Vol.2012-OS-122 No.6
2012/8/1

マルチコア・メニーコア混在型計算機における軽量OS向けI/Oライブラリの提案

深沢 豪[1,a)]　長嶺 精彦[1]　坂本 龍一[1]　佐藤 未来子[1,5]　吉永 一美[2,5]　辻田 祐一[2,5]　堀 敦史[3,5]
石川 裕[4,3]　並木 美太郎[1,5]

概要：マルチコア・メニーコア混在型計算機では，並列演算をメニーコアで，資源管理等をマルチコアでそれぞれ分担処理することにより，メニーコアの高い並列演算性能を生かす．本論文では，メニーコア上の演算 OS で発生した I/O アクセスをマルチコア上の管理 OS が協調して処理するための，I/O ライブラリおよび I/O 代行機構を提案している．演算 OS の動作による擾乱を極力削減するための I/O 代行方式と，管理 OS による低遅延な I/O アクセスにより，演算 OS の軽量性を損ねることなく高い I/O 性能を実現する．本 I/O 代行方式では，管理 OS からメニーコアのメモリへ直接 I/O アクセスを実施するとともに，管理 OS における I/O 代行処理を並列化することで，演算 OS 側での待ち時間を削減する．2 個のマルチコア CPU を用いて，本論文で提案した I/O ライブラリおよび I/O 代行機構を試作し，ファイル I/O の所要時間を評価した結果，I/O アクセスサイズを 16MB 程度とすることで，Linux 単体でのファイル I/O と同等の帯域幅を確認した．今後は，同時 I/O 要求に対する性能改善を行なっていく．

A design of I/O Library on the Light-weight OS for a Multi/Many-core Parallel Computer

Abstract: The Multi/Many-core Parallel Computer has the structure of a hybrid computer architecture with multi-core and many-core processors for Exa-scale computing. In this computer, resources at a lightweight operating system (LWOS) on a many-core processor are managed by a host operating system (Host OS) on a multi-core processor. This paper describes the design of the I/O library and the I/O processing unit to delegate I/O environment such as filesystems and device drivers on the LWOS to the Host OS. In this study, the I/O processing unit on the Host OS performs a direct I/O access to the many-core memory and introduce a parallel processing for a I/O access to reduce the I/O wait time. In the reults, the evaluation is shown that a bandwidth of the 16MB read access is equal to the Linux I/O access. For the future, we will continue to improve the multiple I/O access performance.

1. はじめに

大規模シミュレーション等，高い演算性能が要求される用途に用いられるスーパーコンピュータは，現在，ペタフロップス級の演算性能を有する [1]．高い演算性能を有するスーパーコンピュータを実現する方法として，数値演算を高速に処理可能な GPGPU（General Purpose GPU）を，汎用的な CPU のアクセラレータとして用いる手法がある．一方，汎用的な CPU コアを多数集積したメニーコア CPU を用いて，高演算性能を実現する方式が広く採用されている．

メニーコア CPU には，Intel 社の SCC（Single-chip Cloud Computer）のように，メニーコア CPU 単体でシステムを構成可能なものと，Intel 社の MIC（Many Integrated Core）[2] のように，メニーコア CPU を既存のマルチコア CPU のアクセラレータとして用いるシステム構成がある．

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[2] 近畿大学
Kinki University
[3] 理化学研究所計算科学研究機構
RIKEN AICS
[4] 東京大学
University of Tokyo
[5] 独立行政法人科学技術振興機構 CREST
JST CREST
[a)] fzawa@namikilab.tuat.ac.jp

メニーコア CPU をアクセラレータとして用いる計算機構成は，GPU をアクセラレータとして用いる計算機構成に比べ，プログラミングが容易であるという利点があり，今後，HPC での利用が活発化すると予想される．

メニーコア CPU をマルチコア CPU のアクセラレータとして用いる，ヘテロジニアス構成の計算機では，システム内にそれぞれ特性が異なる 2 種類の CPU が存在することになる．メニーコア CPU はコア単体での演算性能がマルチコア CPU よりも低く，キャッシュメモリの容量もコアあたり数百 KB と少ない．マルチコア CPU はメニーコア CPU とは逆にコア単体での性能が高いが，コア数が少ないために並列演算性能で劣る．この計算機上で Linux 等の汎用 OS を用いた場合，メニーコア CPU の特性を生かせず，満足な性能を得られない可能性が高い．

本研究では，メニーコア CPU の並列演算性能を生かすために，メニーコア CPU とマルチコア CPU で，それぞれ別個の OS を動作させる．メニーコア CPU 上では，汎用 OS の備えるファイルシステムやディスク I/O 機能等を省き，並列演算アプリケーションの実行に特化した軽量な OS を用いる．一方，マルチコア CPU 上では，メニーコア OS 上の軽量 OS から削減した資源管理機能等を構成するため，汎用的な OS を用いる．これら 2 種類の OS 間で，不足する機能を互いに提供し合う OS 間連携機構を導入する．一つのタスクを，並列演算処理はメニーコア CPU 上の軽量な OS で，ファイル I/O 等の資源管理はマルチコア CPU 上の汎用的な OS で，それぞれが分担して処理することで，両 CPU の特性を生かした，高い演算性能を有する計算機環境を構築することができる．

