Vol.2017-HPC-162 No.8
2017/12/18

LWK のバッチジョブ運用に向けた 運用ソフトウェアの設計と試作

二宮温[†1] 平井浩一[†1] 小田和友仁[†1] 岡本高幸[†1] 住元真司[†1] 松山佳彦[†1]
高木将通[†2] Balazs Gerofi[†2] 小倉崇浩[†2] 亀山豊久[†2]
堀敦史[†2] 石川裕[†2]

HPC 分野におけるメニーコア化などのハード資源構成の多様化に対応し, アプリケーション実行環境の最適化を目的として, 近年様々な Light-Weight Kernel（LWK）が提案されている. 本論文では, 大規模共用環境におけるバッチジョブ運用で LWK を利用するための運用ソフトを設計した. また, LWK として McKernel を取り上げ, 物理連続メモリ割り当て, デリゲーション処理の資源制限, 運用ソフトの導入に伴う LWK の外乱の防止の３つの課題を取り上げ, 対策が有効に働くことを評価し, LWK のバッチジョブ運用が実現できることを確認した.

Design and Prototyping of Batch Job System Environment for LWK

ATSUSHI NINOMIYA[†1] KOUICHI HIRAI[†1] TOMOHITO OTAWA[†1]
TAKAYUKI OKAMOTO[†1] SHINJI SUMIMOTO[†1] YOSHIHIRO MATSUYAMA[†1]
MASAMICHI TAKAGI[†2] BALAZS GEROFI[†2] TAKAHIRO OGURA[†2]
TOYOHISA KAMEYAMA[†2] ATSUSHI HORI[†2]
YUTAKA ISHIKAWA[†2]

1. はじめに

近年, ハードウェア資源構成の多様化, ソフトウェア環境の複雑化に伴い, HPC アプリケーションの実行環境は大きく変わりつつある. これまでの物理マシン上でのアプリケーション実行から, クラウド環境に近い仮想マシン[1][2][3]に加え, Linux コンテナなど OS レベル仮想化環境[4][5][6], 更には, Light-Weight Kernel（LWK）のような CPU とメモリを物理的に隔離利用しながら, 既存 Linux とシステム処理を共用する OS[7] が登場し, これらの実行環境を統合的に扱える運用ソフトウェアが求められている.

「京」コンピュータの後継機として開発を進めているポスト京[8]においても, システムソフトウェアとアプリケーションのコデザインに基づくアプリケーション最適化のため, 通常の Linux ベースの OS の他に McKernel[9][10]とよばれる LWK を導入する.

ポスト京では, バッチジョブ運用のもとにユーザに対し Linux のみで動作する環境（Linux オンリーモード）と McKernel を使った環境（LWK モード）の 2 種類の環境を提供し, ユーザはジョブに適したモードを選択可能となる. そのため, 定常的に計算資源を McKernel 用に予約することはできず, 通常の Linux 上のジョブと McKernel を実行時に再起動を伴わず切り替えて運用可能であることが求められる. このように, ポスト京のような大規模共用環境上でのバッチジョブ運用のもとで McKernel を含めた LWK を利用するには様々な課題があり[11], それにむけたバッチジョブの運用ソフトウェアの設計が必要である.

†1 富士通株式会社
FUJITSU LIMITED
†2 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science

2. LWK とバッチジョブ運用の概要

本章では, LWK とポスト京で採用する McKernel の概要について述べた後, バッチジョブ運用の概要について述べる.

2.1 LWK の概要

LWK は, HPC におけるハードウェア資源構成の多様化やアプリケーション複雑化に対し, アプリケーション毎に特化した OS を用いて最適な実行環境を提供することを目的とした軽量 OS である. 様々な種類の LWK が提案されているが, 本論文では, Linux とシステムの資源や処理などを分担して連係動作するハイブリッド型の LWK を対象とする. ハイブリッド型の LWK の特徴は, 隔離した実行環境上での軽量 OS 実行という LWK の利点を保持しつつ, システムコール処理を Linux と連動して処理することで, HPC のスタンダード OS となっている Linux との API 互換性を高められるという特徴がある. これにより LWK 上で実行可能なアプリケーションが増加し, 既存の Linux ユーザにも使いやすい OS となっており, ポスト京など共用システムでの導入にも適している.

ハイブリッド型の LWK の主要なものとしては, FusedOS[12], McKernel, mOS[13], FFMK(L4)[14], Hobbes(Kitten)[15]があり, 次のような特徴を持つ. これらの詳細については論文[7]を参照されたい.

