卒業論文

MPI とマルチスレッドによる MD5 を用いたファイル修復の並列化

氏　　名　　：渡部 宏

学籍番号　　：2210010259-9

指導教員　　：山崎　勝弘　教授

提 出 日　　：2005 年 2 月 21 日

立命館大学理工学部情報学科

内容梗概

本論文では対称型マルチプロセッサ、SMP 計算機から構成されるクラスタを活用するために、MPI とマルチスレッディングの併用による並列化の検証を行った。異なる並列化パラダイムの混在や、同期処理に関する複雑化を回避するために、同期インターフェイスによる自律計算スレッドを考案し、その効果を検証した。

並列化の対象としたアプリケーションは MD5 によるファイル修復で、全数検索を基本にする処理である。MD5 の計算自体に有効な攻撃手段が確立されていないため、全数検索処理を総体的に高速化させる MD5 計算コンテクストのキャッシュを導入し、その効果を確かめた。

実際に MPI と POSIX スレッドによる並列化を行ったソフトウェアを開発し、クラスタで動作させて速度を検証した結果、MPI だけの場合よりも高速であり、16 台 32 プロセッサで 25.3 倍の速度向上比を確認した。一部の処理負荷が少なすぎるケースでは、動的スケジューラの影響もあり、期待した速度向上が得られなかったが、想定している実用範囲では十分な性能を確認できた。並列化混在によるソフトウェア開発時の複雑化を回避し、保守性を向上させつつ、性能を上げることができた。

目次

図目次

表目次

1. はじめに

1.1 背景

SMP (対称型マルチプロセッサ)による計算機の処理能力向上はハイエンドサーバー市場を中心に行われてきたが、昨今では PC レベルでも進んでいる。この流れは近年 PC プロセッサ市場において、発熱や消費電力などの物理特性による動作周波数限界が現実問題として浮上してきた結果、単一演算ユニットの速度向上よりも複数演算ユニットによるスレッド動作効率の向上が狙われ始めたためであり、今後主要ベンダーのマルチコア構想の影響で更に拍車がかかると思われる。

PC クラスタ構築においては性能とコストの両者のバランスが重要である。多数の計算機でクラスタを構築するよりも、各計算機内の演算ユニットだけ増えるほうがコスト面では有利な場合が多い。同じ数の演算器ならば 2way から 4way 程度の SMP 計算機ノードを用いてクラスタを構築する方が、2 倍、4 倍の台数の非 SMP 計算機ノードで構築することよりも、初期導入費用も稼動コスト(電気料金等)も低く、メンテナンス性にも優れている。加えて、一般的にクラスタではボトルネックはネットワークとも言われるため、SMP 計算機によるクラスタ構成は演算器を増やしつつネットワークトラフィックを抑える効果も期待でき、より高速な処理を見込める。

1.2 問題点

しかし、SMP のようなメインメモリを共有する密結合マルチプロセッサ計算機で構成される並列処理環境では、MPI や PVM といったプロセスレベルの分散メモリ型並列化では無駄がある。例えば図 1 のように、演算器の個数にプロセスの個数を調整することで演算器資源を利用すると、計算用のデータなどで重複が発生する可能性がある。各プロセスの連携も MPI などの高位インターフェイスを経由したプロセス間通信になりオーバーヘッドが大きい。また C++のようなオブジェクト指向プログラミング言語を用いる場合、オブジェクトを外部プロセスとやり取りするには直列化や特殊なオブジェクトモデルが絡む場合があり煩雑になる。

理想をいえば、SMP のようなメインメモリを MPU が共有する計算機内部では、MPI や PVM のような分散メモリ型の並列化ではなく、OpenMP やマルチスレッディングなどによる共有メモリ型の並列化を行うべきである。つまり MPI と共有メモリ型の並列化を組み合わせて使うことによって、スレッド間で共有できる資源は共有しつつ、能率よくクラスタのマルチプロセッサ資源を利用できる。実際に OpenMP と MPI を組み合わせて使った事例は多く存在する。

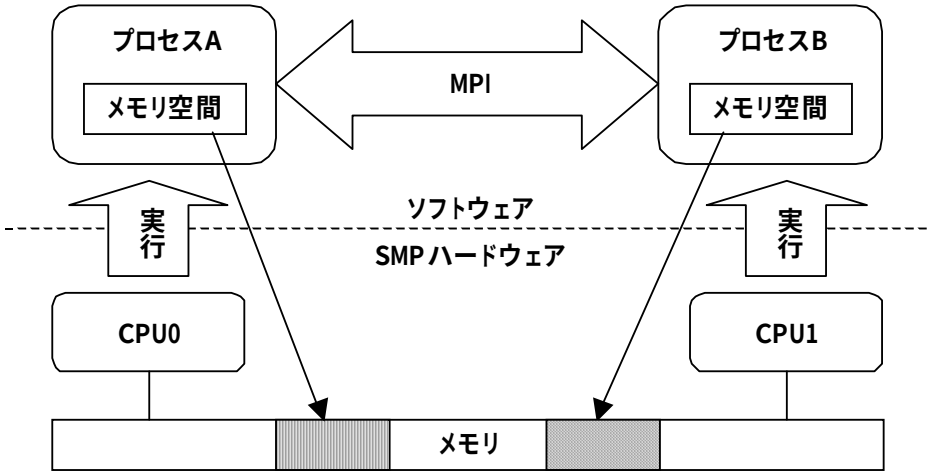

図 1　SMP 環境で 2MPI プロセスが 1 台の計算機内で並列動作しているイメージ

本研究では OpenMP ではなく、より一般的な共有メモリ型のパラダイムであるマルチスレッディングを MPI と併用する。図 1 で挙げた動作イメージは、マルチスレッドを併用すると図 2 のように置き換えられる。同一計算機内部での MPI 通信を取り除き、共有可能なメモリ資源を共有することで、速度と資源利用効率の向上を図る。

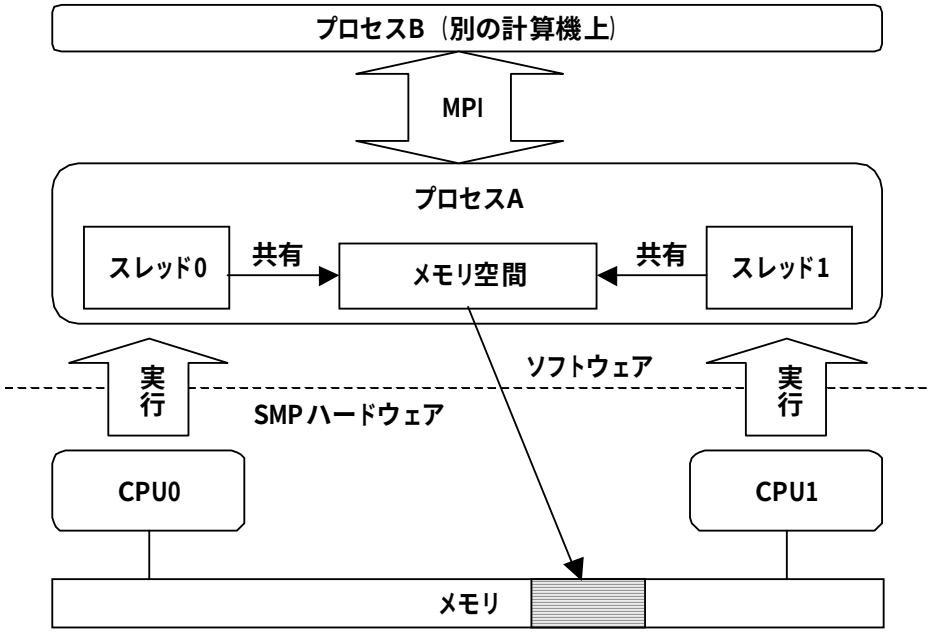

図 2　SMP 環境において 1MPI プロセスで 2 つのスレッドが並列動作するイメージ

また、これまでマルチスレッディングに関しては大枠の概念で共通点が多いものの、各 OS などで独自のインターフェイスの策定と実装が行われてきた経緯があり、OpenMP と MPI の併用とは標準性や汎用性において事情が異なる。

インターフェイスに関して例を挙げると、例えば Windows でサポートされる Win32 スレッドは、他のスレッドモデルとのソースコードレベルでの互換性も、バイナリ互換性も無い。マルチスレッドは最近になって POSIX スレッド(通称 pthread)として標準化が成熟し、多くの UNIX ライクシステム(Linux など)でライブラリとして実装され普及しつつあるが、Windows では現状でも pthread のサポートはなされていない。

各種ネイティブスレッドの実装もここ数年で改良されてきている。例えば Linux は 2003 年 12 月にリリースされたカーネル 2.6 系列で futex (Fast Userspace Mutex)が新たに支援されたことで、ユーザー空間での同期処理を行えるようになり、システムコールを必要とするケースが減ることが期待でき、同期処理を多用するプログラムの高速化が実現されている。

MPI に関しては各実装でのスレッドセーフ問題が懸念される。スレッドセーフとは広義には複数スレッドからの利用が安全であるということであり、例えば特定のシステムコールなどが内部で状態を持つ場合、その状態が同期的に処理されることなどで達成される。スレッドセーフでないコードを複数スレッドが利用した場合、未定義の振る舞いになる。

MPI におけるスレッドセーフも多岐にわたる意味を持つが、特に広く普及している MPICH においては 2004 年 12 月にリリースされた MPICH2 バージョン 1.0 で公式にスレッドセーフが達成されており[18]、より古いものは非スレッドセーフであることが知られている。他にオープンソースで提供される MPI 実装である LAM[19]も現時点ではスレッドセーフではないとされる。これらの実装では MPI プロセス内で複数スレッドが起動することには問題はないが、厳密に MPI を実行するスレッドが単一であることをユーザーが保証する必要がある。ちなみに SGI の提供する商用ＵＮＩＸである IRIX 6.5 上の MPI 実装では、MPI_thread_init という特別に提供される関数を利用することで、複数のスレッドからの MPI 実行を安全に行うことが可能になっている。その他にも多くの商用 UNIX 環境での MPI 実装はスレッドセーフである、もしくは内部で同期処理を行う実装が多いようである。