これまで，メニーコア上 OS の処理軽減のために，演算コア側のプロセス管理やメモリ管理，I/O 管理の一部機能をマルチコア上 OS で代行する方式について提案した [4]．本論文では，メニーコア上のアプリケーションから，資源管理の代行方式を利用するための I/O ライブラリを提案する．マルチコア・メニーコア混在型計算機の模擬環境上で本 I/O ライブラリを試作し，I/O アクセス性能について評価を行う．

2. 課題と目的

Linux に代表される汎用 OS では，アプリケーションが I/O アクセスを要求すると，OS 内のファイルシステムおよびデバイスドライバにより，I/O デバイスへデータの読み書きが行われる．本研究の計算機環境の場合，メニーコア上のアプリケーションが要求した I/O アクセスは，メニーコア上 OS と，マルチコア上 OS で分担処理される．汎用 OS 向けアプリケーションに対する互換性を実現するには，本計算機特有の I/O アクセス方式を，汎用 OS で広く普及している POSIX 互換 API 等を用いて隠蔽する必要がある．

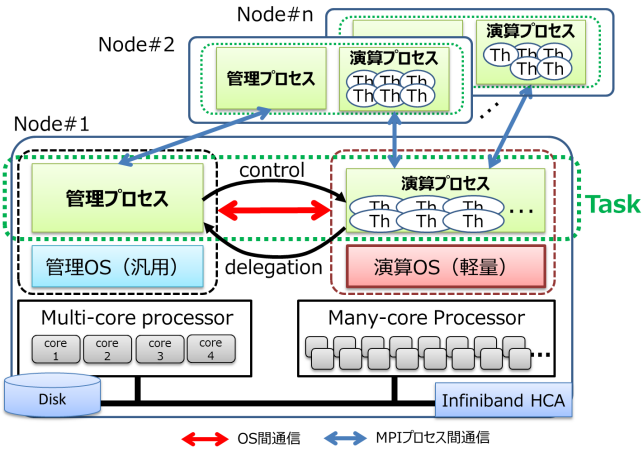

図 1 マルチコア・メニーコア混在型計算機におけるシステム構成
Fig. 1 The system architecture for a Multi/Many-core Parallel Computer.

本計算機では，メニーコア上 OS における I/O 機能を可能な限りマルチコア上 OS へ移管することで，メニーコア上 OS の軽量化を行う．本方針は，I/O 機能を移管した結果，I/O アクセス性能が大幅に低下した場合には意味を成さない．このため，移管にともなうオーバーヘッドを可能な限り削減する必要がある．また，数十個から 100 個程度存在するメニーコアの各コアが，同時に I/O アクセスを要求した場合に備え，マルチコア上での I/O 処理および OS 間での通信経路を並列化する必要がある．

本論文では，メニーコア上 OS における擾乱を防止するため，マルチコア上 OS への I/O アクセス委託機能のみを備えた軽量な I/O ライブラリと，本 I/O ライブラリを実現するためのマルチコア上 OS 向けファイル I/O 機構を提案し，メニーコア上 OS が提供するファイル I/O やノード間 I/O 機能のうち，ファイル I/O 機能をマルチコア上 OS へ移管することを目指す．本ライブラリにより，既存アプリケーションに対するソースコードレベルでの互換性を実現するとともに，マルチコア上 OS によるファイル I/O 処理の工夫による，低遅延なファイル I/O 機能をメニーコア側へ提供する．これにより，メニーコア CPU 上の OS からファイルシステムや I/O デバイス管理を省いたシステム構成における課題を解決し，高い並列演算性能を有するマルチコア・メニーコア混在型計算機の実現に寄与する．

3. システム構成

3.1 ハードウェア構成

本研究で想定するマルチコア・メニーコア混在型計算機は，Intel 社が開発中の MIC（Many Integrated Core）[2] のように，従来の汎用 CPU コアを多数集積したホモジニアス構成のメニーコア CPU に，既存のマルチコア CPU を組み合わせた，ヘテロジニアス構成のハードウェアを備える．本計算機のメニーコア CPU は，コア単体での性能は旧世代の CPU と同等であるが，コア数を数十個の単位で

多数集積する他，コア間でのキャッシュコヒーレンシを保てるため，アプリケーションの並列化を容易に行える利点がある．

メニーコア CPU とマルチコア CPU は，それぞれ独自のメモリとメモリ空間を有するが，メニーコアのメモリ領域の一部を，マルチコアのメモリ空間へ，マルチコアのメモリ領域の一部を，メニーコアのメモリ空間へ，それぞれマッピングし，共有メモリとして利用することができる．CPU 間でのデータ転送時に共有メモリへ直接データを書き込むことで，無駄なデータコピーを削減した低遅延な通信を実現できる．また，CPU 間割り込み（IPI）を合わせて利用することでメモリのポーリングを排除し，CPU 資源を節減することができる．

本計算機でクラスタを構成した際のノード間通信基盤として，低遅延かつ大容量な通信を特徴とした Infiniband を用いる．マルチコアとメニーコアを接続する内部バス上にアダプタ（HCA）を配置することで，メニーコアのメモリへの直接アクセスを可能とする．