(1) Linux との並存

物理ノード内で，Linux と LWK が平行して起動する．Linux が起動した後に，Linux 上から LWK を起動する方式や，LWK を単体で起動した後，仮想マシンとして Linux を起動する方式がある．

(2) メモリ，CPU 資源の分離

独自のメモリ，CPU 管理を可能とするため，LWK 向けの資源をパーティション分割して管理するため，Linux 上からは LWK 向け資源は見えなくなる．そのため，連携動作する Linux が動作するためのメモリ，CPU が別途必要である．

(3) Linux のシステムコール処理を利用

Linux のシステムコール処理を利用するために，Linux 側にも LWK との連携処理が必要である．カーネルモジュールとモニタプロセスを Linux 上に動作させる方式，Linux を改変し LWK を埋め込む方式などで実施されている．

2.2 McKernel の概要

McKernel は，理化学研究所が中心となって開発が進められているハイブリッド型の LWK である．McKernel の利用シーンの例としては，大規模並列環境でのスケーラビリティ向上にむけたノイズレス環境の提供，独自のメモリ管理による Linux が提供していない複数のラージページサイズ利用などが挙げられる．

McKernel は，Linux との間のカーネル間インターフェース IHK（Interface for Heterogeneous Kernels）を持ち，Linux 上のカーネルモジュールを介して，McKernel の制御やシステムコールデリゲーション処理を実現している．

McKernel をバッチジョブスケジューラと連携させる場合には，以下の３つを考慮する必要がある．

2.2.1 McKernel に割り当てる CPU 及びメモリの分離

McKernel は，Linux が管理している CPU，メモリから，必要な分だけ動的にパーティション分離し，それらの資源上でカーネルを起動する方式をとる．また，McKernel 終了後に再度 Linux に資源を返却する．従って，Linux はノード上に常駐して全ての CPU，メモリ資源を管理し，必要なときに McKernel に貸し出す形態となる．

2.2.2 McKernel の起動・終了操作

前述の CPU とメモリの確保を含め McKernel の起動及び終了の流れを説明する．全体としては，以下の(1)から(7)のような流れで IHK のインターフェースを実行する．

(1) CPU 及びメモリの確保，McKernel 用資源領域の作成
(2) McKernel バイナリのロード
(3) McKernel の起動
(4) McKernel 上でのプロセス実行
(5) McKernel の終了
(6) McKernel 用資源領域の解放，Linux への資源の返却

特徴としては，(2)で動的にバイナリのロードを行うため，起動毎にバイナリを選択することができる．また，(1)で Linux から確保した資源をさらに複数の領域に分割して，それぞれ異なった McKernel に割り当てることも可能である．

2.2.3 システムコールデリゲーションと Proxy Process

McKernel はアプリケーションに対し，Linux API 互換のシステムコール機能を提供する．これは 300 以上存在する Linux システムコールの大部分を，Linux にデリゲーションすることで実現している．McKernel 上でプロセスを起動すると同時に，Linux 上でもそのプロセスのシステムコールを代行するプロセス（Proxy Process）を起動する．2.2.2 節の（４）で起動するプログラム（mcexec）がこのプロセスとなる．

McKernel 上のプロセスからシステムコールが発行されると，その処理は McKernel 内部で実行されるか，または Linux の Proxy Process にデリゲーションされる．McKernel の管理する資源に対する処理（brk, mmap など）や高速な応答が必要とされる処理（futex など）は McKernel 内部で直接処理される．それらを除いた大部分のシステムコール（ファイル操作系，ユーザ管理系など）は IHK を通して，Linux 上の Proxy Process にデリゲーションされる．

2.3 バッチジョブシステム

HPC の大規模クラスタの運用においては，多数のユーザの要求を効率よく実行するため，一般にバッチジョブ方式によるシステム運用が採用されている．本章ではバッチジョブ運用の目的と動作について述べる．

2.3.1 バッチジョブ運用の目的

バッチジョブ運用とは，ジョブスケジューラがクラスタを一元管理し，ユーザのジョブ実行要求を保持して運用ポリシにもとづき，順次実行する方式である．ユーザが投入するバッチジョブを管理し，運用するシステムをバッチジョブシステムと呼ぶ．バッチジョブシステムは一般的に，以下の 3 つの要求を満たす機能を提供する．

(1) 使いやすいジョブ実行インターフェース

ジョブ実行ユーザが複雑なクラスタ構成を意識することなくジョブを実行し，ジョブの種別や並列度をオプションなどで指定可能とすることで，統一的なインターフェースでユーザが必要な実行環境を提供する．また，対話的に実行するインターフェースや，ジョブの性能情報を取得するインターフェースを提供することにより，大規模ジョブの性能チューニングなどを容易にする．さらに，ユーザは登録したジョブの実行状態や処理完了予定時刻などのジョブに関する管理情報を把握するインターフェースを提供する．

(2) ジョブ実行要求に対する適切なスケジューリング

多数のユーザのジョブ実行要求を即座に受け付け，適切なタイミングでクラスタ構成内の計算機の CPU やメモリなどの計算機資源を割り当て，ジョブを実行させるようなスケジューリングを行う．

(3) 計算機資源の効率的利用

クラスタ全体の計算機資源の利用状況を管理し，ジョブの要求資源や優先度などの属性を加味して適切に計算機資源を割り当てることで，クラスタ全体の資源の利用効率を最適化する．

