ORACLE
SPARC SERVERS

Oracleテクニカル・ホワイト・ペーパー
2013年8月

READ_ME_FIRST：多数のSPARCスレッドの使用方法

ORACLE

概要

Oracle M5-32システム内の1,536個という膨大なスレッド数に見られるように、単一システム内のスレッド数はこれまでになく増えています。これにより、きわめて大きな処理能力が提供されますが、ユーザーは、これらのすべてのリソースの最適な使用方法を知りたいと思うかもしれません。

このテクニカル・ホワイト・ペーパーでは、スレッド数の多いOracle T5およびM5サーバーをどのように展開すれば、Oracle Solaris Zones、Oracle VM Server for SPARC、およびOracle Enterprise Manager Ops Centerによる仮想化を使用してワークロードを効率的に統合および管理できるか、および単一アプリケーションのパフォーマンスをマルチスレッド化によってどのように向上させるかについて説明します。

はじめに

最近、T5およびM5サーバーの導入によってOracleシステムのレベルが向上しました。これらのシステムはそれぞれ、最大1,024個と1,536個のスレッドを備え、単一システム内でのこれまでにない処理能力を提供します。

顧客がこれらのすべてのリソースを使用して単一ワークロードまたは1つの単一アプリケーションを実行する場合もありますが、状況によっては、これらのサーバーを使用して複数のアプリケーション・ワークロードを統合することが予測されます。その際、これらのアプリケーションの一部またはすべてはマルチスレッド化できます。

オラクルでは、両方のニーズに対処するソリューションをいくつか提供しています。それが、このホワイト・ペーパーのトピックです。

最初の部分では、マルチスレッド化されたシステムのためのプログラミングに関連した重要な概念について簡単に説明します。次の部分では、8個のハードウェア・スレッドをサポートし、T4、T5、およびM5のすべてのシステムの中核をなすSPARC S3コアについて説明します。

次に、SPARC S3コアを使用することにより、システムのサイズに応じて拡大縮小できる高度に最適化されたキャッシュ・コヒーレンシのあるインターコネクトによって共有メモリ・サーバーを構築する方法を示します。また、T4-4、T5-8、およびM5-32のトポロジについても説明します。

単一アプリケーションのパフォーマンスは、マルチスレッド化によって向上させることができます。このトピックはきわめて重要であるため、Oracle Studioコンパイラおよび分析ツールの使用に重点を置いて、非常に広範囲にわたって説明されています。そこでは、この実現方法を示すために2つの異なるパラレル・プログラミング・モデルを使用します。OpenMPパラレル・プログラミング・モデルの概要の後、アプリケーションをパラレル化するためのPOSIXスレッドの方法が示されます。

マルチスレッド化の目的はパフォーマンスとスケーラビリティですが、これはまさに、"言うは易く行うは難し"です。スケーラブルなアプリケーションの記述方法に関する推奨事項を示しているのは、そのためです。これらのルールに従うことは、高いパラレル・パフォーマンスに向けた大きなステップです。

上で説明したように、これらの新しいシステムは、顧客が1つのこのようなシステムを使用してワークロードを統合および管理することを検討するに足る非常に高い処理能力を備えています。

オラクルは、特定の統合ニーズに適合するさまざまなソリューションを提供しています。Oracle Solaris ZonesとOracle VM Server for SPARCは、これに対処するための2つの異なる仮想化テクノロジーです。Oracle Enterprise Manager Ops Centerは、物理および仮想サーバーの完全なライフ・サイクル管理のための、費用効果に優れた統合されたソリューションを提供します。これらのすべてが詳細に説明されています。

パフォーマンス・アナリストによって見落とされる場合があるその他の重要なトピックの1つとして、オペレーティング・システムの影響があります。これらの非常にスレッド数が多いシステムをサポートするには、真剣で、継続的なエンジニアリング作業が必要です。これが、最後のセクションのトピックです。

このホワイト・ペーパーの最後には、使用されている用語を説明した用語集が付属しています。

マシンのパラレル化の活用

はじめに

大規模なマルチスレッド・サーバーの時代が到来しました。これは、最新のプロセッサ・テクノロジーがムーアの法則の追跡に成功しているためです。[1]オラクルが最近発表したSPARC T5およびM5プロセッサ[2、3]（SPARC T4プロセッサで最初に登場した、大きな成功を収めたS3コアに基づく）を搭載したSolarisベースのサーバーは、処理用に1,000個の高パフォーマンス・ハードウェア・スレッド（HWT）を超えるスケーラビリティを備えています。プロセッサあたりのコア数およびサーバーあたりのプロセッサ数をさらに増やすというこの傾向は、今後も続くことが予測されます。現在の課題は、これらのシステムの増加する処理能力を活用するための方法を見つけることです。このあり余る処理能力のために、SPARCの顧客は、これらのすべてのHWTの効率的な使用に関する何らかのガイダンスを必要としています。このドキュメントは、そのガイダンスを提供するものです。

