実行まで含めたジョブ実行時間 A_x+B_x+$A_x^{'}$+$B_x^{'}$ を正常時に比べて異常ロットの実行時間分だけに抑えながら，再実行のための運用時間 Ex+Fx をロット数によらず削減できるため，自動回復不可能障害発生時のジョブ実行時間を削減することができる．

4.2 超多重度分割手法の評価

自動回復可能障害発生時のジョブ実行時間の性能評価を実施した．典型的な基幹バッチ処理である，取引データとマスタデータ (ともにレコード長 80Byte 固定，総データ長= 100KB) の突き合わせ処理のジョブ実行時間を測定した．マスタデータは複製して各ノードのローカルストレージに事前配置し，取引データとなる他方を事前に分割して共有ストレージに配置した．以下の 2 つの方法で分割したデータを処理するタスクを 8 多重で実行し，比較する．超多重度分割手法では，ノード障害事象を疑似発生させるため，ジョブ実行中に 1 つのノードの agent を強制終了させた．(a) 少数固定分割手法 多重度数と同数の 8 ファイルに分割 (各ファイル= 12.5KB) (b) 超多重度分割手法 多重度を超える 20 ファイルに分割（1 ファイル= 5KB）タスクを実行したノードのスペックを以下に示す．

CPU	Athlon4450B(2.3GHz)
メモリ容量	2GB
OS	CentOS5.1

少数固定分割手法の測定結果（各ノードのタスク実行時間）を図 5 に，超多重度分割手法の測定結果を図 6 および図 7 に示す．超多重度分割手法の実行時間 (図 6) が少数固定分割手法の実行時間 (図 5) が長くなっているのは，超多重度分割手法ではジョブ数が多重度の整数倍でなかったためである．最後のタスク実行時にはアイドル状態のノードが発生するため，処理時間は単純多重度より遅れる．

超多重度分割手法では，正常時の結果（図 6）と，ノード 1 の最初のタスク実行時にノード障害を疑似発生させた結果（図 7）の実行時間はほぼ等しい．障害ノードで実行する予定であったロットを，最後にアイドル状態のノードで実行するためである．超多重度分割手法では，1 ノード障害では正常時の 1〜（1+（↑ロット数/多重度↑）/多重度）倍 (8 多重 20 ロットでは 133 倍) の実行時間で実行可能である．一方，少数固定分割手法では最大 2 倍になる．これが超多重度分割手法による再実行時間短縮効果である．

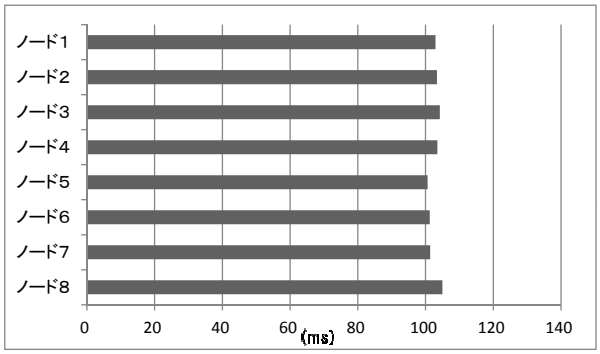

図 5 少数固定分割手法のジョブ実行時間 (正常時)
Fig. 5 Job execution time in normal case with fixed & limited partitioning.

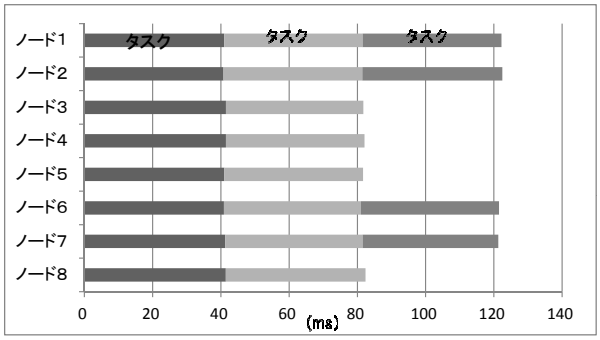

図 6 超多重度分割手法のジョブ実行時間 (正常時)
Fig. 6 Job execution time in normal case with over-multiplicity partitioning.

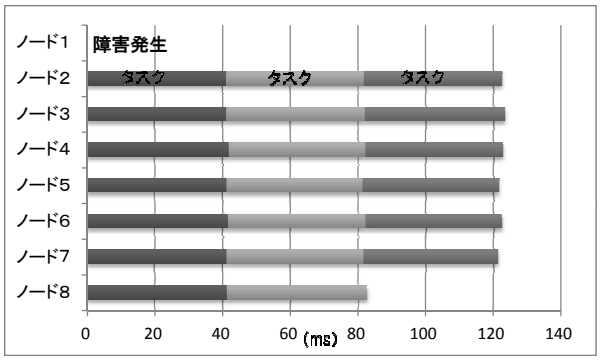

図 7 超多重度分割手法のジョブ実行時間 (障害時)
Fig. 7 Job execution time in fault case with over-multiplicity partitioning.