2.3.2 バッチジョブ実行の流れ

図 1 に具体的なバッチジョブ運用におけるジョブ実行処理の流れの例を示す．

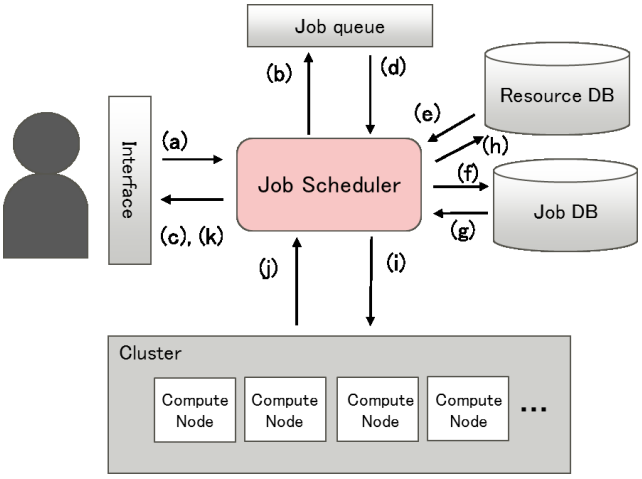

図 1 バッチジョブ実行の流れの例

以下では図 1 を用いてジョブ実行処理の流れを説明する．

(1) ジョブ実行要求受け付け処理

ユーザからのジョブ実行要求を受け付け(a)，ジョブ実行要求を Job queue に登録し(b)，ユーザに受け付け完了を通知する(c)．

(2) ジョブスケジューリング処理

Job queue からジョブ実行要求を取り出し(d)，計算機資源を管理するための Resource DB (Database)から資源利用状況を取得し(e)，実行予定時刻と割り当て資源を決定し，スケジューリング情報を Job DB (Database)に登録する(f)．

(3) ジョブ実行処理

Job DB からジョブスケジューリング情報を取得し(g)，Resource DB を更新しジョブ実行に必要な計算機資源を割り当て(h)，ジョブ実行を開始する(i)．ジョブが終了したら，ジョブの実行結果を回収し(j)，計算機資源を解放する．その後実行結果をユーザに通知する(k)．

本論文では，これらの処理のうち図 1. の Computing Node(計算ノード)内のジョブ実行処理(i)で実施される計算資源管理を対象とする．次節で Linux オンリーモードにおける計算資源管理の概要を述べる．

2.3.3 計算ノード内の計算資源管理の概要

各計算ノードには，ジョブ実行処理を実施するためのノード内ジョブマネージャが常駐し,ノード内の計算機資源を管理する．ノード内ジョブマネージャは，ジョブ実行処理(i)で次の処理を実施する．

(1) ユーザがジョブ投入時に指定する，要求メモリ量，要求コア数，NUMA ポリシに従って，CPU，メモリをジョブ単位に割り当て，占有利用可能とする．

(2) ジョブスクリプトと呼ばれるジョブ実行の手続きが記述されたシェルスクリプトを起動する．ジョブスクリプトのなかで，ジョブプログラムが起動される．

(3) ジョブの状態監視，定期的に統計情報の収集を実施する．

(4) ジョブスクリプトの終了を検知し，必要なジョブ情報を収集した後, CPU，メモリ資源を解放する．

図 2 にすべての計算機資源を利用して Linux を動作させている場合のジョブ資源管理方式の概要を示す．

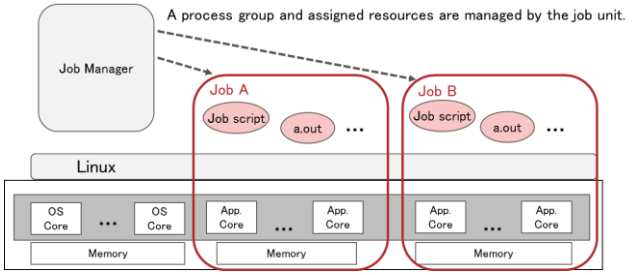

図 2 計算ノード上でのジョブ管理方式

3. 実現すべき運用ソフトウェアの要件と課題

3.1 要件

第 2 章で述べたように，ポスト京では，バッチジョブシステム運用の中で，各実行環境(LWK,Linux)を使ってアプリケーション性能を最大限発揮できる必要がある．

これを実現する運用ソフトウェアに対しては次の３つの要件を満たす必要がある．

（１）各実行環境で性能を引き出す資源割り当て
（２）実行環境間の干渉による性能影響の防止
（３）運用ソフトの導入に伴う性能影響の防止

これらの要件に対して，Linux 向けに実現されている運用ソフトをベースとし，ジョブの実行環境として LWK を追加する方針をとる．

3.2 要件に対する課題

前述した３つの要件に対し，アプリケーション性能に着目し，関連した設計課題を挙げる．要件（１）に対しては以下 3.2.1～3.2.3 節の課題が，また要件（２）および（３）には以下それぞれ 3.2.4, 3.2.5 節の課題がある．

3.2.1 CPU，メモリの排他利用

2.1 節に記載したように, LWK は CPU とメモリをパーティション分割して占有利用するため, LWK モードでは専用の CPU とメモリを割り当てる必要がある．そのため, CPU，メモリを LWK に排他的に割り当てる仕組みが必要である．

3.2.2 NUMA を意識した CPU，メモリ割り当て

LWK は資源のハード構成を意識し性能を引き出すことが目的である．NUMA 構成のハードウェアにおいては，同じ NUMA 内の CPU とメモリを割り当てないと性能にばらつきが発生するため，LWK に対しては NUMA ポリシを反映し同じ NUMA 内の CPU とメモリを割り当てる必要があ

る.