つまり MPI とマルチスレッディングの組み合わせは概念的には真新しいものではないが、スレッドセーフな MPI 環境は一般的に普及しているとは言いがたく、また標準としての POSIX スレッドも十分に普及しているわけではない。実用に際して、研究の余地は大きい。

また、並列処理ではなんらかの同期処理が存在する。MPI とマルチスレッドを組み合わせて使う場合、両者が異なるパラダイムであるため、プログラムは特に同期処理に関して複雑化してしまう。これは単なるプログラミング作業における煩雑さだけに留まらない。各種同期処理によってもたらされるプロセスあるいはスレッドレベルのブロッキングによる停止性は、プログラムの性能低下や、最悪の場合、意図しないプログラムの中途での停止を招き得る。安直に併用するだけでは、信頼性と性能を両立させるどころか、品質維持を難しくするだけに終わってしまう可能性が懸念される。

1.3 目的と概要

本研究では MPI と POSIX スレッド併用による並列化について、MD5 を利用したファイル修復

アプリケーションの作成と評価を通して考究する。懸念される MPI 実装のスレッドセーフ問題やスレッド同期処理による複雑化を回避するために、同期インターフェイスによる自律型計算スレッドを考案し、可能な限りソフトウェアをシンプルにしつつ、資源利用効率と性能、そして汎用性を向上させることが狙いである。

並列化の対象とする MD5 によるファイル修復アプリケーションは、MD5 の認証性を利用して一部分が破損したファイルの修復を試みる。MD5 は強い衝突耐性があり、現在までに有効な攻撃手段が確立されていないため、修復は可能性のある修復案を全数検索によって列挙し、各々の MD5 ハッシュを元のものと比較することで行う。計算量は膨大になるが個々の計算の独立性があるので、容易に並列化できることが本研究で選んだ理由である。

本研究で考案した同期インターフェイスによる自律計算スレッドとは、いってみればプロセッサ演算能力の抽象としてのスレッドの構築である。純粋に並列化された部分的な計算タスクを行うためだけのスレッドであり、効果的に演算能力を活用するために未消化の仕事があるときだけに稼動するように自律化させる。計算スレッドのインターフェイスはスレッドレベルの同期が取られるものに限定する。例えば、今回はタスクオブジェクトを扱うキューを入出力にそれぞれ 1 つずつ利用しており、計算スレッドは入力キューが空の場合はその状態で停止し、何らかの入力が発生したときに自動的に活動を開始し、処理を行って出力キューへ出力する。適切な抽象化を行うことで、タスクの MPI 経由でのプロセスへの割り当てから、同期インターフェイス経由での処理、結果の修得、その結果の別計算への利用、そして MPI 経由での別プロセスへの結果情報送出などが、ほぼ同一の手法で実装でき、ソフトウェア開発時の MPI とマルチスレッド併用に関する煩雑さはかなり削減できることが期待できる。

1.4 本論文の構成

2 章では並列化対象である MD5 によるファイル修復について述べる。全数検索の必要性と理論上の修復の限界と現実的な処理時間内で可能な修復の限界について述べ、高速化の手段として考案した MD5 ハッシュ計算コンテクストのキャッシュについてその効果を含めて解説する。

3 章では MPI とマルチスレッドの併用による並列化を解説する。利点と欠点、MPI を排他利用する必要性を挙げ、今回提案するタスクキューによる同期インターフェイスを用いた自律型計算スレッドについて実装を踏まえて解説する。

4 章では MPI レベルのスケジューリングについて触れる。静的な巡回スケジューリングの問題を挙げ、次に今回考案した待機タスク数に基づく動的スケジューリングを解説する。

5 章において実際に作成したプログラムを SMP 計算機で構築されたクラスタで実行させ、MPI とマルチスレッド併用による並列化の効果を主に処理時間の面から検証する。プログラミングの手間や難易度についてもあくまで主観であるが意見を述べている。

6 章では本研究のまとめを述べている。

2. MD5 を利用したファイル修復

MD5 は広く普及したハッシュアルゴリズムである。MIT の Ron Rivest が開発し RFC-1321 によってインターネット上で標準化されている[1]。

MD5 は任意長(0〜無限ビット)の入力メッセージに対して 128 ビットのハッシュ値を計算する。高い認証性があるため、一時期はパスワードハッシュにも用いられたが、現在ではセキュリティ上十分な強度が無いことが指摘され、主にインターネットを通じてファイルを配布する際に、送信されたファイルが通信によって破損、或いはクラッキングなどで改竄されていないかを検証する目的等で広く用いられている。例えば、Debian GNU/Linux のパッケージ管理機構である APT では、パッケージファイルを自動的にインターネット経由などでサーバーからダウンロードした際に、MD5 によるパッケージファイルの整合性の検証を行っている。

本研究で開発する MD5 を用いたファイル修復アプリケーションでは、MD5 の認証性を利用して一部分が破損したファイルの修復を試みる。破損したファイルと、元のファイルの指紋ともいえる 128 ビットの MD5 ハッシュを利用し、破損部位と推定されるその部位の元の内容を全数検索にかけることで、元のファイルと同一と見なすことができる修復候補を検索する。

2.1 概要

まず対象とする N バイトのファイル original を考える。予め MD5 ハッシュ H(original)を計算しておき、H(original)と共にファイル original を保存したとする。

ファイル original を再度利用する際、再度 MD5 ハッシュを計算し、保存しておいたハッシュと比較することで、その整合性を検証できる。例えば 1 ビットでも反転があれば 128 ビットのハッシュ値は全く異なるものになるので、保存前のハッシュ値と再計算したハッシュ値が同一であればファイル全体も同一と考えることができる。

この検証に失敗したことを想定し、破損しているファイル malformed と H(original)を用いて、ファイル original の候補であるファイル(群)candidate の獲得を試みる。図 3 に概要を示す。

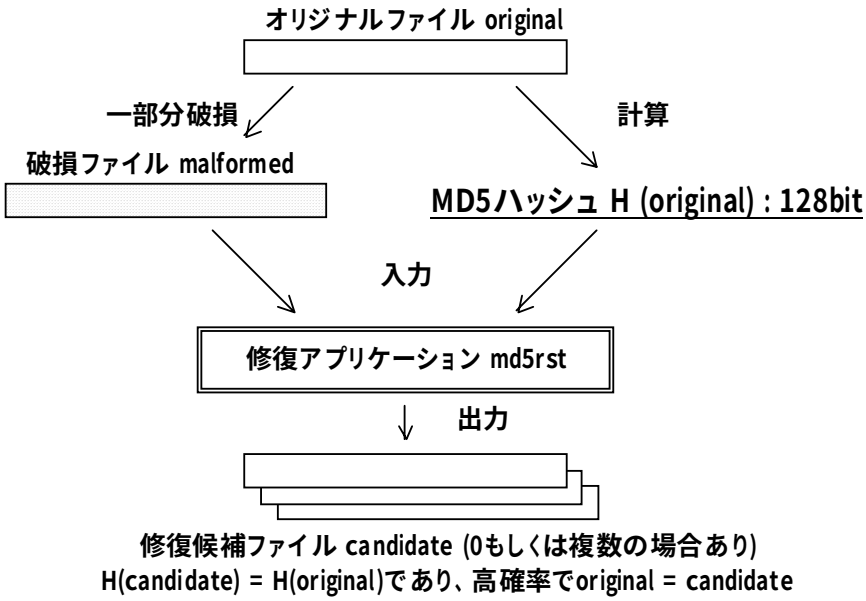

図 3 ファイル修復の概要

malformed をヒントに用いて candidate を列挙し、H(original) = H(candidate)を検索する。MD5 の認証性からいって、ハッシュ値が一致する candidate は original に等しい確率が高い。

なお修復候補は複数生じえるが、複数の候補から original に等しいものを検出することは MD5 の利用では不可能である。ハッシュ値が一致することは MD5 認証において同一とみなされることを意味するためであり、その場合は他の手段で検証する必要がある。

2.2 修復の限界

MD5 ハッシュは 128 ビットである。[2]の誕生日のパラドックスで述べられているとおり、$2^{\frac{128}{2}}$ 程度の施行回数でハッシュの衝突が発生する確率は 50%を超える。衝突とは異なる入力で同一のハッシュ値が生じるケースのことを指す。つまり破損が 64 ビット以上にわたる場合は複数の候補が発生することが懸念される。そのような場合は結果として修復候補を得ることができても(そしてそれが唯一の候補であっても)、元のファイルと同様のものが検出されたとは限らない。逆に、極めて小さな施行回数でも衝突が発生する可能性は存在する。なお 64 ビット以上の破損の想定は後述する計算量の観点から言っても現実的ではない。

これらの理由から、修復の限界は総計して高々64 ビット以内の破損とするのが妥当である。また破損個所は本来分散していても一箇所に集まっていても変わりないが、本研究では単一個所の破損に限定する。こういった制約についての詳細は 2.4.3 で述べる。

2.3 全数検索

MD5 は一方向性を持つ。これはハッシュアルゴリズムの特性としては必須条件であり、入力メッセージ a のハッシュ値 H(a)を考えるとき、H(a)から a を算出することができない。

MD5 は他のハッシュアルゴリズム同様、衝突耐性を持つ。衝突耐性には主に弱い衝突耐性と強い衝突耐性の二つがある。弱い衝突耐性のあるハッシュアルゴリズムにおいて、ある入力 a のハッシュ値 H(a)に対して、H(b) = H(a)となる別の入力 b を計算することはできない。強い衝突耐性のある場合は H(a) = H(b)となるような異なる入力の組(a, b)を計算することもできない。MD5 には両方の衝突耐性があると言われている。

このため今回のファイル修復おいて、保存していた MD5 ハッシュから元のファイルを計算することも、あるいは破損内容を計算によって推測することもできない。また現在までに MD5 に対する有効な攻撃手法も確立されていない。このため全数検索によって修復を試みる。図 4 にその概略を示す。