3.2 システムソフトウェアの構成

3.2.1 OS 構成

本研究では，メニーコア CPU の豊富なコア資源を有効に活用することで，高効率な並列演算アプリケーションの実行環境を実現する．しかし，メニーコアのコア単体性能が低いため，メニーコア上では並列演算処理に特化した軽量 OS を用い，マルチコア上の汎用 OS で軽量 OS の動作を支援する方針としている．そこで，本研究では**図 1** に示すように，メニーコア上で軽量なスレッド実行基盤とカーネルを特徴とした軽量 OS ”Future/MULiTh” を「**演算 OS**」として用いる．一方，マルチコア上では汎用 OS の Linux を「**管理 OS**」として用いる．

3.2.2 管理 OS による演算コア側資源管理の代行

演算 OS で行う資源管理には，プロセス・スレッド管理，CPU コア管理，メモリ管理，I/O 管理があるが，軽量実行基盤を利用可能なスレッド管理を除き，可能な限り管理 OS が資源管理を代行することで，演算 OS による擾乱で並列演算アプリケーションの性能が低下する事態を防いでいる．管理 OS による資源管理の代行内容は次のとおりである．

- プロセスの生成・管理
- プロセスへの CPU コア割り当て
- プロセスへの物理/仮想メモリ割り当て
- スレッドで発生した I/O アクセスの処理

資源管理の代行により，並列演算アプリケーションが待ち状態になる事態を防ぐため，プロセス内でのスレッド管理やヒープメモリ拡張等の資源管理は，演算 OS 内で処理を完結させるとともに，新規のプロセスに対して，動作に必要となる量の物理メモリを予め割り当てることで，ページフォルトの発生を抑止する．

3.2.3 プロセスモデル

演算 OS 上のプロセス（以下，**演算プロセス**）と，管理 OS 上のプロセス (以下，**管理プロセス**) は，それぞれ異なる役割を持つ．演算プロセスは，メニーコア側資源の割り当て単位として用意し，プロセス内で多数の軽量なスレッド（以下，**演算スレッド**，図 1 における “Th”）を並列に実行することで，メニーコアプロセッサの並列演算性能を追求する．一方，管理プロセスは演算プロセスの管理を行うために用意され，演算プロセスの生成・実行状態の制御や，演算プロセスに対するメニーコア側資源の割り当てを行う．演算プロセスと管理プロセスが相互に連携し，一つのアプリケーション実行単位として「Task」を形成する．Task は，クラスタの管理ノードにより生成される．具体的には，クラスタ管理ノードが管理プロセスを生成し，管理プロセスが演算プロセスを生成する．

4. ファイル I/O の代行方式

4.1 ファイル I/O の分担方針

Linux に代表される汎用 OS では，I/O デバイスに対するファイル I/O をアプリケーションへ提供するために，標準入出力ライブラリ，ファイルシステム，I/O デバイスドライバを OS 内に備えている．メニーコア上の演算 OS でファイル I/O 機能を実現するにあたり，これら全ての機能を演算 OS 内に構成した場合，演算 OS の軽量性を損ねてしまい，メニーコア CPU の並列演算性能を生かすことができない．このため，ファイル I/O を実現するための機能を可能な限り管理 OS へ移管することで，演算 OS でファイル I/O を実現しつつ，OS の軽量性を維持する方針とする．

本研究では，標準入出力ライブラリ，ファイルシステム，I/O デバイスドライバの 3 機能のうち，標準入出力ライブラリのみを演算 OS 上に構成し，ファイルシステムと I/O デバイスドライバ機能を管理 OS へ移管する．ファイルシステムは，I/O デバイスの抽象化のほか，I/O アクセス性能向上や高い信頼性を実現するため，数多くの処理を内部で行なっている．また，I/O デバイスドライバでは，OS とデバイスの間で通信を行うために割り込みを利用することが多い．このため，ファイルシステムと I/O デバイスドライバを管理 OS へ移管することで，メニーコアの CPU 時間節減や，キャッシュメモリの撹乱防止を実現することができる．一方，標準入出力ライブラリは管理 OS へ移管せずに演算 OS 上で構成することで，バッファリングによる I/O アクセス性能向上を演算 OS 上で実現する．

4.2 演算 OS におけるファイル I/O 機構

本計算機では，標準入出力ライブラリを演算 OS 上で，ファイルシステムと I/O デバイスドライバを管理 OS 上で構成する．この方針を実現するため，本研究では演算 OS

表 1　標準入出力ライブラリによる主なファイル I/O API
Table 1　The file I/O API of the standard I/O library.

分類	関数名
ファイルのオープン・クローズ	fopen, fclose
ファイルへの I/O アクセス	fseek, fread, fwrite, fgetc, fgets, fputc, fputs
バッファリングの制御	fflush, setvbuf

表 2　POSXI.1 互換のファイル I/O API
Table 2　The POSIX.1 file I/O API.