3.2.3 物理連続メモリの割り当て

単一の NUMA ノード上に複数の処理が混在した場合, 長時間運用を継続するとメモリの断片化が生じる. McKernel のような LWK は連続した物理メモリを割り当てられなくても動作は可能だが, LWK によっては Linux 側で提供されない複数のラージページを実現する場合があるため, LWK に対しては連続した物理ページを割り当てることが望ましい. そのため, メモリの断片化を抑止する必要がある.

3.2.4 デリゲーション処理に対する資源使用制限

LWK 自体は Linux からパーティション分離された資源を利用するが, McKernel のような LWK では, デリゲーション処理は Linux 上で動作するものがある. そのため, Linux 上で動作するデリゲーション処理, Linux オンリーモードのジョブ, およびシステム処理が互いに干渉しないようにする必要がある.

3.2.5 LWK の外乱とならない LWK のジョブ連動制御

LWK モードのジョブに対しても Linux と同じジョブ操作 IF(ジョブ投入, 中断, 統計情報参照, シグナル送信, 一時停止・再開など)が求められる. これらの LWK に対する操作が, LWK 上で動作するアプリケーションに対する外乱となることを防止する必要がある.

4. 設計

本章では, 前章で挙げたそれぞれの課題に対する設計を述べる.

4.1 CPU, メモリの排他利用

2.3.3 節に記載したように, Linux オンリーモードのバッチジョブ運用でも, ジョブ毎の資源排他利用を実施している. LWK に対しても, 他のジョブと排他的に確保した資源をそのまま LWK 向け資源として割り当てることで, 排他的な割り当てが可能である. そのため, LWK をジョブの枠組みで扱い, LWK 向け資源割当処理を追加することで, CPU, メモリの占有割当を実施できる.

4.2 NUMA を意識した CPU, メモリ割り当て

2.3.3 節に記載したように, Linux オンリーモードのバッチジョブ運用でも, ジョブの NUMA ポリシを反映した資源割当を実施している. そのため, 4.1 節に記載した CPU, メモリの排他割り当てと同様に, LWK をジョブの枠組みで扱い, LWK 向け資源割当処理を追加することで, NUMA ポリシを意識した資源割当が実施できる.

4.3 物理連続メモリの割り当て

Linux オンリーモードではメモリの絶対量の管理と NUMA ポリシの反映を行っているが, 連続メモリを意識した資源管理は実施しておらず, 断片化を抑止できない.

そこで, Linux のメモリ管理が NUMA 単位で行われ, メモリ獲得は距離の近い NUMA から行われることに着目し, ソフトウェアで NUMA 構成情報を変更してジョブに割り当てるメモリを独立した仮想的な NUMA ノードとして分割する対策を実施する.

図 3 のように, 物理 NUMA 構成をジョブ用とシステム用の 2 つに分割する.

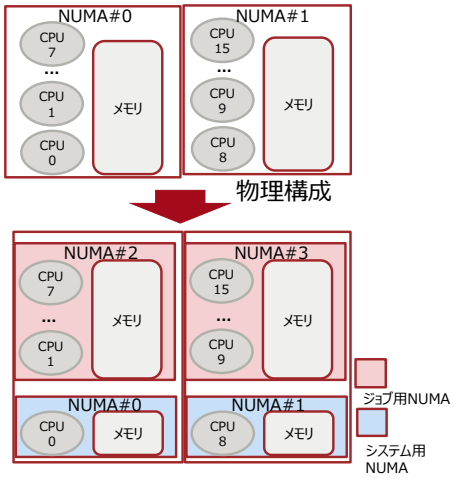

図 3 NUMA 構成の分割の概要

システム用 NUMA に, システムデーモン, OS の割り込み処理をバインドする. これにより, システムプロセスのメモリはシステム用 NUMA から割り当てられる. これにより, ジョブ用 NUMA はジョブに専有させる. これによりジョブ用 NUMA のメモリをクリーンな状態に保ち, メモリ断片化を防止することが可能となる.

4.4 デリゲーション処理に対する資源使用制限

デリゲーション処理は LWK 毎に仕組みは異なるが, Linux 上では LWK 専用の特別な仕組みではなく, Linux ネイティブのプロセスやカーネルスレッドの仕組みを使って実現されている<脚注 i>. そのため, Linux オンリーモードと同様に Linux 標準の資源管理機能 cgroups[16]を利用してメモリ, CPU の資源使用量制限を実施可能である.

まずノード内に存在するコアを, システム処理用に使うコア(OS コア)と, ジョブ実行で占有利用するコア(App コア)の 2 つに分けて管理する. システムデーモン, デバイス割り込みの受け付けなどのシステム処理に加え, デリゲーション処理も OS コアに動作を制限することで, ノード上で並列動作する他ジョブの専有する CPU の使用を防止する.

加えて, OS コア上で複数のデリゲーション処理が動作する場合にも, システムプロセスや, OS コア上で動作する Linux オンリーモードのジョブプロセスの動作を妨害することを防ぐため, 単位時間当たりに使用可能な CPU 使用時間を制限する. これにより, 多数のデリゲーション処理が高負荷状態で動作する場合にも OS コアを専有しなくなる.

i) mOS の場合は Linux に LWK を埋め込み, mOS の動作コアで直接 Linux のコードが実行されるが, この処理は mOS 側の資源を使って動作するため資源制限の必要はない.