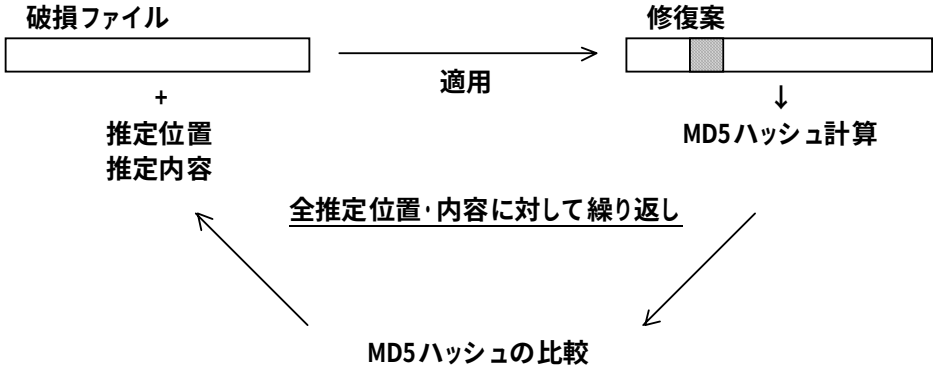

図 4 全数検索の概略

今回作成するアプリケーションはファイル malformed をヒントにし、推定位置と推定内容からなる修正案を全数検索に基づいて順次推測する。それぞれの修正案を適用した場合の MD5 ハッシュを計算し、保存されている MD5 ハッシュ H(original)と比較し、一致するハッシュを得ることができる修復案、つまり MD5 の衝突を引き起こすケースを検索する。

2.4 計算量

MD5 は入力を先頭から 512 ビットの単位で計算する。末尾には全体の入力が 512 ビットの倍数長となるようにパディングがなされ、それには全長の 2 の 64 乗の法による入力長の埋め込みも含まれる。512 ビットのブロックを用いて 4 つの 32 ビットレジスタに対して 4 つのラウンドからなる計算を

行い、それを全ブロックに対して順番に行う。最終的に 4 つのレジスタを並べた 128 ビットの値がハッシュ値である。各ラウンド内では F、G、H、I の 4 つの原始関数からなるビット演算やローテートなどを基本とする演算が行われる。原始関数の定義やラウンドでの計算は[1]で詳細に延べられている他、[2]でも解説されている。計算はリトルエンディアンで格納された整数として 32 ビット単位で行われる。リトルエンディアンとビッグエンディアンにおいて、一般的にどちらかに優位性があると言うことは無いが、現在広く PC で用いられている Intel 社の IA-32 型 MPU はリトルエンディアンであり、その他のビッグエンディアンの環境に比べると MD5 に関してはわずかながら計算コストが少ない。

一般的な入力長からするとパディングの占める率は極めて少ない点と、ブロック単位で行われる計算が分岐もループも存在しない単純な逐次演算である点を踏まえると、ブロック単位で計算を行うにもかかわらず実質的には計算の複雑さは O(n)と考えて差し支えない。つまり入力長に比例して MD5 計算に要するコストは増加すると考えられる。

全数検索処理の計算は推定位置の列挙、推定内容の列挙、そしてそれらを適用したメッセージに対する MD5 計算からなる。推定位置の検索回数を Co、推定内容の検索回数を Cp とすると、これらの検索は基本的に加減演算のみで計算コストは少ないため、Co*Cp 回の MD5 計算が全数検索の計算量のほぼすべてを占めると考えられる。

ファイルサイズを N バイト、破損区間長を L バイト (N>=L)とすると、Co、Cp はおのおの次のようになる。なお今回処理はバイト単位で行っており、1 バイトは 8 ビットと想定する。

$$Co = N - L + 1$$
$$Cp = 2^{8L}$$

よって MD5 の計算回数 Cmd5(N, L) は次の式で求めることができる。

$$Cmd5(N, L) = 2^{8L}(N - L + 1) \qquad \cdots \quad (1)$$

MD5 の計算の複雑さが O(n)であることを考えると、Cmd5(N, L) は N に比例するため全体の計算の複雑さは O(n^2)であることが推測できる。また、L に関しては指数的であり、2.2 で挙げた 64 ビットの破損の場合の計算量が非現実的であることは想像に難くない。

非並列版で実際にどの程度計算時間が増加するのか検証したところ、表 1 の通り入力ファイルサイズの 2 乗に比例して計算時間が増加していることが確認できた。

実際問題として N の数値はキロ以上のオーダーになることが多い。つまり最終的な計算量はメガ単位回数の MD5 の計算であり、膨大な計算時間を要することは表 1 からも確認できる。なお MD5 計算も全数検索処理もファイル内容には依存しないため、純粋にファイル長によって計算時間は決まる。

表 1 ファイル長と計算時間の関係

処理ファイル長 [バイト]	10000	20000	40000
ファイル長増加率	1.00	2.00	4.00
処理時間 [秒]	71.96	283.02	1122.60
処理時間増加率	1.00	3.93	15.60
推定処理時間増加率	1.00	4.00	16.00

2.4.1 衝突の発見と計算終了の条件

計算は全数検索の各推定内容と推定位置に対する MD5 計算で独立しており、各計算が終了した段階でその候補における衝突の有無が検証できる。このため衝突の発見は全体の計算の間で確率的に発生する。

一方アプリケーションの終了条件である全体の計算終了は、衝突が発見される、されないに関わらず、想定した範囲の全数検索が全て終了する時となる。

このためアプリケーションの計算が終了しても修復候補が無い場合があり、またアプリケーション終了以前にファイル修復を達成できる可能性もある。

2.4.2 計算順序と計算量の関係

本研究での全数検索はこれまで述べたように推定位置と推定内容の全組み合わせの検索である。これらを列挙するに当たって、計算機上での計算量は位置を固定して内容、パターンを列挙する場合と、パターンを固定して位置を列挙する場合で異なる場合がある。

これはパターンの列挙の実装方法によっては、位置の列挙とパターンの列挙が異なる負荷量になるためである。例えば、位置の列挙は単純な加算のみで行えるが、一定以上の区間長のパターン数は数回の加算を組み合わせることになり、位置の列挙に比べて数倍のコストとなる。具体的には 1 バイト整数の配列による生成の場合などが挙げられる。

このため順序としては、パターンの生成からそれを適用する位置の列挙とすることがより効率的になると思われる。ただし、パターン列挙は 32 ビット以内の場合は大抵の計算機において単純な加算のみで行えるため、この順序性が影響を及ぼすとは考えられない。実際に非並列版において差を検出することはできなかった。

2.4.3 修復への制限

今回の破損はファイル長に変化が無い場合を想定しているが、それでも完全な修復は不可能である。これは MD5 の認証性の限界のほかに、現実的な時間内での計算終了がある程度以上の条件での計算量では期待できないことに起因する。式(1)を見れば計算回数が破損長に対しては指数的に伸びることは明らかであり、2.2 でも述べた 64 ビット相当の破損長においては文字通り天文学的な計算回数、計算時間を要する。

そこで、認証性と計算時間の両側面を鑑みて次の制限を行った。

まず破損部位を単一の連続区間とする。これにより、複数区間を想定する場合と比べて推定位置の量をかなり軽減できる。複数区間を考える場合、それぞれの区間位置の組み合わせが生じ、それがやはり指数的に計算回数を増加させるためである。

破損区間長は最低で 1 ビット、最大で N バイトであり、全数検索では破損区間長の中の全パターンを考えるため、破損区間長を Lb ビットとするとパターン数は 2^{Lb} 個となる。1 バイトの区間を想定すると高々256 パターンだが、2 バイト区間になると 65536 パターンとなり、指数関数的に増加する。パターンの生成方法と現在主流の計算機のアーキテクチャ、ならびに演算時間の観点から、最大区間長は32ビットが実質的な上限になる。なお本研究の実験においては、主に時間の都合により 8 ビットの区間長を用いた。

2.5 MD5 ハッシュ計算コンテクストのキャッシュによる高速化

MD5 は 4 つの原始関数とローテートからなる複雑な計算を大量に行う。一般的な用途において、この計算時間はI/O処理時間などと比べて小さいため気になるものではないが、今回のアプリケーションでは無視できるものではない。

先ず簡単な負荷測定を行うために MD5 の計算のみを行うベンチマークを作成した。メモリ内のデータに対して 1MB 相当の入力値の MD5 計算を 1000 回行ったところ、Intel Pentium 4 2.6GHz の PC 上で 8 秒付近の計算時間を要することがわかった。今回の全数検索において、1MB というサイズは大きい分類ではあるが、1000 回という数値は極めて小さい分類である。式(1)を用いれば、1 バイト長の破損区間 1 箇所でも 256,000,000 回の MD5 計算が生じることがわかる。何らかの高速化が重要であるが、MD5 には現在有効な攻撃手法は確立されていないため、MD5 の計算自体に改良を加えるなどで高速化を図ることは見込めない。

そこで今回 MD5 の計算が先頭から順に行われる点に着眼した。MD5 の計算に関与する情報は 512 ビットのブロックと 128 ビットのレジスタ、そしてそこまでに処理した入力メッセージの長さの64ビットでの表現であるが、これらを修正パターン適用以前の部位まで計算した時点で計算コンテクストのキャッシュとして保存し、全数検索処理の高速化を計る。

計算が進んでいない場合、特に先頭付近では 512+128+64 ビット、つまり 704 ビット(=88 バイト)分のコピーを最低 2 度要する分デメリットともなりうるが、十分に長い入力(数 KB 以上)に対しては、ファイル後半における破損想定時の計算量削減に高い効果が期待できる。全体で理論上 2 倍弱の高速化が可能であると推測できる。