5. 関連研究

耐障害機能を備えたグリッドスケジューラや分散並列処理フレームワークとしては，Grid Engine[5]，LSF[6]，MapReduce[7]，Hadoop[8]，Ninf[13]，Condor MW[14]，Ninf-C[15] があげられる．Grid Engine[5]，LSF[6]，MapReduce[7]，Hadoop[8] は，他ノードでのタスク自動代替実行技術を実現している．この技術は，正常な他のノードで実行すれば回復可能なハードウェア障害には有効であるが，本研究が対象とした自動回復不可能なソフトウェア障害は対象外である．Ninf[13] が提案している同一タスクを複数ノードで多重実行する手法も，他のノードで実行すれば正常終了する可能性がある障害に効果が限られる．

Condor MW[14]，Ninf-C[15] では，チェックポイントを記録し，タスクを途中から再実行する技術を実現している．このようなチェックポイントを用いた耐障害機能は，チェックポイント以降のタスクは再実行するまで実行されないため，一部のタスクのみが異常の場合でも，ジョブ全体が中断してしまう．また，複数のジョブから構成される場合，先行ジョブで障害になると，後続ジョブは実行されない．このため，障害が発生しても正常なロットの処理は継続する本研究の手法と異なり，早い段階で中断するほど，ジョブの再実行負荷が大きくなり，規定終了時刻を超えるリスクは高まる．

そのほか，Hadoop[8] で実現しているエラーレコードスキップ機能[8] も耐障害機能の１つであるが，データ完全性が要求される基幹系への適用は容易でない．

また，ワークフロー制御を備えたグリッドスケジューラとしては，Condor DAGMan[3]，Pegasus[16]，Dryad[17] があげられる．これらは，非循環ワークフローの表現方法である DAG(Directed Acyclic Graph) を用いたワークフロー制御技術を実装している．これらのワークフロー制御は，ワークフロー内の各ジョブの順序制御やタスク並列制御を目的としており，本研究のような複数のジョブにまたがるデータ並列制御は考慮されてはいない．また，MapReduce[7] も Map ジョブと Reduce ジョブから構成されるワークフローとも見えるが，フロー構成が限定的であり制約が大きい．

6. おわりに

基幹バッチ処理向けのグリッドスケジューラである uGPS を，既存システムからの低コスト移行を重視し，ソースコード・定義・運用への影響が少ない従来スケジューラへのジョブレベル並列機能追加により実装した．超多重度分割手法を用いることで，障害時のジョブ終了時刻改善効果があることを示した．

基幹バッチの最大要件である終了時刻厳守のため，障害回復迅速化のための機能を重視した．従来のグリッドスケジューラの耐障害機能である代替実行ではカバーできていない自動回復不可能障害の運用支援機能として，データ視点によるフロー管理と再実行範囲局所化の機能・インタフェースを提案した．これにより，グリッド化に伴いデータやジョブの分割数が増加しても，基幹バッチ特有のジョブ横断型大量データ処理ワークフローでも，再実行のための運用時間や再実行負荷を増加させず，障害による遅延を短縮できることを示した．

参考文献

1) 宮澤貴之，鈴木俊也，福丸昌宏: オープン勘定系システムにおけるバッチ処理を支える仕組み，ユニシス技法 96 号 (2008).
2) Ian Foster, Carl Kesselman: Globus: A Metacomputing Infrastructure Toolkit. Intl J. Supercomputer Applications,11 (2), pp. 115–128 (1997).
3) Douglas Thain, Todd Tannenbaum, and Miron Livny: Distributed Computing in Practice: The Condor Experience Concurrency and Computation: Practice and Experience, Vol.17, No. 2-4, pp. 323–356 (2005).
4) 白坂純一: グリッド協議会金融分科会の活動について, グリッド協議会 第 26 回ワークショップ（オンライン）, http://www.jpgrid.org/event/2009/pdf/ws26/ws26-shirasaka.pdf, 2009
5) W. Gentzsch: Sun Grid Engine:Towards Creating a Compute Power Grid, Cluster Computing and the Grid, IEEE International Symposium on, pp. 35–36 (2001).
6) Ian Lum,Chris Smith: Scheduling attributes and platform LSF, Grid resource management, pp. 171–182, ISBN1-4020-7575 (2004).
7) Jeffrey Dean, Sanjay Ghemawat: MapReduce:Simplified Data Processing on Large Clusters, USENIX OSDI'04, pp. 137–150(2004).
8) Tom White: Hadoop:The Definitive Guide, First Edition, ISBN978-4-87311-439-2, 2010
9) 細川努: アーキテクチャと性能,IT アーキテクト Vol. 22 pp. 31,ISBN978-4-87280-293-1, 2009
10) 金指 正和, 三谷 崇: ジョブのロット分割を考慮したフローショップスケジューリング問題の解法, 日本経営学会論文誌, Vol. 53, No.3, pp. 241–249 (2002)
11) 仲田智将, 藤井研一, 石合秀喜, 渡辺和彦: エン

ータープライズグリッドによる次世代オープンアーキテクチャ, 日立評論 2010 年 5 月号 ,2010
12) 日本銀行金融機構局: 金融機関におけるシステム障害に関するリスク管理の現状と課題, 日本銀行（オンライン）, $http://www.boj.or.jp/research/brp/ron_2010/data/ron1011a.pdf$, 2010
13) 川上健太, 合田憲人: 並列分枝限定法における耐故障アルゴリズムの評価, 情報処理学会研究報告 2005-HPC-103(9), pp. 49–54(2005).
14) Jean-Pierre Goux, Sanjeev Kulkarni, Jeff Linderoth, MichaelYoder: An Enabling Framework for Master-Worker Applications on the Computational Grid, In Proceedings of HPDC-9, pp. 43–50 (2000).
15) 中田秀基, 田中良夫, 松岡聡, 関口智嗣: 耐故障性を重視した RPC システム Ninf-C の設計と実装, 先進的計算基盤システムシンポジウム（SACSIS）論文集 (2004).
16) Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi and Miron Livny: Pegasus: Mapping Scientific Workflows onto the Grid, Proceedings of the Second European Across Grids Conference, pp11-20 (2004).
17) Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly: Dryad: Distributed Data- Parallel Programs from Sequential BuildingBlocks, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007,pp59-72(2007).