4.5 LWK の外乱とならない LWK のジョブ連動制御

運用ソフトは LWK モードのジョブ実行中にジョブ状態監視とジョブ統計情報収集を実施する必要がある. LWK 上のプログラムへの性能影響を配慮すると, これらの処理を LWK 内にデーモンやカーネルスレッドなどで実施する方式は取れない. そこで, LWK 自体の資源管理機能を拡張し, LWK の制御インターフェースの一部として外から状態監視と統計情報収集を実施な機能を追加する方式をとる. 運用ソフトからこれらの制御インターフェースの呼び出しのみを実施し, LWK 内部への直接的な操作は行わないことで, 外乱を防止する.

5. 試作

本章では, 前章の 4.3, 4.4, 4.5 節で述べた 3 つの課題に対する設計の試作結果を述べる. 本試作では LWK として McKernel を採用し, x86_64 アーキテクチャ上で実装を行った.

(1) 物理連続メモリの割り当て

- 物理 NUMA 構成をソフトウェア的に分割するため, Linux カーネルの CONFIG_NUMA_EMULATION を有効化し, Linux カーネルの NUMA 構成情報をカーネルの起動パラメタで変更可能とした. この方法では各 NUMA ノードが均等のメモリ量を持つ NUMA 構成となるため, ひとつの NUMA ノードのメモリサイズはシステム用に足りる程度にとどめ, ジョブ用に必要な数の NUMA ノードを割り当てることとした.
- システムプロセスと OS の割り込み処理の CPU アフィニティを変更し, システム用 NUMA にバインドした.
- Linux カーネルを修正し, 起動後に sysfs インターフェースからメモリの属性を変更可能とした. このインターフェースを用いて, ジョブ用 NUMA に属するメモリを MOVABLE 属性に設定した. Linux は, メモリマイグレーションできないカーネルが使用するメモリは MOVABLE 属性の領域からの割り当てを行わない. そのため, カーネル処理によるジョブ用 NUMA のメモリ断片化を防止できる.

(2) デリゲーション処理に対する資源使用制限

LWK モードのジョブ起動時に, OS コア上に cgroup を作成し, McKernel に対応する全ての mcexec を 1 つの cgroup で管理した.

- cpuset サブシステムの cpuset.cpus に OS コアの CPU 番号を登録し, Linux 上で動作する mcexec の動作コアを OS コアのみに制限した.
- memory サブシステムの memory.max_usage_in_bytes にメモリ制限値を設定し, Linux 上で動作する mcexec の利用可能なメモリ量を制限した. 今回の試作では制限値を 128MB とした.
- cpu サブシステムの cpu.cfs_period_us と cpu.cfs_quota_us を設定し, 単位時間当たりに使用可能な CPU 時間を制限した. 今回の試作では OS コアを 2 つと想定し, その半分(CPU 使用率 100%)までに制限した. この設定により, mcexec が複数起動しても, OS コアを使い切ることがなくなる.

(3) LWK の外乱とならない LWK のジョブ連動制御

McKernel の資源管理モジュールである IHK に, 状態監視のための McKernel の状態取得（表 1 の 1,2,7）と統計情報取得用（表 1 の 3～6,8）のインターフェースを追加し, 運用ソフトからライブラリ経由で取得可能とした. IHK ではこれらの情報を取得する際, McKernel に対する割り込み処理を行わず, McKernel と共有する特定のメモリ領域を読むことで実現しており, McKernel 上のプログラムに対する影響を防止している.

表 1 追加インターフェース一覧

項番	機能	インターフェース名
1	OS indexの一覧取得	ihk_getoslist()
2	OSの状態取得	ihk_getosinfo()
3	統計情報取得	ihk_getrusage()
4	CPU PA採取イベント登録	ihk_setperfevent
5	CPU PA収集開始・停止・イベント削除	ihk_perfctl()
6	CPU PA情報取得	ihk_getperfevent()
7	監視eventfd取得	ihk_geteventfd()
8	資源情報取得	ihk_getihkinfo()

6. 評価

6.1 評価環境

以下環境で試作の評価を実施した.

- Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- McKernel v1.4.0
- CentOS 7.3(linux カーネルは修正)
- 16 論理コア, 12GB メモリ搭載
- OS コアは 2 コア(CPU0,CPU1),
  App コアは 14 コア(CPU2～CPU15)
- ジョブ用メモリを 10GB, システム用メモリを 2GB

図 4 に評価環境の NUMA 構成を示す.

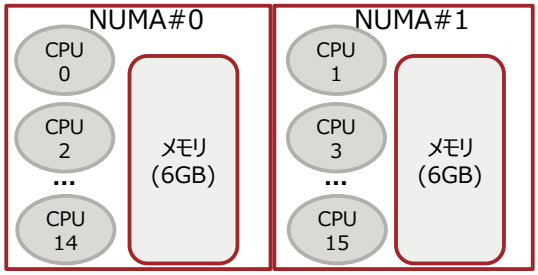

図 4 評価環境概要

6.2 物理連続メモリの割り当て

6.2.1 評価方法

12GB のメモリを, 1GB メモリを持つ NUMA ノード 12 個（#0～#11）に分割し, #0,#1 をシステム用（2GB）, #2～#11 をジョブ用（10GB）として割り当てた.