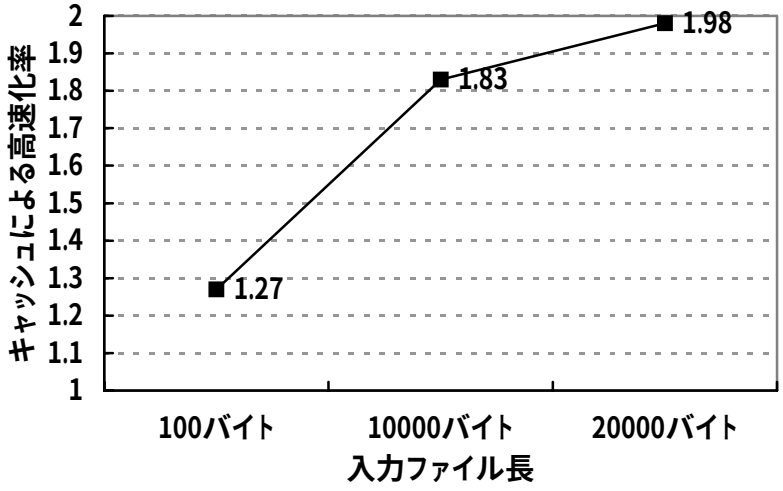

図 5　MD5 計算コンテクストのキャッシュによる高速化比率

この効果を非並列版においていくつかの処理サイズ別に測定したところ、図 5 に示す結果となった。サイズが極端に小さい 100 バイトでは効果は少ないが、10000 バイト程度で既に十分な速度向上が見られ、20000 バイトで理論最大である 2 倍に肉薄している。今回の実験では基本的にこのキャッシュによる高速化を行った上で実行時間を計測している。

3. マルチスレッドと MPI の併用による並列化

MPI は統一された形式でのメッセージ通信を行うインターフェイスで、プロセスレベルでの分散メモリ型並列化を実現できる。例えば複数の計算機をネットワークによって統合して連動させることができ、また単一計算機の中に複数プロセスを動かすこともできる。プロセス単位であるため総体として利用できるメモリ空間は広大になり、大規模問題の並列化による解決に用いることができる。

マルチスレッディングは軽量プロセスとも呼ばれる、実行コンテクストとしてのスレッドを単一プロセスの中に複数動作させる技術である。各 OS ベンダーがそれぞれの OS において独自に実装してきたが、標準として POSIX スレッド(pthread)が制定され、普及し始めており、UNIX ライク環境を始めとして多くの環境で利用できる。プロセスは単一のメインスレッドから始まり、必要に応じてスレッドを分離していくことが基本となり、プロセス内の全スレッドは同じプロセス空間に存在し、プロセス資源を共有する。それらの共有資源を必要に応じて明示的に排他制御する必要性があるが、SMP 計算機においてはハードウェアアーキテクチャとの親和性が高く、効率よく並列処理を行うことができる。

今回は MPI によってネットワークレベルで接続された計算機群による並列化を行い、さらにスレッドによって計算機内部の演算器単位での並列化を併用することでより効率の高い並列化を図る。

3.1 利点

3.1.1 高い資源利用効率

MPI だけでプロセス数を増やして演算ユニットを利用すると、マルチスレッドでは共有可能な資源もそれぞれのプロセスが持つためにメモリなどの資源利用効率は悪くなる。特に計算に大量の入力データや計算バッファを用いる場合、これはメモリ不足などの無視できない問題になる可能性がある。今回作成したソフトでも MD5 計算に用いるファイルデータをメモリ上に設置するが、MPI だけではプロセス単位でそれぞれがデータを持つことになり、明らかに無駄が多い。

MPI とマルチスレッドの併用を行うことで、計算スレッドが資源の一部を共有することで、データの重複などの無駄を省くことができる。なおこれは実際にどのような処理を行うかによって大きく変わるために、一概に得られる効果とは言えない。扱う問題が大規模な共有可能データを必要として、かつより多い演算器を搭載した SMP 環境ほど効果が出やすいと考えられる。

3.1.2 オブジェクト指向プログラミングとの親和性

今回プログラムは C++によって記述している。この場合、スレッドではオブジェクトを排他制御のみで共有できるというメリットがある。MPI でオブジェクトを送受信する際はオブジェクト直列化と再構築が必要であるため、オブジェクト指向プログラミングでは煩雑になり、また効率的ではない。

一般的に、C++では非組み込み型のコンストラクタとデストラクタにおいて比較的大きな処理時間を要する。このために参照渡しなどを駆使してオブジェクトの不要な生成と破棄の処理を省くことが重要である。しかし MPI のようなプロセスレベルの連携ではメモリ空間が異なるためあるプロセス内のオブジェクトが別のプロセスからは参照できず、MPI の基本型に一旦分解して(オブジェクトの直列化)MPI 経由で他プロセスへ必要なデータを渡し、渡された方のプロセスで再構築する必要がある。このため頻繁にオブジェクトのやり取りを行うことは MPI プログラミングにおいて避けるべきである。

MPI とマルチスレッドを併用する場合、少なくともスレッド間では必要に応じて排他制御を行うだけでオブジェクトが共有でき、MPI だけでプログラミングする場合に比べて、オブジェクト指向プログラミングに親和性があると言える。今回は計算タスクオブジェクトの送信で MPI を利用する際に直列化と再構築が発生しているが、その他の部位に関しては特に制約の無いプログラミングができた。

3.1.3 少ないオーバーヘッド

マルチスレッドによる並列化での速度向上が期待できる環境では、MPI のみの並列化でも効果を期待することができる。これは単純に MPI がプロセスレベルであり、プロセスはスレッドからなる為である。

しかし MPI だけで SMP 環境の演算器を活用しようとすると、同じメモリを利用する計算機内部であるに関わらず、MPI プロセス間では MPI を経由したプロセス間通信を行うことになり、オーバーヘッドが大きい。これは MPI でやり取りするデータが多いほど、スレッド化して共有した場合との差が出ると考えられる。

具体的には併用した場合のほうが MPI だけの場合と比べて速度面で有利になると予測できる。これは今回実際に検証を行っており、5 章で述べている。

3.2 欠点

3.2.1 停止性に関する煩雑さ

マルチスレッド併用は多くのマルチスレッドプログラミングと本質的に同じであるため、スレッドレベルの明示的な手動での同期処理が必要になる。それ自体は問題ではないが、スレッドプログラミングの同期処理と MPI プログラミングにおける同期処理に、それぞれ異なる停止性があることを考慮しなければならない。

例えば一般的に MPI の送受信に用いる MPI_Send や MPI_Recv はブロッキング通信である。単一スレッドで利用する場合はこのブロッキングによる停止は同期を MPI 側で行ってくれる結果であるために便利であるが、複数スレッドの場合は不要なスレッドの停止を招きかねない。例えば、これは計算の途中で出た結果を一部先行して返信する場合に MPI_Send を使うことなどで発生する。その場合、解決に非ブロッキング通信を使うことも出来るが、バッファ管理などで注意が必要になる。

他に POSIX スレッドでは同期処理に mutex と条件変数が利用できるが、MPI を手動で排他利用する際に mutex によって排他制御を行うと、単純な順番に関するミスでデッドロックに陥る可能性がある。例えばある共有資源 A に関して A に関連した mutex のロックを行ってクリティカルセクション内で処理を行い、次に MPI 通信を MPI の mutex をロックした上で使用、最後に MPI、A の順番で各々の mutex を解放する場合を考える。他のスレッドが MPI の mutex をロックして何らかの通信を行い、A をロックして作業を行って A のロックを解放、そして更に MPI 通信をして最後に MPI の mutex を解放すると、タイミング次第で A と MPI の mutex 獲得の競合状態に陥りデッドロックが発生、プログラムが停止してしまう。つまり、方や A をロックしつつ MPI の解放を待ち、方や MPI をロックしつつ A の解放を待っている状態になるため止まってしまう。これは二重に mutex をロックする場合の典型的なミスに過ぎず、MPI の利用は関係なく発生しうるが、MPI の排他制御を行う場合は考慮する必要がある。

デッドロックのような致命的なものでは無くても、不用意なブロッキングによるスレッドやプロセスの停止は、並列化における速度面でのデメリットにもなる。MPI とマルチスレッドを併用する場合には、両者のブロッキングなどでの停止状態を考慮しなければ効果的な速度向上は期待できず、そのことがソフトウェア開発をより複雑にする可能性が懸念される。

3.2.2 MPI の排他利用の必要性

原則として、MPI は複数のスレッドから利用すべきでないことが調査の結果判明した。複数スレッドから利用できる MPI 実装が存在しないというわけではないが、各 MPI 実装でのスレッドセーフ性を

考慮すると、厳密に単一スレッドから MPI を利用するべきである。そうしなければ、ある環境では正常に動作するのに、別の環境では不安定になってしまう可能性が生じる。スレッドセーフでない MPI 実装としては、広く利用されている MPICH や LAM が挙げられる。特に MPICH は SCore で利用されている MPI 実装の元でもあり、GlobusToolkit 上で MPI を利用できるようにする MPICH-G2 もそれを用いている。2004 年 12 月に MPICH2 が発表されたので、現在ではスレッドセーフ MPICH が利用可能であるが、多くの既存の MPICH や LAM による MPI 環境において、何らかのスレッド同期処理を行う必要があることには変わりはない。

この MPI の排他利用にはいくらかの方法が考えられる。一番単純なのは、MPI に対する mutex の設置による排他利用である。各スレッドでの MPI の利用前後で、手動で同期を行うことで排他利用を達成する。だがこの方法は 3.2.1 で挙げたような問題を起こす可能性がある上、プログラミングが煩雑になる。この手法は MPI 実装側で行うことで効果が発揮されると思われる。

今回このような MPI に対する排他制御は行わず、逆に MPI を用いるスレッドを限定した。詳しくは 3.3 から述べている。

3.3 全体像

今回確実に増加する開発上の手間や煩雑さを簡略にするため、同期キューを用いた自律計算スレッドを考案した。図 6 にそれを適用したソフトウェアにおける、MPI とスレッドの関係を示す。