分類	関数名
ファイルのオープン・クローズ	open, close
ファイルへの I/O アクセス	lseek, read, write
キャッシュのフラッシュ	fsync, fdatasync

上に，標準入出力ライブラリ機能と管理 OS へのファイル I/O 委託機能を備えた I/O ライブラリを構築する．Linux に代表される汎用 OS では，ファイル I/O 機能がカーネル空間上で実装されているため，ユーザ空間上で動作するアプリケーションからのファイル I/O には，システムコールによる特権レベルの切り替えが必要である．ハードウェアへのアクセスを行うため特権レベルが必要となるファイルシステムおよびデバイスドライバを管理 OS 上で構成する本計算機の特徴を生かし，本ライブラリでは，演算 OS におけるファイル I/O 処理をユーザ空間で完結させる．これにより，特権レベル切り替えによるオーバーヘッドや，ユーザ空間とカーネル空間の間で行うデータコピーを排除した，低遅延なファイル I/O を実現する．

本ライブラリでは，標準入出力ライブラリによるファイル I/O API とともに，POSIX.1（Portable Operating System Interface）[6] で定められたファイル I/O API をアプリケーションへ提供する．UNIX 系 OS で広く普及している 2 種類のファイル I/O API をサポートすることで，既存の UNIX 系 OS 向けのアプリーケーションに対するソースコードレベルでの互換性を実現する．本ライブラリにおいてサポートする標準入出力ライブラリによる主な API を**表 1** に，POSIX.1 互換の API を**表 2** に示す．現状では，POSIX.1 互換 API はファイルへの I/O アクセスを行うために必要最低限な関数のみをサポートしているため，ディレクトリ操作等は管理 OS 側で行う必要がある．今後，対応関数を増やし，完全な POSIX.1 互換 API を実現する方針である．

POSIX.1 互換 API を経由して要求されたファイル I/O では，管理 OS へファイル I/O を委託するために必要となるコンテキストの送信処理と，I/O アクセス結果の受信処理のみで，ライブラリ内での処理が終了するため，キャッシュメモリの撹乱を必要最低限に抑えることができる．このため，演算性能向上を目指したアプリーケーションでは，標準入出力ライブラリによる API ではなく，POSIX.1 互換 API を用いることで，メニーコア CPU の少ないキャッシュメモリを節減することができる．

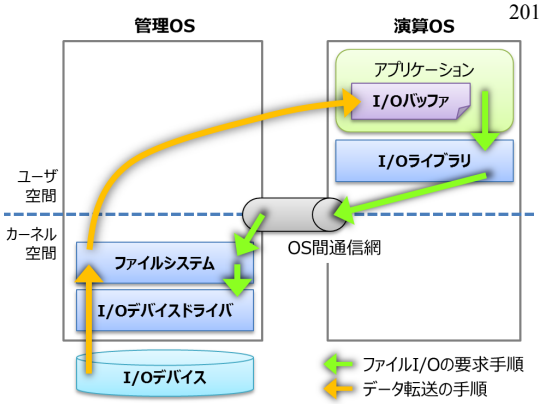

図 2　ファイル I/O の代行手順
Fig. 2　The process for remote file I/O.

4.3　管理 OS におけるファイル I/O 機構

本計算機では，演算 OS の軽量性を向上させるため，演算 OS で発生したファイル I/O を処理するファイルシステムおよび I/O デバイスドライバを管理 OS 上で構成する．ファイルシステムおよび I/O デバイスドライバはハードウェアへのアクセスを伴うため，カーネル空間へ配置するが，OS 間通信による I/O アクセス要求の受信から，ファイルシステム呼び出しまでの処理をカーネル空間内で完結させることで，特権レベル切り替えによるオーバーヘッドを排除する．また，管理 OS 上のファイルシステムから演算 OS 上のメモリに対して，直接 I/O アクセスを実施することで，OS 間での余分なデータコピーを排除する．以上の工夫により，演算 OS 上で発生した I/O アクセスを可能な限り低遅延に処理し，演算 OS 上アプリケーションの待ち時間を削減する．

また，本計算機では，メニーコアの並列演算性能を生かすため，演算 OS 上で多数のプロセス・スレッドを用いたアプリケーションを動作させる．このため，演算 OS からのファイル I/O 委託要求が，短時間に集中する可能性がある．本研究では，ファイル I/O 委託要求の受信からファイルシステム呼び出しまでの処理を並列化することで，複数のファイル I/O 委託要求を同時に処理する．これにより，演算 OS 上アプリケーションにおけるファイル I/O の待ち時間を削減する．

演算 OS から管理 OS に向けたファイル I/O の要求手順，および直接 I/O アクセスの流れを**図 2** に示す．管理 OS では，演算 OS の I/O ライブラリによるファイル I/O 委託要求を受け付け，ファイルシステムへ I/O アクセスを依頼する．その後，I/O アクセスの実行結果を演算 OS の I/O ライブラリへ返答する．ファイルシステムへの I/O アクセス依頼にあたっては，演算 OS 上アプリケーションの I/O バッファを I/O アクセス先とすることで，管理 OS のファイルシステムと演算コア側メモリの間で直接データ転送を

表 3　ファイル I/O 代行における OS 間通信の内容
Table 3　Inter-OS communication item for remote I/O access.