次の断片化検証テストを 12 時間実行し，終了時のメモリ情報を確認することで，システムの断片化に対する耐性を評価した．

- システムプロセスとして定常的にメモリ獲得開放を繰り返すプログラムと，ファイルキャッシュの獲得開放を繰り返すプログラムを実行し，システムメモリ 95%まで消費する．
- 同一ノード上で次の(i)，(ii)のジョブを繰り返し実行する．
  (i) Linux オンリーモードで，STREAM[17],姫野ベンチマーク[18]，IOzone[19]を並列で実行し，ジョブメモリの 95%以上を消費する．STREAM と姫野ベンチマークは，ファイルキャッシュをほとんど使用せず，開始時に大きなメモリ獲得を実施し終了時に開放する動作を行う．これによりジョブメモリの使用率が高い状態を作り出し，IOzone がファイルキャッシュを獲得し小さなメモリ獲得を繰り返すことで，断片化が発生しやすい状況を作り出す．
  (ii) LWK モードのジョブを実行し，ジョブメモリの 90%以上を McKernel に割り当てる．
- プログラム実行後以下 2 つの情報を確認する．
  - McKernel に割り当たった物理連続メモリの一覧
  - Linux 上の 2MB ラージページの枚数

6.2.2 結果

断片化検証テスト実施後の McKernel に割り当て可能な連続メモリ領域の割合について，対策実施前と実施後の結果をそれぞれ図 5，図 6 に示す．対策を実施した結果，McKernel に割当可能な 256MB 以上の連続メモリ領域の割合が，57%から 88%に改善した．

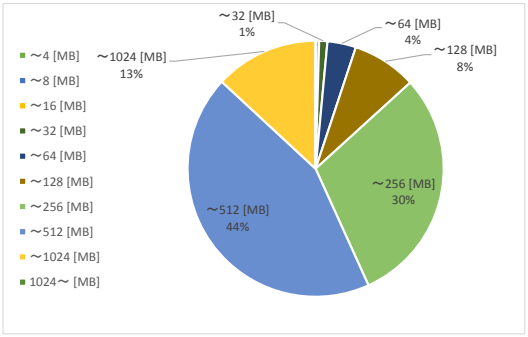

図 5 対策実施前の連続メモリ領域の割合

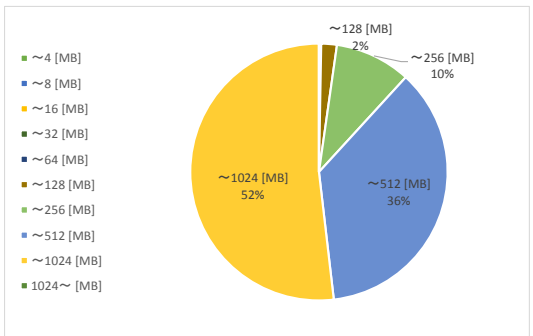

図 6 対策実施後の連続メモリ領域の割合

図 7 に連続メモリ領域の数で比較した結果を示す．対策後では対策前よりもサイズの大きな領域の数が増加し，断片化が改善されていることが分かった．

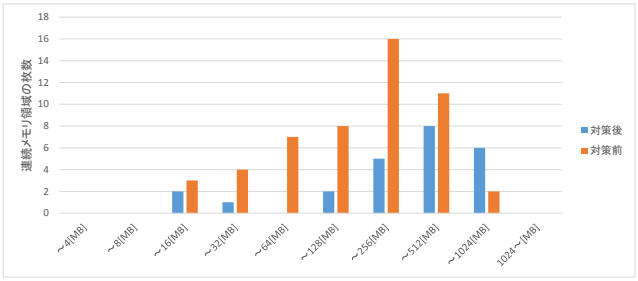

図 7 McKernel に割り当たる連続メモリ領域の枚数

図 8 で Linux 上の 2MB ラージページの枚数を比較した．対策実施前は 2MB ページが 428 枚減少したのに対し，実施後は 32 枚の減少に留まっており，ジョブ用 NUMA での断片化を改善したことを確認した．

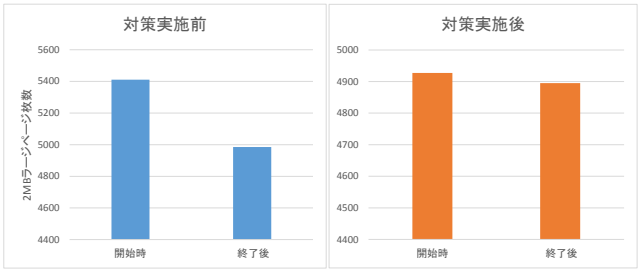

図 8 2MB ラージページの枚数

6.2.3 考察

試作内容がメモリ断片化の抑止に効果があることが確認できたが，断片化を完全に防止することはできていない．

この原因としては，本試作では Linux カーネル起動後に sysfs インターフェースからメモリの属性を変更したが，この方法だとカーネル起動時のメモリ獲得の影響で NUMA ノード毎に必ず 1 メモリブロック(x86_64 では 128MB)は MOVABLE 属性に設定できないことが考えられる．今回の評価ではジョブ用に 10NUMA ノードを割り当てたため，1.28GB 分のメモリが断片化のリスクがある状況であった．