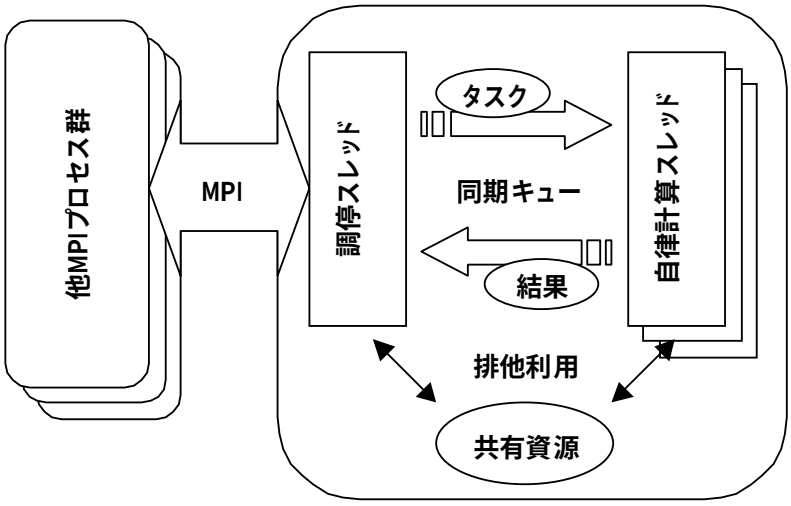

図 6　MPI とスレッドの関係

今回のプログラムでは共有資源はヒントに用いる破損ファイルのデータであり、MPI や同期キューでやり取りさせるのは 3.4.1 で挙げる計算タスクとなる。

このような形態にすることで、パラダイムの異なる MPI とスレッドの同期問題を分離して考えること

ができる。調停スレッドはスレッド性を意識することなく MPI 通信を行い、抽象化されたタスクを自律計算スレッドに同期インターフェイス経由で引き渡し、結果を受け取る。自律計算スレッドの方では同期キューからタスクがある場合はそれを受け取って処理を行い、その結果を返すが、処理すべきタスクが無い場合は停止するためプロセッサ資源を無駄にすることが無く、次のタスクの投下まで待機する。調停スレッドと自律計算スレッド間で共有されるプロセス内共有資源に関しては、通常のスレッドプログラミングで必要なものと同様のスレッドレベルの同期処理を行う。

ちなみに今回作成したソフトウェアでは、共有資源であるファイルデータは計算スレッドから読み取るのみであるため、排他制御は不要である。

3.4 同期インターフェイスと自律計算スレッド

3.2.2 で述べたとおり、スレッド間での MPI 排他利用の簡易化のため、全数検索のタスク処理はそれを専門的に行うスレッドで処理させることにした。単に処理を分けるだけではなく、計算スレッドには自律性を持たせ、同期インターフェイスのみで調停スレッドと接続することで、ソフトウェア開発を容易化している。以下、この自律計算スレッドについて詳細を述べる。

3.4.1 計算タスク

全数検索の細分化された一部処理を計算タスクと呼ぶ。並列化はこのタスク単位で行われ、MPI によって計算機レベルでタスク割り当てを行った後、スレッドが処理を行う。

一般的に、MPI のような分散メモリ型の並列環境ではメッセージ通信がボトルネックとなる。並列化による通信回数とその量は、可能な限り低くすることで計算を高速することが期待できる。このため、計算タスクの粒度は推定位置と推定内容の 2 つを決定することで最も細かくなるが、MPI での通信頻度を減らすためにある程度粒度を大きくするため、片方はスレッドで検索させることにした。

推定位置と推定内容のどちらを先に決めても結果は変わらない。2.4.2 で述べたように推定内容に基づく分割のほうがわずかであるがアドバンテージがあるように思われるが、MPI 通信量でいうと逆になる。しかし MD5 のキャッシュを利用すると計算量が推定位置によって変化するため、推定位置を固定した場合はタスクの負荷量が不均等になり、後半のタスクが早く終了するため、MPI の通信頻度が上がってしまう。

以上のことを踏まえ、計算タスクはある推定内容に対する推定位置の検索とした。

3.4.2 タスクプロセッサとしての計算スレッド

計算スレッドは極めて単純な構造をしている。以下に擬似言語で概略を示す。

```
taskProcessingThread
{
        TaskReference task;
        while (true)
        {
                task = awaitingQueue.next();        //未処理のタスクを習得
                task.process();                     //処理を実行
                resultQueue.add(task);              //結果を送り返す
        }
}
```

計算スレッドは未処理のタスクが存在すればそれを習得、処理を実行し、そして結果を送り返す、いわゆるタスクのプロセッサとして動作する。この際 task が挿すオブジェクトは計算すべき処理手順とその結果の抽象である。

今回の実装では task は基底クラスを持たず、派生クラスも想定しないため、仮想関数も存在しないシンプルな構造である。これは行う計算が全て同じであったためだが、複数の異なる内容のタスクがある場合は、例えば抽象基底クラスを導入し、上の例で言うメンバー関数 process をオーバーライドさせることで同じ手法で実現できる。ただしそれだけでは出力キューから取り出して結果を個別に処理する場合に、RTTI（実行時型情報）による型検査を行う必要が生じるかもしれない。ただし、MPI で別プロセスと送受信するだけの場合は、同様に送受信を行うインターフェイスを用意し、各実装クラスでオーバーライドすることで解決できる。MPI 送受信レベルで RTTI に準じる型情報を用いた検査は必要であろうが、これは既に C++のオブジェクトレベルでないため問題視する必要は無いだろう。

3.4.3 同期キューによるスレッドインターフェイス

タスクのやり取りには入出力のための 2 本のスレッド同期キューを用いている。先の例で挙げた awaitingQueue と resultQueue がそれぞれ入力と出力に該当し、これらのキューでタスクオブジェクトの参照を管理する。因みにタスクオブジェクト自身はスレッドが共有する形になるが、厳密に取り扱うスレッドを単一化しているので同期処理は今回行っていない。

計算スレッドのインターフェイスは、基本的にそのタスク受け渡しの同期インターフェイスのみで、

残りは読み取り専用の不変データを除いて基本的に通常のスレッドレベルの同期処理を行うことになる。

今回の同期キューは mutex と条件変数によってスレッドレベルの同期を行う。先の例での awaitingQueue.next()は条件変数によって管理されており、Queue が空の場合はなんらかの入力が生じるまで条件待機に入り、そのスレッドはブロッキングされる。3.4.4 にてその目的と詳細を解説する。

3.4.4 スレッド条件変数によるタスク待ち

単純な排他制御だけなら mutex だけで問題はないが、今回の計算スレッドの「タスクがあるときは」という自律動作性を実現するために、条件変数による待機を用いている。条件変数や同期を行わない場合は、次の擬似言語例のような構造になる。

```
while (true)
{
        while (queue.empty()) ;                 // キューが空の間ループしつつ待機
        queue.next().doSomething();             // キューから要素を取り出して処理
}
```

このようにすると、キューのサイズチェックによるポーリングが発生し、無駄に演算機の処理能力を浪費することになる。上の例では、queue が空の場合に while (queue.empty()) ;が恒常的に実行され、その間演算ユニットは空回りしている。

この手法の問題はポーリングによるリソース浪費に留まらない。一見問題ないかに見える上の例の条件評価文は、マルチスレッドで実行した場合は複数スレッドがすり抜ける可能性がある。例えば全てのスレッドが queue が空のときに while (queue.empty()) ;で空回り停止になった後、queue に 1 つの入力がなされたとする。このときにあるスレッドが queue.next()を実行して、再度 queue が空の状態になる合間に他のスレッドが while (queue.empty()) ;を抜け得てしまう。この為 queue.next()を queue.empty()状態で実行や、queue.next()への複数スレッド進行の可能性があり、危険な振る舞いを呼び起こしかねない(例えば STL コンテナでは、そのような状況で未定義の振る舞いを引き起こす可能性がある)。

これを効率的かつ安全に行いたい。そこで次の擬似言語が示すような改良を行った。

```
while (true)
{
        while (queue.empty()) queue.waitNew(); // キューが空の場合停止して待機
```

```
        queue.next().doSomething();          // キューから要素を取り出して処理
}
```

waitNew()は queue が空の間はブロッキングし、空でなくなった時点で制御を返す。またその処理はスレッドセーフに行う。この待機を実現するために条件変数を用いた。

実際にはこれらの実装はクラステンプレート SynchronizedQueue の中で行っており、next()の処理の中でこの条件のチェックと待機を同時に行わせている。以下に実際のコードの断片(上記擬似言語の next()に相当する部位)を示す。

```
1  T
2  next()
3  {
4      ::pthread_mutex_lock(&(this->deque_mutex_));
5      while (this->deque_.empty())
6          ::pthread_cond_wait(&(this->notEmpty_),
7                              &(this->deque_mutex_));
8      assert( !this->deque_.empty() );
9      T elem(this->deque_.front());
10     this->deque_.pop_front();
11     ::pthread_mutex_unlock(&(this->deque_mutex_));
12     return elem;
13 }
```

擬似言語で挙げた waitNew()に相当するブロッキング部位が 6、7 行目の pthread_cond_wait によって実現される。pthread_cond_wait は pthread_cond_signal(これは空のキューに入力が成されたときに呼び出される)によってシグナルを送られるまでの間、ブロッキングによって制御を返さず、またそのブロッキング中は安全に設定された mutex を解放している。シグナルを受け取ると再度 mutex をロックした上で制御を返すため、先のコードでの 4 行目の pthread_mutex_lock と 11 行目の pthread_mutext_unlock 間はクリティカルセクションであり続け、目標とする効率的でスレッドセーフなタスク待機状態を実現している。

3.4.5 タスクの多重割り当てとスレッドの CPU 利用率

同期インターフェイスにキューを用いる利点は、プロセス単位でタスクを多重割り当てできること

である。より短時間で計算を終えるには、全演算ユニットを常にタスク処理に割り当て、間断なくタスクを処理させることが重要であり、それが同期キューによるインターフェイスでは簡単に実現できる。

逆に、ひとつひとつスレッドにタスクを割り当てるような手法では、計算スレッドが増加するにつれてタスク割り当ての頻度や処理に要する時間を調整しなければならなくなる場合が多い。

今回の実装では最高 4 つのタスクを各 MPI プロセスに割り当てるようにした。全プロセスが 4 つの待機中のタスクを持った場合、全プロセスが一つ以上のタスクを終了するのを待ち、バリア同期を経たあとで再度割り当てを開始させている。待機タスクが 1 以上の間そのプロセス内の計算スレッドはタスクの処理中であることを意味し、今回行った実験では大半のケースでその状態が維持できており、ほぼ全ての演算器資源を活用していたと考えられる。