項目	通信方向，通信種別 通信コンテキスト
ファイルのオープン要求	演算 OS →管理 OS：非同期通信 ファイルパス，オプション
ファイルのクローズ要求	演算 OS →管理 OS：非同期通信 ファイル識別子
ファイルへの リード・ライト要求	演算 OS →管理 OS：非同期通信 ファイル識別子， I/O バッファアドレス，I/O サイズ
ファイルの I/O 位置設定	演算 OS →管理 OS：同期通信 ファイル識別子， 基準位置，オフセット
ディスクキャッシュの フラッシュ要求	演算 OS →管理 OS：非同期通信 （なし）

行う．以上の手順で，低遅延な I/O アクセスを実現する．

4.4　ファイル I/O 代行における OS 間通信

本計算機では，演算 OS 上で発生したファイル I/O を，演算 OS と管理 OS で分担処理する．このため，演算 OS から管理 OS へファイル I/O を委託するための OS 間通信が必要となる．本研究では，管理 OS 上で構成するファイルシステムへのアクセス機能を，演算 OS 上の I/O ライブラリへ提供するための OS 間通信として，**表 3** に示す通信内容を定義する．演算 OS 上の I/O ライブラリで提供するファイル I/O API に対応するため，基本的なファイル操作を実現するファイルのオープン・クローズ，リード・ライト，I/O 位置の設定要求や，ディスクキャッシュのフラッシュ要求等の通信を定義した．

本研究では，OS 間通信種別として同期通信と非同期通信を設けた．同期通信と非同期通信はともに双方向型の通信としているが，非同期通信では，受信者が返答を 2 度行う．1 度目の返答は受信後の一定時間内に行うが，2 度目の返答には時間制約を設けない．ファイルへのリード・ライト要求等，返答までの時間が不定となる通信を非同期通信とし，受信側による I/O アクセスの開始前に 1 度目の返答を発行することで，送信側は通信が正常に行われたか否かを一定時間内に判別することができる．

4.5　ファイル I/O 以外の資源管理代行方式

本計算機では，ファイル I/O 以外の資源管理として，演算コア側のプロセス管理，CPU コア管理，メモリ管理の一部を管理 OS で代行する．プロセスの代行管理では，演算プロセスの生成管理を管理 OS が代行する．演算プロセスで動作させるアプリケーションを管理 OS がメニーコア側のメモリへ直接展開するとともに，仮想アドレスを実現するためのページテーブルを生成・管理することで，演算 OS は管理 OS から通知されたアプリケーションのエントリポイントのみで，演算プロセスの実行を開始することができる．CPU コアおよびメモリの代行管理では，演算プロセスに対する CPU コアおよび物理メモリの割り当てを管理 OS が代行する．これにより，管理 OS 内で演算プロセス生成処理が完結するため，特に大量の演算プロセスを一度に生成する際のオーバーヘッドを削減できる．また，演算 OS で発生したページフォルト等の例外処理への対処を管理 OS で代行することで，演算 OS はプログラムの実行のみに専念することが可能となる．以上の代行処理により，メニーコアの演算能力を可能な限りアプリケーションの実行に割り当てることで，計算機全体の並列演算性能向上を実現する．

5.　演算 OS 向け I/O ライブラリと管理 OS によるファイル I/O 代行方式の実現方法

4 章で述べたファイル I/O 代行方式を実現するためには，(1) 演算 OS と管理 OS でファイル I/O を分担処理するためのコンテキスト管理方式，(2) 管理 OS における低遅延なファイル I/O の実現方式，(3) 管理 OS から演算コア側メモリへの直接 I/O の実施方式，また，(4) 高効率な OS 間通信の実現方式が課題となる．以下では，これらの課題に対して，実現方法を考察する．

5.1　ファイルディスクリプタの管理方式

ファイルディスクリプタ（以下，FD）は，演算 OS 上の I/O ライブラリで POSIX.1 互換 API を利用するための識別子である．FD はファイルシステムを操作するための識別子を仮想化した存在であるため，ファイルシステムを呼び出す I/O ライブラリは演算 OS 側に，ファイルシステムは管理 OS 側に構成する本計算機では，(1) 演算 OS が FD を管理する方式，(2) 管理 OS が FD を管理する方式が考えられる．しかし，(1) の方式では演算 OS 上の I/O ライブラリに，FD を管理するためのテーブル（以下，FD テーブル）を構成する必要があるため，I/O ライブラリの軽量性を損ねる可能性がある．

本研究では，演算 OS におけるファイル I/O 処理を可能な限り管理 OS 側へ委託する方針としているため，FD を管理 OS 側で管理する (2) の方式を用いる．本方式を用いることで，演算 OS 上の I/O ライブラリは，POSIX.1 互換 API を介して受け取った FD を，そのままの状態で管理 OS へ送信することができるため，演算 OS による擾乱の低減が期待できる．

5.2　管理 OS におけるファイル I/O 機構の実現方式

管理 OS におけるファイル I/O 機構では，演算 OS 上で発生したファイル I/O を処理するためのファイルシステ

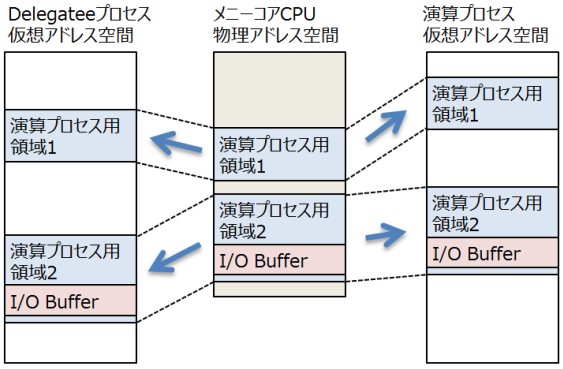

図 3 演算プロセス用物理メモリ領域のマッピング
Fig. 3 A memory mapping of the LW-process memory region.