コア単位スレッドなど一部のカーネルスレッドは CPU アフィニティ設定によるシステム NUMA へのバインドが不可能なものが存在し，ジョブ用 NUMA からメモリを獲得する可能性があり，これらが MOVABLE 属性に設定できないメモリを使用し，断片化が進んだと推測される．

今後の改善として，ACPI テーブルの情報を変更してカーネル起動時にメモリを MOVABLE 属性に設定することにより，断片化の更なる解消を検討する．加えて，MOVABLE 属性を活かし，McKernel 起動前にメモリを一旦オフラインとして断片化を解消する対策を検討する．

また，本体策による断片化抑止の直接的な性能向上を確認するため，連続メモリ領域が増加した場合の McKernel 上で性能検証に取り組んでいく．

6.3 デリゲーション処理に対する資源使用制限

6.3.1 評価方法

図 9 のように，Linux オンリーモードのジョブと LWK モードのジョブが 1 ノード上で並列に動作し，かつ OS コアを Linux オンリーモードのジョブと mcexec，システムデーモンが共同利用する状況で，Linux オンリーモードのジョブとして実行した UnixBench ベンチマーク[20]の性能変化を確認した．

McKernel 上のプロセスとしては，mcexec に負荷が掛かるように，ramfs 上への read(2)を McKernel に割り当てたコア数分(7 プロセス)並列で実施した．

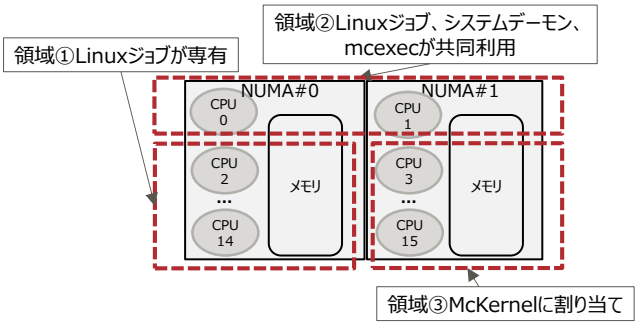

図 9　評価環境の概要

6.3.2 結果

表 2 にベンチマークの実行結果を示す．1 ノードで Linux オンリーモードのジョブと LWK モードのジョブが混在した場合にも，ベンチマークスコアの比が 5%程度の誤差範囲に収まっており，mcexec の動作による Linux オンリーモードのジョブへの性能影響がないことを確認できた．

表 2　Linux 上での UnixBench スコア

ベンチマーク項目	領域③でのMcKernel実行		性能比%
	a. 有り	b. 無し	(a./b.)
Execl Throughput (loop per second)	3150.8	3324.1	94.8
System Call Overhead	5060.2	4993.6	101.3
System Benchmarks Index Score	3791.1	3741.7	101.3

6.3.3 考察

今回は Linux オンリーモードのジョブが OS コアまで使用し，mcexec の処理に影響されるケースを評価したが，影響は見られなかった．mcexec に対する cgroup による CPU 帯域制限が有効に働くことが分かった．

今後の課題としては，測定パターンを増やし，LWK モードのジョブで通信のオフロード処理を実施する場合など，さらに OS コアへの負荷が想定されるケースでも，cgroups による公平な OS コアの帯域制御が可能かを確認していく．

6.4 LWK の外乱とならない LWK のジョブ連動制御

6.4.1 評価方法

McKernel を単体で起動した場合と，運用ソフト経由でジョブとして起動した場合で，ベンチマーク実行性能を比較した．運用ソフトが実施するジョブ統計情報収集処理が McKernel の外乱とならないかを評価した．ジョブ統計情報の収集処理の間隔は，通常の運用では 10 分間隔だが，影響をみるため負荷を増加させ 1 秒間隔とした．

結果

表 3 に McKernel 上でのベンチマークスコアを示す．運用ソフト利用時の性能低下は 0.6%以下と無視できる程度であった．

表 3　McKernel 上でのベンチマークスコア

ベンチマーク項目		①単体実行		②運用ソフト経由		性能比
		スコア	標準偏差 (3回測定)	スコア	標準偏差 (3回測定)	%(②/①)
Himeno bench 98 (MFLOPS)[2thread]		3594.99	0.91	3598.19	0.49	100.1
STREAM (MB/s) [8thread, 1NUMA]	Copy	9134.5	9.5	9082.1	9.9	99.4
	Scale	9374.5	4.9	9360.4	3.8	99.8
	Add	10129.5	26	10146	5.9	100.1
	Triad	10353.9	20.1	10345.1	4.4	100

6.4.2 考察

期待通りジョブ運用する際のジョブ統計情報収集処理が LWK 上のプログラムに対する外乱とならないことを確認できた．

今後の課題としては，複数の McKernel インスタンスを並列でジョブ実行し，1 つの McKernel 上でプログラム実行中に別の McKernel の起動終了が行われ，IHK に負荷がかかる場合の性能影響を確認し，McKernel インスタンス数が増加した場合にも運用可能かを検証していく．