4. スケジューリング

MPI とマルチスレッドの併用のためスケジューリングでは、MPI レベルのスケジューリングとスレッドレベルの二つを考えることができる。今回の研究では MPI レベルのみ明示的なスケジューリングを行う。スケジューリングの対象は各 MPI プロセスであり、ある時点でどのプロセスに次のタスクを割り当てるかが今回のスケジューリングの焦点となる。

スレッドレベルのスケジューリングは基本的に行っていないが、スレッドへの優先順位の設定なども考えられる。そもそも、今回の計算スレッドは全て同一のタスク処理を行い、かつタスク間依存性が無いため、スレッドレベルスケジューリングは考慮する必要が無い。しかし、より複雑な問題では効果的な手段として利用できるかもしれない。

ちなみに pthread_cond_signal で待機状態から復帰するスレッドがどのスレッドになるか、或いは演算器以上のスレッドが存在する場合(多くの場合システムには数百以上のスレッドがある)に、どのスレッドがどのタイミングでどの程度実行されるかなど、多くのスレッドに関するスケジューリングは OS 内部のスケジューラで行われている。このため、スレッドレベルスケジューリングは安易に導入すべきではないと考える。

4.1 以降ではプロセスレベルでのスケジューリングについて詳細を述べる。

4.1 静的巡回スケジューリングの問題

恐らく最も単純なスケジューリングは、静的に順次該当するプロセス群にタスクを割り当てるものである。大抵の場合、特にクラスタなどの構成ノード性能が均一に近い状況ではそれでも問題はないかもしれない。

静的巡回スケジューリングの問題は、構成ノード性能が均一でなかったり、投下タスクの負荷が不均一であったり、或いは両方の場合に生じる。根底にある原因は静的に順序をつけてしまうことで、これがボトルネックへの依存を引き起こすことである。

例えばノード群 ABCD…に対して順次割り当てを行うとき、ノード B だけが他のノードより処理能力が低いとする。一度の巡回が終わった後に、B 以外のノードは処理を終え、B だけが処理中であるとしよう。他のノードは次のタスクを割り当てられたら処理を開始するが、B は以前割り当てられたタスクを処理し終えるまで反応できない。その場合 C への割り当てが延期されるので、全体で評価すると B の性能をボトルネックとした計算時間を要することになってしまう。

また、逆に確実に割り当て前に処理が終わるような状況では演算器が 100%活用されない。結果計算機資源に見合った性能が出ないことになる。

4.2 待機タスク数に基づく動的スケジューリング

今回、全体の計算時間を最大限短縮できるような動的スケジューリング方式を考えた。タスクを多重割り当てできることを利用し、シンプルな待機タスク数に基づくスケジューリングを行っている。

これは一定の閾値に到達するまで、待機数が少ない MPI プロセスに対して次のタスクを割り当てるだけのスケジューリングである。一定数の待機タスクが全プロセスに存在する場合、1 つ以上の結果を全プロセスが戻すまでスケジューラが待機する。

結果として、計算能力が等しい場合は静的な巡回スケジューリングと同様に働き、処理能力の不均等な場合は常に計算能力が高い計算機に対してタスクを投じることができる。そのため、性能の違いや一時的な負荷変動などで特定の処理能力の低いノードをボトルネックとしたスループットになることが回避できることが期待される。

このスケジューリングが効果を発揮するのはヘテロジーニアスな MPI 環境であるが、今回その検証は行っていない。導入した目的は、0 番の MPI プロセスが他のプロセスと比べてタスク生成とスケジューリングを行うため比較的負荷が重く、静的巡回スケジューリングではボトルネックになる可能性があったためである。実際に動作させた結果、高速化が達成されている条件では 0 番にはわずかながら少ない数のタスクが投下されており、期待した効果があったことが確認できている。

5. 性能評価

5.1 計測環境と条件

並列化による速度向上の検証には、Intel Xeon 2.8GHz の 2way SMP ノード 16 台、計 32 プロセッサからなる PC クラスタ atlantis を用いた。各ノードは 2GB のメモリを持ち、Gigabit Ethernet スイッチによって接続され、SCore によってクラスタ構成されている。図 7 に概要を示す。

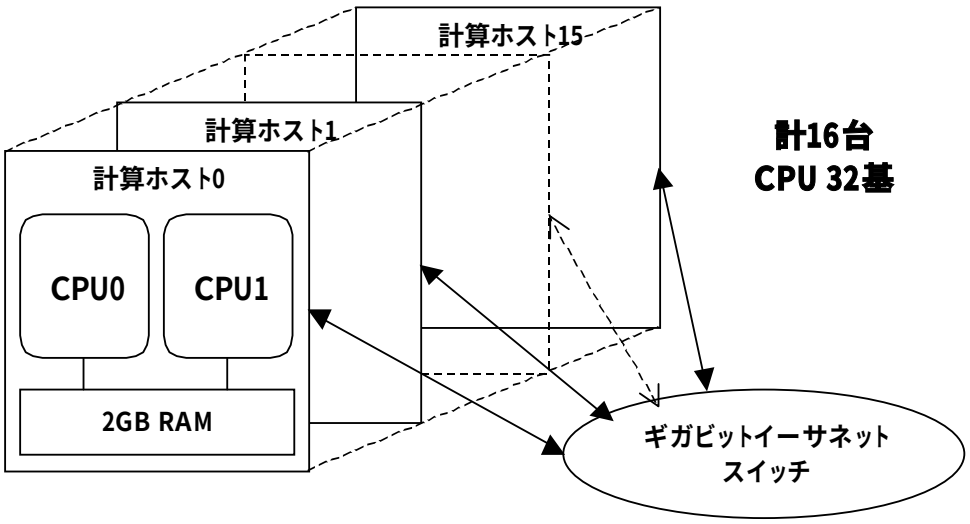

図 7　PC クラスタ atlantis 概要

今回利用した SCore の並列プログラム実行コマンドである scrun は、明示的に指定しなければ可能な限りノード数を減らすように MPI プロセスを起動した。そのままではプロセス数とスレッド数を各ノード内で調整する際に都合が悪いため、計測に当たって MPI プロセスが各ノードに幾つ動作するかはオプションにおいて”x1”によって明示している。つまり、N ノードを利用した場合は原則的にそれぞれのノードに 1 つの MPI プロセスが動作している。例外は比較のために MPI プロセスのみで処理を行わせた場合で、その際には 1 ノードに 2 つの MPI プロセスが動作している。

なお、ここで掲載する処理時間はソフトウェアが起動してから終了するまでに実際にかかった時間であり、並列化していないファイル IO 部位なども含まれている。ただしテストを終え、結果が正しく出力されることを確認した上で、不要なファイル出力などは抑制して時間計測を行った。

利用したファイルは適当に作成したものだが、2.4 でもふれた通り、内容は処理速度を含むいか

なる結果にも依存はしないし、また各同一サイズのファイル内容は全実験を通して同一である。今回行った実験全てにおいて元のファイルの修復は達成できており、また開発の初期に行った非並列版でのアプリケーションデバッグの段階では、他にも様々なファイルに対するテストを行い、正しい結果が得られることを確認している。

5.2 PC クラスタ atlantis での実行結果

先ず表 2 にファイルサイズ別に 1 台、2 台、4 台、8 台、そして 16 台全ての計算機を利用した場合の処理時間を示す。この場合計算スレッド数は 1 つのみで、MPI レベルのみでの並列化効果と考えることができる。図 8 には速度向上の様子を示す。

表 2 計算スレッド数 1 での並列化による実行速度変化 [秒]

	1 台	2 台	4 台	8 台	16 台
100 バイト	0.65	0.68	0.67	0.70	0.74
10000 バイト	71.96	36.23	18.75	9.66	5.64
20000 バイト	283.02	144.00	73.08	36.39	20.11
40000 バイト	1122.6	565.98	286.98	142.98	79.69

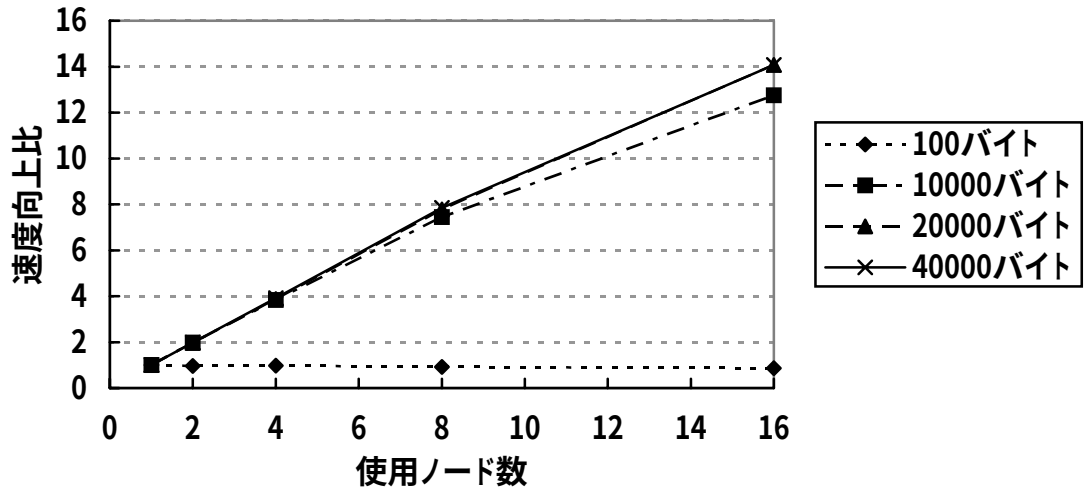

図 8 計算スレッド数 1 での並列化による速度向上比

100 バイトでは速度向上が見られないが、これは処理が小さすぎて実質 2 台の計算機しか利用されていないためである。これは今回利用した動的スケジューラの影響で、そのログで確認した(全体の待機タスク数が 2 以上になることが無かった)。タスクの発生と処理終了が短時間過ぎたために、新規タスクの生成と直前のタスクの終了がほぼ同時に起こった結果と思われる。