ムと I/O デバイスドライバを構成する．本研究では，管理 OS である Linux が備えるファイルシステムと I/O デバイスドライバを，演算 OS から委託されたファイル I/O の処理に用いることで，管理 OS におけるファイル I/O 機構を実現する．独自の実装ではなく，Linux の機能を流用することで，既存の優れたファイルシステムの機能や，豊富なデバイスドライバの恩恵を受けた，ファイル I/O 機構を実現することができる．

Linux のファイルシステムは，プロセスコンテキストから呼び出されることを前提に構築されているため，OS 間通信を IPI ハンドラで受信する本計算機では，割り込みコンテキストからプロセスコンテキストへの切り替えが必要となる．本研究では，演算 OS から委託されたファイル I/O を処理するための Delegatee プロセスを管理 OS 上に生成することで，Linux のファイルシステムを利用したファイル I/O を実現する．Delegatee プロセス内では，複数のファイル I/O 委託要求を同時に処理するため，ワーカースレッドを複数個生成する．各ワーカースレッドは，ファイル I/O 委託要求の受信に備え，カーネル空間内で演算 OS からの OS 間通信を待機する．要求を受信した場合は，カーネル空間内で直ちにファイルシステムを呼び出すことで，低遅延に I/O アクセスを開始することができる．

本計算機における演算 OS では，演算プロセスごとにファイル I/O 資源を割り当てる方針としている．これを実現するため，本研究では Delegatee プロセスを演算プロセスごとに生成し，各 Delegatee プロセスごとに FD テーブルを設けることで FD を管理する．Delegatee プロセスと演算プロセスの対応を固定とすることで，異なる演算プロセスのファイル I/O 資源に対するアクセスを防止し，システムのセキュリティを保つ．

5.3 管理 OS による直接 I/O アクセスの実現方式

管理 OS におけるファイル I/O 機構では，管理 OS 上のファイルシステムから演算 OS 側の I/O バッファへ直接 I/O アクセスを行うことで，I/O アクセスの所要時間を削減する．Linux のファイルシステムにおいて，演算 OS 側の I/O バッファに対する直接 I/O アクセスを実現するためには，**図 3** のように，ファイルシステムの呼び出し元である Delegatee プロセスの仮想アドレス空間に対して，演算コアの物理メモリを予めマッピングする必要がある．その上で，I/O アクセスが要求される度に，演算プロセス仮想アドレス空間における I/O バッファのアドレスと，Delegatee プロセス仮想アドレス空間における I/O バッファのアドレスを変換する必要がある．I/O アクセスの所要時間削減には，アドレス変換を低遅延に実施する必要がある．以下では，アドレス変換に既存のページテーブルを用いる方式 (1) と，専用のページテーブルを用いる方式 (2) をそれぞれ提案する．

(1) 既存のページテーブルを用いたアドレス変換

演算プロセスの仮想アドレス空間と，Delegatee プロセスの仮想アドレス空間を結びつける情報として，両プロセスの仮想アドレス空間を構成しているページテーブルが存在する．ページテーブルは仮想アドレスを物理アドレスへ変換するためのテーブルであるため，演算プロセスの仮想アドレスを一旦物理アドレスへ変換した上で，Delegatee プロセスの仮想アドレスへ変換することになる．また，物理アドレスから Delegatee プロセスの仮想アドレスへの変換は，通常のページテーブル探索の逆順で行う必要があるため，多大な遅延時間の発生が見込まれる．物理アドレスから Delegatee プロセスの仮想アドレスへの変換に専用のページテーブルを用意することで，遅延時間を削減することができるが，2 つのページテーブルを探索する手順に変わりはない．

(2) 専用のページテーブルを用いたアドレス変換

本方式では，演算プロセスの仮想アドレス空間と，Delegatee プロセスの仮想アドレス空間を直接結びつける Shadow ページテーブルを管理 OS 上に構築し，演算プロセス生成処理の過程で両プロセスの仮想アドレスのエントリを追加する．本方式におけるアドレス変換では，1 つのページテーブルを探索するのみでアドレス変換が完了するため，2 つのページテーブルを探索する必要がある方式 (1) よりも低遅延にアドレス変換を実現できる．

本研究では，低遅延なアドレス変換処理を実現可能な方式 (2) を用いて，演算プロセスの仮想アドレスから Delegatee プロセスの仮想アドレスへの変換を実施する．管理 OS 側でアドレス変換処理を実施することで，演算 OS 上の I/O ライブラリは，アドレス変換を一切行わずに I/O アクセス要求を発行することができる．これにより，演算 OS 上の I/O ライブラリを軽量化する．

5.4 OS 間並列通信の実現方式

本計算機ではメニーコア側の資源管理の一部を管理 OS

表 4　ファイル I/O 依頼時のオーバーヘッド
Table 4　The overhead of file I/O request.