7. まとめ

本論文では，バッチジョブ運用で LWK を利用するための運用ソフトの設計と試作評価について述べた．

実現すべき運用ソフトの要件として，

（1）各実行環境で性能を引き出す資源割り当て
（2）実行環境間の干渉による性能影響の防止
（3）運用ソフトの導入伴う性能影響の防止

を実現する運用ソフトについて設計し，LWK として McKernel を取り上げて次の 3 つの課題に対する試作評価を実施した．

(1) 物理連続メモリの割り当て

Linux の NUMA 情報を変更し，McKernel に割り当てるメモリをジョブが専有する独立した NUMA ノードとして管理することで，断片化を改善し，McKernel に割り当てる連続メモリ量を増やすことができた．

今後の課題として，カーネル起動時にメモリを MOVABLE 属性に設定することにより，断片化の更なる解

消を検討する．および，連続メモリ領域を利用する McKernel 上でのアプリケーション性能評価を行っていく．

(2) デリゲーション処理に対する資源使用制限

McKernel のデリゲーション処理プロセスである mcexec を Linux 上の cgroups で管理することにより，Linux 上の処理に対する性能影響を防止できることを確認した．

今後の課題として，通信のオフロードを実施する場合など McKernel の利用用途に応じた測定パターンを増やし，安定したジョブ運用が可能であることを検証していく．

(3) LWK の外乱とならない LWK のジョブ連動制御

ジョブ実行中に運用ソフトが行う状態監視，統計情報収集処理を McKernel 外部から取得可能な機能を IHK-McKernel に追加することで，McKernel 内部のプログラム実行に影響しない形で，Linux オンリーモードのジョブと同等のジョブ運用が可能であることを確認した．

今後の課題として，McKernel のインスタンス数が増加した場合など想定する運用範囲を広げて検証を行っていく．

また，本論文では，LWK に焦点を当てて設計を行ったが，同様の仕組みで Linux コンテナや仮想マシンを含めた統合的な仮想環境のバッチジョブ運用の検討を進めていく．

謝辞 本論文の一部は，文部科学省「特定先端大型研究施設運営費等補助金（次世代超高速電子計算機システムの開発・整備等 ）」で実施された内容に基づくものである．

参考文献

[1] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. "A case for high performance computing with virtual machines." In Proceedings of the 20th annual international conference on Supercomputing (ICS '06). ACM, New York, NY, USA, 2006. pp. 125-134.
[2] A. Reuther, P. Michaleas, A. Prout and J. Kepner, "HPC-VMs: Virtual machines in high performance computing systems," 2012 IEEE Conference on High Performance Extreme Computing, Waltham, MA, 2012, pp. 1-6.
[3] S. Varrette, M. Guzek, V. Plugaru, X. Besseron and P. Bouvry, "HPC Performance and Energy-Efficiency of Xen, KVM and VMware Hypervisors," 2013 25th International Symposium on Computer Architecture and High Performance Computing, Porto de Galinhas, 2013, pp. 89-96.
[4] M. T. Chung, N. Quang-Hung, M. T. Nguyen and N. Thoai, "Using Docker in high performance computing applications," 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), Ha Long, 2016, pp. 52-57
[5] Douglas M. Jacobsen, Richard Shane Canon. "Contain This, Unleashing Docker for HPC.", Cray User Group 2015, 2015.
[6] Kurtzer, Gregory M., Vanessa Sochat, and Michael W. Bauer. “Singularity: Scientific Containers for Mobility of Compute.” Ed. Attila Gursoy. PLoS ONE 12.5 (2017): e0177459. PMC. Web. 24 Nov. 2017.
[7] Balazs Gerofi, Yutaka Ishikawa, Rolf Riesen, Robert W. Wisniewski, Yoonho Park, and Bryan Rosenburg. "A Multi-Kernel Survey for High-Performance Computing." In Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16). ACM, New York, NY, USA, 2016. pp. 5-8.
[8] ポスト「京」プロジェクト, http://www.aics.riken.jp/fs2020p
[9] McKernel, http://www-sys-aics.riken.jp/ResearchTopics/os/mckernel.html
[10] Masamichi Takagi, Balazs, Gerofi, Norio Yamaguchi, Takahiro Ogura, Toyohisa Kameyama, Atsushi Hori, Yutaka Ishikawa. Operating System Design for Next Generation Many-core based Supercomputer. IPSJ SIG Notes 2015-OS-133, 2015.
[11] 平井浩一, 小田和友仁, 岡本高幸, 二宮温, 住元真司, 高木将通, Balazs Gerofi, 山口訓央, 小倉崇浩, 亀山豊久, 堀敦史, 石川裕. "HPC 向けメニーコア OS を用いたバッチジョブ運用の課題検討." 情報処理学会研究会報告 2015-OS-133(2) , May. 2015.
[12] FusedOS, https://github.com/ibm-research/fusedos
[13] mOS, https://github.com/intel/mOS/wiki
[14] FFMK, https://ffmk.tudos.org/
[15] Hobbes, http://xstack.sandia.gov/hobbes/index.html
[16] Documentation of cgroups on kernel.org, https://www.kernel.org/doc/Documentation/cgroup-v1/
[17] STREAM, https://www.cs.virginia.edu/stream/
[18] 姫野ベンチマーク, http://accc.riken.jp/supercom/himenobmt/
[19] IOzone Filesystem Benchmark, http://www.iozone.org/
[20] UnixBench, https://github.com/kdlucas/byte-unixbench