10000 バイトから、利用した計算機の台数にほぼ比例した速度向上が見られる。20000 バイトと40000 バイトでは速度向上比において差がほぼ無くなっており、これがこの条件での最高速度向上比であると思われる。

次に、同じ条件で計算スレッドを 2 つにした場合の結果を表 3 と図 9、そして図 10 に示す。図 10 は単一 MPI プロセス単一スレッド、つまり並列処理を行わない場合の処理時間を元にした速度向上比を表しており、今回の MPI とスレッド両方によって 32 プロセッサのクラスタで得られた並列化による速度向上比と考えることができる。

表 3 計算スレッド数 2 での並列化による実行速度変化 [秒]

	1 台	2 台	4 台	8 台	16 台
100 バイト	1.79	2.82	2.98	1.61	1.28
10000 バイト	48.02	34.18	29.11	14.72	10.106
20000 バイト	151.98	77.00	44.33	22.69	13.34
40000 バイト	562.02	333.00	169.02	88.88	44.32

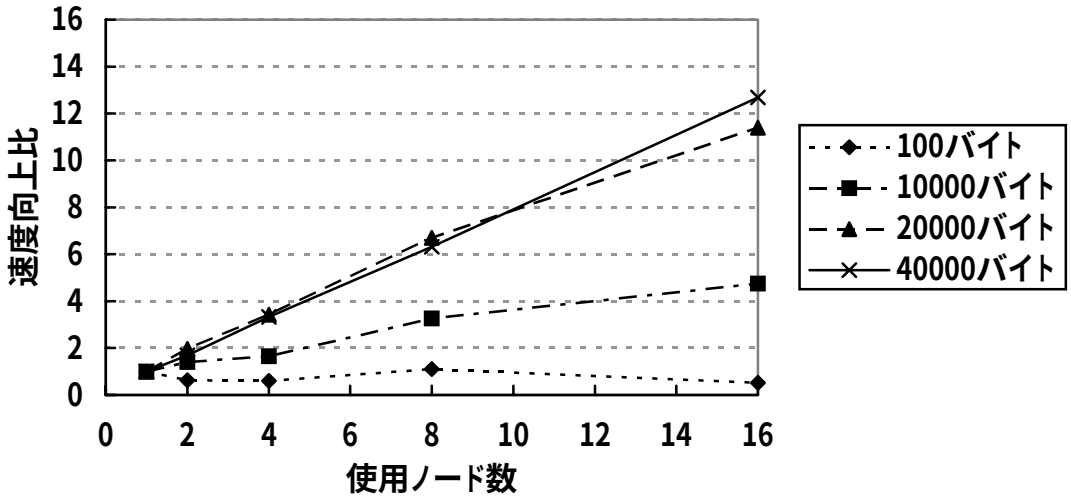

図 9 計算スレッド数 2 での並列化による速度向上比 (1 台を基準)

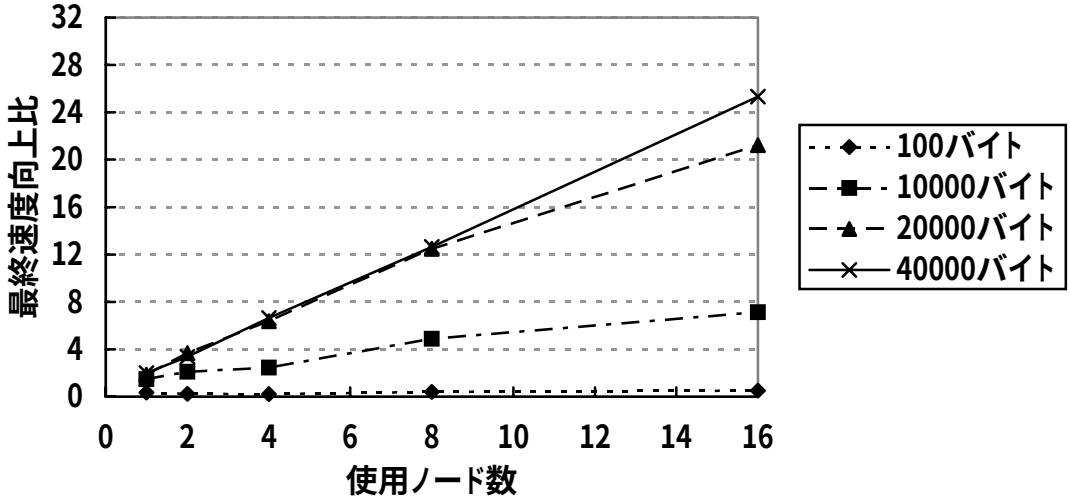

図 10 最終的な並列化による速度向上比(スレッド数 1 の 1 台を基準)

100 バイトではやはり処理量が少なすぎるためか、高速化ではなく低速化してしまっている。スケジューラのログでも確認したが、実質数台しか利用されていなかった。原因は各タスクの処理速度がタスクの生成と割り当て速度を上回っているためと推測する。

10000 バイトではスレッド数 1 の時と比べて向上比は芳しくない。これは 100 バイトと同様にタスク生成速度が十分で無いことが原因で、演算器群が活用されていないためと考えられる。

20000 バイト以上では一定の向上比を記録している。タスク処理時間が延びたことで生成が十分追いつき、各演算器が十分活用されていると思われる。40000 バイトでは最大 25.3 倍の速度向上比を確認した。

全体的に、向上比はスレッド数 1 と比べて低いが、これはスレッド同期処理が生じている点と、スレッド数 1 の場合は MPI スレッドや他のプロセスのスレッド(OS その他)に対して、より効果的に演算器能力が割り当てられている可能性が高く、その結果の現れと思われる。

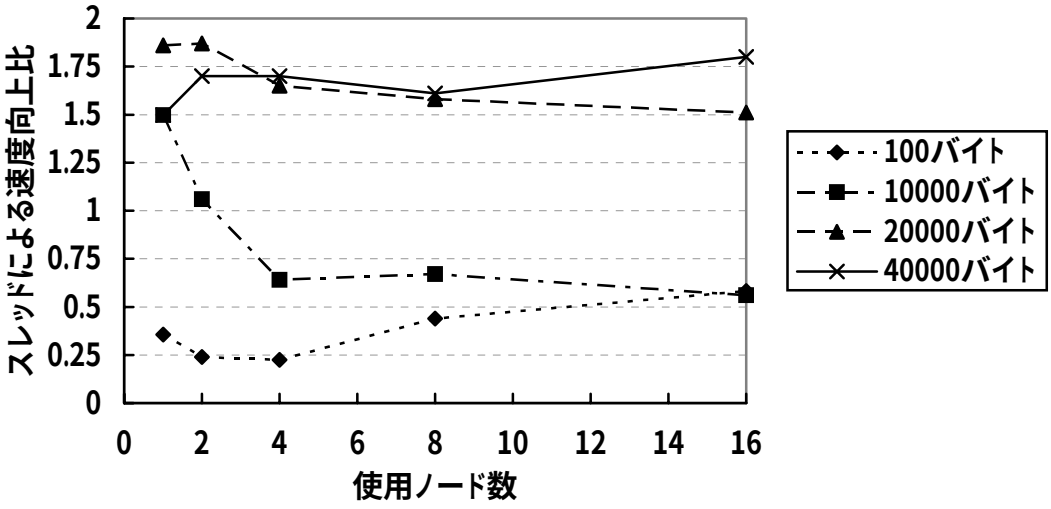

図 11 スレッド並列化による速度向上比

図 11 にはスレッド数の設定によって得られた速度向上比を示す。

これは計算のマルチスレッド化による高速化を示しており、今回使用した PC クラスタ atlantis では各ノードが 2 つの MPU を持ち、2 つのスレッドが並列動作するため、理論上、2 倍弱の高速化が期待できる。ちなみに、現実問題として OS を始めとした他のプロセスのスレッドが存在し、また各演算器がメモリバスを共有しているので、2 倍未満にしかならない。

処理量の小さい 100 バイトや 10000 バイトでは芳しくない結果であるが、20000 バイト以上では安定して高い並列化効果が得られていることがわかる。原因の一つは、先にも述べているとおり、ファイルのサイズが小さすぎるためにタスクの負荷が小さ過ぎ、その結果、速すぎるタスク処理速度にタスクの生成スピードが追いついていないためであろうが、スレッド導入によるオーバーヘッドも無視できないものであると考える。一部の条件下で、スレッド数 2 の場合に低速化していることがそのことを示していると思われる。

興味深いのは 10000 バイトの結果であるが、これは台数を増やした結果タスクの回転率が上がり、スケジューラのタスク生成と割り当ての速度が不足したため、速度向上比が低下したと考えられる。20000 バイトもその傾向がわずかに残っているが速度向上は保てており、40000 バイトではほぼ安定していることが確認できる。この傾向は実験結果と実装方式の両者から推測するに、ファイルサイズが増えて処理量が増えるにつれてより安定すると思われる。

結論として、今回の 2way の SMP 環境でのスレッドによる速度向上は 1.7 倍付近であると考えられる。

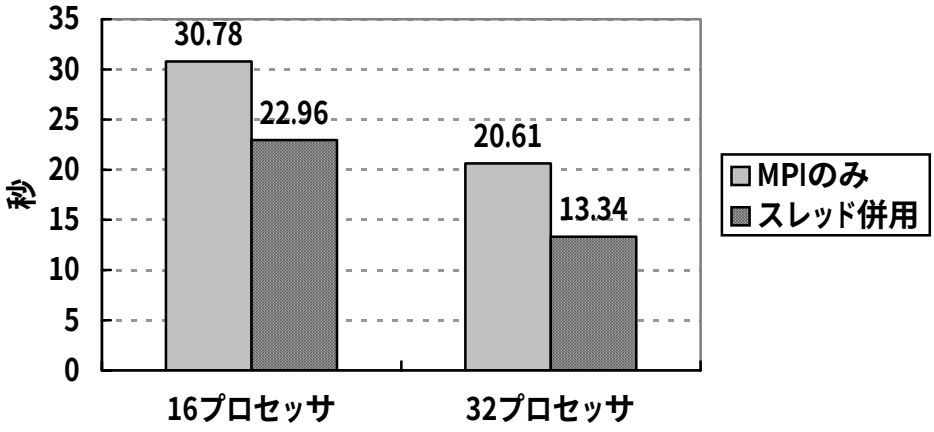

図 12 同数の演算器を MPI のみで利用した場合と併用した場合の実行時間

図 12 が最大限演算ユニットを利用するパターンについて、MPI レベルで行う場合と、計算スレッドによって調整する場合とで、かかる時間を比較した結果である。20000 バイトのファイルの処理において、同じ数の演算器を MPI プロセスレベルだけで利用する場合と、MPI とスレッドの併用で利用する場合を比較している。1 つの計算スレッドの場合は N 個のプロセッサを利用するために N 個の MPI プロセスを利用し、2 つの計算スレッドでは N/2 個の MPI プロセスを利用した。なお MPI のみとはいえ、タスクの計算は別スレッドで行われ同期インターフェイスを経由している。しかし事実上そのオーバーヘッドに要する時間は 20000 バイトの処理では総計算時間の中で極めて少なく、MPI のみでの実装と事実上、同等の比較と考えて差し支えない。