項目	所要時間	割合
OS 間通信	2.24us	46%
ワーカースレッド呼び出し準備	1.09us	22%
ワーカースレッド呼び出し	0.76us	16%
Shadow ページテーブルの探索	0.34us	7%
その他	0.45us	9%
合計	4.88us	100%

で代行するため，メニーコア上で複数のプロセスおよびスレッドが同時に動作した場合，OS 間通信路が逼迫し，各コア上のアプリケーションに待ち時間が発生する恐れがある．このため，本研究では，複数のコアからの通信を同時に処理可能な並列通信路を構築し，計算機全体の演算性能向上を実現する．

本計算機における OS 間通信路は，メニーコア CPU とマルチコア CPU から直接アクセス可能な共有メモリ上に構築したパケットキューと，CPU 間割り込みである IPI（Inter-Processor Interrupt）で構成する．通信データの交換に共有メモリを，通信の開始を IPI で通知することで，共有メモリ領域のみで通信路を構築する方法と比較し，ポーリング排除による CPU 時間削減を実現できる．また，共有メモリ上へキューを構成することで，複数パケットの一括送受信とパケットの取りこぼし対策を実現する．

並列通信路を実現するにあたり，1 つのパケットキューを複数単位の通信で流用した場合，キューの操作時に排他制御が必要となり，通信の並列性が損なわれるため，本研究ではパケットキューを多重化する．また，キューごとに専用の IPI ベクタ番号を割り当てることで，IPI により起床した OS 間通信のレシーバは，O(1) で読み取り先のパケットキューを特定することができる．

6. 評価

6.1 評価システム

本研究では，Intel Xeon X5690（6 コア，3.47GHz）プロセッサを 2 個搭載した NUMA 型計算機を，マルチコア・メニーコア混在型計算機に見立てて評価環境を構築した．一方のプロセッサをマルチコアプロセッサとして，管理 OS である Linux を搭載した．もう一方のプロセッサをメニーコアプロセッサとして，独自に開発中の演算 OS カーネル “Future” を実装した．本環境に対して，演算 OS 向け I/O ライブラリと管理 OS 側の I/O 代行機構を試作した．

6.2 評価と考察

I/O 代行方式の性能を評価するため，はじめに，演算 OS から管理 OS へファイル I/O を依頼する際のオーバーヘッドを測定した．測定結果を**表 4** に示す．ファイル I/O 依頼

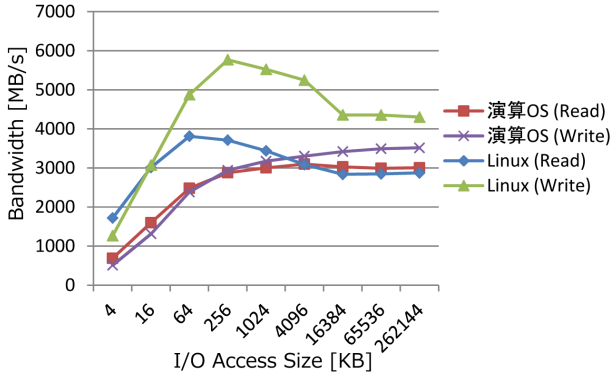

図 4　read/write アクセスの帯域幅
Fig. 4　The band width of read/write access.

にともなうオーバーヘッドは全体で 5us 程度となった．その内訳は，約半分が OS 間通信にともなう遅延時間，残りが管理 OS の I/O 代行機構での遅延時間である．管理 OS 側ではワーカースレッドの呼び出し準備に時間を要しているが，呼び出すワーカースレッドの決定が主たる処理となるため，実装方式の改良で時間削減が可能である．Shadow ページテーブルの探索は全体の 7%の所要時間で完了しているが，既存のページテーブルを用いた方式では少なくとも 2 倍程度の時間を要するため，Shadow ページテーブルの導入により全体の遅延時間を 6%程度削減することができた．

次に，演算 OS 上のアプリケーションにおいて POSIX.1 互換 API の read/write 関数を呼び出し，管理 OS で構成した RAM ディスクへ I/O アクセスを発行した際の所要時間を測定した．I/O アクセスサイズによる帯域幅への影響を評価するため，4KB から 256MB までの 9 種類のサイズについてそれぞれ測定を行った．また，比較対象として，管理 OS である Linux 上のアプリケーションからも同様の測定を行った．これらの測定結果を**図 4** に示す．

演算 OS からの read/write では，I/O アクセスサイズの増加にともない帯域幅が増加しているが，比較対象の Linux では，I/O アクセスサイズが 64KB から 256KB の間で帯域幅が最高値を示している．本試作システムの L3 キャッシュサイズである 16MB を超えた I/O アクセスでは帯域幅の変動が収束することから，CPU のキャッシュメモリによる性能向上であると考えられる．演算 OS からの read/write で Linux と同様の性能向上が発生しない要因は，NUMA 環境によるメモリアクセスコストの差であると考える．演算 OS からの read/write では，管理 OS が動作するマルチコアから演算 OS が動作するメニーコアに対するデータ転送が発生するため，ローカルメモリへのアクセスを行う Linux よりも帯域幅が小さくなる．16MB 以上の I/O アクセスサイズでは，read アクセスが Linux と互

角の帯域幅を確認することができたが，write アクセスは Linux の約 0.82 倍に収束した．今後，他の I/O デバイスに対する I/O アクセス性能の評価を進め，write アクセスにおいて Linux との性能差が発生する要因を明らかにする．