MPI だけでも、プロセッサ個数に合わせたプロセス数にすることで、SMP ノードからなるクラスタで高速化を計ることが可能なことも確認できた。しかし併用して演算器資源を利用するほうが処理時間を更に 70%程度にまで短縮しており、より高い効率を発揮していることは明らかである。

なお今回は数値面で評価していないが、メモリの使用量もスレッド併用時のほうが MD5 計算のための数値テーブルやファイルデータを共有できるため優位である。今回は高々KB 程度のオーダーの共有データ量であるが、更に大きなデータをメモリ上におく必要があり、かつそれがスレッドレベルで共有可能ならば、MPI のみの場合と比べて高い資源利用効率が期待できるであろう。

5.3 考察

MPI とマルチスレッドの併用による並列化によって、SMP 計算機からなるクラスタでプロセッサ数に見合う速度向上が得られることが確認できた。しかし、処理が小さいなどの理由で頻繁に同期処理を要する場合は必ずしも効果的でないことも判明したため、大規模問題の並列化において、共有可能資源(主にメモリ)の無駄な重複を避けつつ演算資源を利用する場合に有効な手段であると

考えられる。

反面、開発の手間は確実に増加している。定量的な評価は困難であるが簡単に触れておく。

先ず性能面での無駄を省くために、無駄なスピンロックを回避することが重要であった。スレッドの条件変数による停止のほか、MPI の利用を単一スレッドへ限定し、敢えてブロッキング通信を行わせて MPI 通信が不要な場合に停止させる等、各スレッドが動作する必要が無い場合にリソースを消費しない形で停止させている。

また、あらゆる関数呼び出しにおいて、ブロッキングの有無や処理時間の大まかな目安、そして共有資源の排他制御の必要性の有無を考慮する必要があった。特に MPI 利用において強く意識する必要があり、一般的な MPI プログラミングと比べて自由度は低かった印象がある。

だがこういった設計上の問題は自律型計算スレッドを考案する前に浮上したものである。逆に自律型計算スレッドを利用する前提で今回の MD5 によるファイル修復アプリケーションを並列化することを想定すると、その作業の量や難易度は MPI での並列化と比べて特に増加しているとは思えない。スレッドプログラミングの観点から言っても特に困難ではなかった。考案した同期インターフェイスによる自律計算スレッドの効果は少なくないと考える。

表 4 に参考までに今回作成したソフトウェアのコードに関して、MPI、スレッド、そして併用に関してどのような関連性があったか、クラス単位でまとめ、各クラスのコード行数を挙げる。

表 4　MPI とスレッド併用に関するクラス単位の関連性 (参考)

クラス名	行数	MPI 関連性	スレッド関連性	併用関連性
FixCandidate	35	×	×	×
KnownCollision	91	×	×	×
MD5	390	×	×	×
Patch	28	×	×	×
Patch8bit	42	×	×	×
Patch16bit	43	×	×	×
PowerBasedScheduler	30	△	×	×
Scheduler	74	△	×	×
Setting	171	×	×	×
SynchronizedQueue	69	×	○	○
Task	74	×	×	△
TaskDispatcher	223	○	×	○
TaskProcessor	116	×	○	○
TaskReciever	132	○	×	○

表 4 で○となる関連性は関数呼び出しなどの直接的な関係がある、もしくは非常に高い関連性があることを示す。△は直接的な関係はないものの、やや関連性があることを、×は一切関連性が

無いことを示している。任意のクラスに対して MPI とスレッドの両方への直接的な関連性が無いことは今回の設計方針であり、SynchronizedQueue と TaskProcessor の二つで自律計算スレッドは完結している。

より複雑な問題を取り扱う場合は、例えば Task を抽象基底クラスとし、処理を行う部位を純粋仮想関数にして派生クラスで実装することになる。その際に、MPI による送受信インターフェイスも導入する場合は、Task の実装を行う派生クラスにおいて MPI 関連性は生じることになる。

6. おわりに

本研究では SMP 計算機からなるクラスタでの効率的な資源利用手段として、MPI とマルチスレッドを併用した並列化を検証した。複雑化を回避するために、少々オーバーヘッドが大きいものの、スレッドレベルの同期インターフェイスによる自律計算スレッドを導入し、ソフトウェア開発時の負担を減らすことができたと考える。またスケジューリングに関しては、処理能力に注目した手法で動的に行わせた。

対象とした MD5 によるファイル修復アプリケーションは全数検索を中心にしており、並列化は全数検索の一部分の範囲を単位に行った。並列化とは別に MD5 の連続的な計算を行う場合に、計算コンテクストのキャッシュを導入することで 2 倍弱の速度向上を得ることができた。

測定の結果、16 台 32 プロセッサの PC クラスタ atlantis において 25.3 倍の速度向上比を記録した。MPI だけで処理を行った場合と比べると約 70%の処理時間となっており、また定量的な測定はしていないがメモリ使用量も少ない。しかし全体の処理量が少ない場合はスケジューラの特性が影響したため、速度向上比は減少傾向が見られた。ただし今回のアプリケーションで想定するレベルの規模では十分な速度向上は得られている。

更なる研究としては、より大規模で複雑な問題への今回の同期インターフェイスによる自律計算スレッドの適用を行い、ソフトウェア開発難易度や工数への影響の評価、及び MPI のみでの場合との性能差を検証することが挙げられる。

謝辞

先ず小柳滋教授に大いに感謝する。小柳滋教授はコンピュータシステム研究室の PC クラスタ atlantis の利用を快諾して下さった。研究の最大の焦点である MPI とスレッドの併用に関する効果の検証のためには、SMP 計算機からなるクラスタの利用は必須であった。

次に中村浩一郎と Ngyuen Viet Anh に感謝する。両氏には PC クラスタ atlantis の設定の解説や、ユーザーアカウントの登録などの開発の利便を図って頂いた。コンピュータシステム研究室の全ての構成員にも感謝したい。研究室に自由に出入りすることができ、快適に作業ができた。

高性能計算研究室の構成員にも感謝する。彼らには研究と論文執筆を通して、実に多くの助言をもらった。特に林雅樹には SCore の設定を始めとして多くのことを解説してもらった。

最後に私の指導教官である山崎勝弘教授に感謝する。山崎勝弘教授には研究を通じて多くのコメントを頂いた。PC クラスタ atlantis も山崎教授の助言があったため円滑に利用することができた。

参考文献

[1] Ronald L. Rivest: The MD5 Message-Digest Algorithm, RFC-1321, 1992.

[2] William Stallings 著、石橋 啓一郎, 福田 剛士, 三川 荘子 共訳: 暗号とネットワークセキュリティ 理論と実際, ピアソン・エデュケーション, 2001.

[3] Message Passing Interface Forum: A Message-Passing Interface Standard, 1995.

[4] Lawrence Livermore National Laboratory : POSIX Threads Programming, http://www.llnl.gov/computing/tutorials/pthreads/

[5] The Open Group: The Single UNIX Specification, Version 2, 1997.

[6] Nicolai M. Josuttis、 吉川 邦夫 訳: C++標準ライブラリ チュートリアル&リファレンス, ASCII, Addison Wesley, 2001.

[7] Ray Lischner, 株式会社クイープ 訳: C++ランゲージクイックリファレンス, オライリー・ジャパン, 2004.

[8] Ray Lischner, 株式会社クイープ 訳: C++ライブラリクイックリファレンス, オライリー・ジャパン, 2004.

[9] Barry Wilkinson, Michael Allen: Parallel Programming, Prentice Hall, 1999.

[10] Marshall Cline, Mike Girou, Greg Lomow, 金沢 典子 訳: C++ FAQ, ピアソン・エデュケーション, 2000.

[11] Neil Matthew, Richard Stones, 葛西 重夫 訳: Linux プログラミング, ソフトバンクパブリッシング, 2004.

[12] Andrei Alexandrescu, 村上 雅章 訳: Modern C++ Design, ピアソン・エデュケーション, 2001.

[13] Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides 著, 本位田 真一, 吉田 和樹 共訳: オブジェクト指向における再利用のためのデザインパターン, ソフトバンクパブリッシング, 1999.

[14] Dov Bulka, David Mayhew 著, 浜田 真理, 浜田 光之 共訳: Efficient C++, ピアソン・エデュケーション, 2000.

[15] Scott Meyers, 吉川 邦夫 訳: Effective C++, アスキー, 1998.

[16] P.J. Plauger, Meng Lee, Alexander A. Stepanov, David R. Musser 著, 長尾 高弘, 株式会社ロングテール 共訳: C++標準テンプレートライブラリ, ピアソンエデュケーション, 2001.

[17] 山本高志: 全貌を現した Linux カーネル 2.6, 第 1 章, アットマーク・アイティ, http://www.atmarkit.co.jp/flinux/special/kernel26/kernel26_01a.html, 2003.

[18] MPICH2 home page: http://www-unix.mcs.anl.gov/mpi/mpich2/

[19] LAM/MPI Parallel Computing: http://www.lam-mpi.org/

[20] Silicon Graphics, Inc.: IRIX 6.5 Man page MPI (1)