本評価における最小 I/O アクセスサイズの 4KB では，read/write の帯域幅が Linux の 0.4 倍となった．より大きな単位での I/O アクセスにおいては Linux との性能差が小さくなるため，OS をまたいだ I/O アクセスを実施することによる遅延時間が，性能低下の原因であると考える．小さな単位での I/O アクセスに関しては，POSIX.1 互換 API ではなく標準入出力ライブラリの API を経由することで，バッファリングにより OS 間通信の頻度を削減し，帯域幅の改善を行うことができる．

7. 関連研究

BlueGene[7][8] や Shimizu 等 [9] は，並列演算処理に特化した計算専用ノードと，計算プログラムの資源管理を行う管理ノードで構成されるクラスタシステムを提案している．計算専用ノードに軽量の OS を，管理ノードに汎用 OS を搭載する OS 構成や，汎用 OS 上のファイルシステムで計算プログラムの I/O アクセスを処理する方式は同じであるが，本研究では，内部バスを介してプロセッサ間通信とメモリ共有が可能な点を生かし，汎用 OS から軽量 OS へ直接 I/O を実施する点が異なる．

fos[10] は，CPU コア間通信機能を備えたマイクロカーネルにより，複数の CPU コアへファイルシステム等の OS 機能を分散させた，メニーコアおよびクラスタ向けの OS である．本研究では，対象としているプロセッサの特性が異なる 2 つの OS でファイル I/O を分担処理している点や，直接 I/O を実施する点が異なる．

8. おわりに

本論文では，マルチコア・メニーコア混在型計算機において，メニーコア上の演算 OS で発生した I/O アクセスを，マルチコア上の管理 OS が協調して処理するための，演算 OS 向けファイル I/O ライブラリおよび管理 OS 向けファイル I/O 機構を提案した．演算 OS の動作による擾乱を極力削減するための I/O ライブラリと，管理 OS による低遅延な I/O アクセスの実現方式について述べた．Intel Xeon X5690 プロセッサにて構築した模擬環境上で，演算 OS 上の I/O ライブラリと管理 OS 上の I/O 代行機構を試作し，ファイル I/O の性能を評価した．その結果，演算 OS から管理 OS への I/O 要求を 5us 程度の遅延で実現可能であることを示した．また，read 関数による I/O アクセスでは，I/O アクセスサイズを 16MB 程度に設定することで，Linux 単体でのファイル I/O と同等の帯域幅を確認した．

今後は，多数の演算プロセスが動作する環境において本 I/O 代行方式の性能を評価し，同時 I/O 要求に対する性能改善を行なっていく．また，演算 OS 向け I/O ライブラリをノード間 I/O へ対応させることで，本計算機をクラスタ化し，並列演算性能のさらなる向上を目指す．

謝辞 本研究は，科学技術振興機構（JST）戦略的創造研究推進事業（CREST）における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」研究課題「メニーコア混在型並列計算機用基盤ソフトウェア」によるものである．

参考文献

[1] Top500: Supercomputer Sites (online), Available via “http://www.top500.org/” (accessed June.26.2012).

[2] Intel: Many Integrated Core (MIC) Architecture - Advanced (online), Available via “http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html” (accessed June.26.2012).

[3] Skaugen, K.: Petascale to Exascale - Extending Intel’s HPC Commitment (online), Available via “http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf” (accessed June.26.2012).

[4] 深沢豪, 佐藤未来子, 長嶺精彦, 坂本龍一, 吉永一美, 辻田祐一, 堀敦史, 石川裕, 並木美太郎: マルチコア・メニーコア混在型計算機における演算コア側資源管理の代行方式, 情報処理学会「ハイパフォーマンスコンピューティング」第 134 回研究報告, Vol. 2012-HPC-134, No. 7, pp. 1–8 (2012.06.01).

[5] Yoshinaga, K., Mugurama, H., Tsujita, Y., Hori, A., Namiki, M., Sato, M., Shimosawa, T., Sawada, T. and Ishikawa, Y.: Realizing Scalable MPI Communication on a Heterogeneous Platform with Multi-Core and Many-Core CPUs, *Proceedings of the Work in Progress Session, PDP2012* (2012).

[6] THE Open GROUP: The Single UNIX Specification, Version 4 (online), Available via “http://http://www.unix.org/version4/” (accessed June.26.2012).

[7] Giampapa, M., Gooding, T., Inglett, T., Wisniewski, R.W.: Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene’ s CNK, In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’10, pp. 1–10 (2010).

[8] Moreira, J., Brutman, M., Castaños, J., Engelsiepen, T., Giampapa, M., Gooding, T. *et al.*: Designing a highly-scalable operating system: the Blue Gene/L story, In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC ’06 (2006).

[9] Shimizu, M., Yonezawa, A.: Remote process execution and remote file i/o for heterogeneous processors in cluster systems, In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID ’10, pp. 145–154 (2010).

[10] Wentzlaff, D., Gruenwald III, C., Beckmann, N., Modzelewski, K., Belay, A., Youseff, L., Miller, J., Agarwal, A.: An operating system for multicore and clouds: mechanisms and implementation, In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC ’10, pp. 3–14 (2010).