パラレルでの思考およびパラレル化の利点

すべてのプログラムの実行を高速化し、全体的なシステム・パフォーマンスを向上させるための大きな課題は、独立したソフトウェア・コンポーネントを識別し、それらを複数のリソース上で"パラレルで"（つまり、同時に）実行することです。パラレル化の活用は、コンピュータ・システムのほぼすべての領域で不可欠です。プロセッサ・レベルでだけでなく、I/Oやデータ通信もオーバラップした同時動作のメリットが得られます。そのため、アプリケーション開発者とITアーキテクトの両方が、担当するアプリケーションやシステム構成のパフォーマンス、スケーラビリティ、および信頼性を向上させるためにどこにパラレル化を使用できるかを理解する必要があります。

これは当たり前のように思われるかもしれませんが、プログラマとシステム・アーキテクトの両方が、あらゆる形式でパラレル化を活用する可能性を見落とす場合が多いことを歴史が示しています。クラウド・コンピューティングおよびビッグ・データ実装の可能性を実現するには、スケーラブルなパラレル化を使用する必要があります。

さらに、最新のマルチプロセッサ・マルチコア・サーバーは非常に高い処理能力を集中的に備えており、これらのシステムでは、財務コスト、サーバー台数、床面積、ソフトウェア・ライセンス、電気、冷却、および保守作業の大幅な削減を可能にする仮想化テクノロジーの使用を通して、サーバー統合のための卓越したプラットフォームが提供されます。特に、オラクルの最新のSPARCサーバーは、多数のハイパーバイザーベースのVMドメインに容易に再分割できるだけでなく、Solarisの組込みのOS仮想化テクノロジーであるゾーンも使用できます。

オラクルのアプローチ：製品およびテクノロジー

オラクルのSPARCプロセッサ、オペレーティング・システム、アプリケーション・ソフトウェア、エンジニアド・システム（Exadata、Exalogic、SPARC SuperCluster、Big Data Applianceなど）[4]、およびOptimized Solutions [5]はすべて、オラクルの製品ライン全体にわたるパラレル化の概念の採用と統合の主な例です。

そのほんのいくつかの例として、次のものがあります。

- コアあたり8つのスレッドをサポートするS3マルチコア・プロセッサ設計
- 最大1,536個のハードウェア・スレッドを備えたT4、T5、およびM5の共有メモリ・サーバー
- Oracle Real Application Clusters（Oracle RAC）
- 広範囲にマルチスレッド化されたOracle Solarisオペレーティング・システム・カーネル
- Solaris 11に統合されたロードバランサ
- ZFS（RAID-Z、ZIL用の個別のディスク）
- SPARCおよびx86仮想化
- マルチスレッドの完全サポートを含むSolaris Studioコンパイラおよびツール
- マルチスレッドの完全サポートを含むDTrace [6、7]システム・プロファイリング・ツール

これらの製品およびテクノロジーには、次のような、全体的なシステム・パフォーマンスを向上させるための複数の方法が組み込まれています。

- Solarisによるパラレル化およびキャッシュの使用の向上
- アプリケーション開発者のためのマルチスレッドAPIの提供およびコンパイラの最適化
- インテリジェント・キャッシュ、クロック速度の高速化、および命令のパラレル化を通したプロセッサの高速化
- DTrace [6、7]やOracle Solaris Studio [8]などの、パラレル化の機会を検出するツールの提供

このドキュメントは、開発者、システム・アーキテクト、パフォーマンス・アナリスト、およびシステム管理者がパラレル化を活用するための機会を認識したり、オラクルの広範囲にスレッド化されたシステムをサーバーの統合用に構成する方法を説明したりするのに役立ちます。

そのために、基盤となるパラレル化のさまざまな概念に関する情報を提供するとともに、各種のユースケースに対処するためにオラクルの製品およびテクノロジーをどのように使用できるかについても説明します。特に、最近導入されたSPARCマルチコア・サーバーに重点が置かれています。

SPARC T4、T5、およびM5の共有メモリ・システム

ここでは、SPARC S3コアの他、T4、T5、およびM5システムの概要について説明します。これらのすべてのサーバーは基本的な構成要素として同じS3コアを使用し、システム・ファミリ全体にわたるバイナリ互換性や、以前のSPARCシステムとのバイナリ互換性を保証しています。ここでは、アウトオブオーダー実行や命令レベルのパラレル化などの、おもな機能のごく一部だけが強調されていることに注意してください。S3コアおよびS3コアが使用されているSPARCサーバーについて詳しくは、[9、10、11、12]を参照してください。

アウト・オブ・オーダー実行

従来の"順序どおりの"プロセッサ設計では、各命令は、コード内に現れるのとまったく同じ順序で命令キャッシュからフェッチされ、デコードおよび実行されます。このアプローチのデメリットは、完了するために多くのクロック・サイクルが必要な除算などの"コストの高い"命令によって、以降の命令の実行が（それらの命令が独立していたとしても）ブロックされる点です。これにより、その命令に依存しない命令で高コストな命令をバイパスする方法が存在すれば回避できたであろう遅延（つまり、"ストール"）が実行フローで簡単に発生します。

アウト・オブ・オーダー（"OOO(out-of-order)"）アーキテクチャでは、まさにそれを実行します。OOO実行では、後で発行された命令が、前に発行された命令が完了する前に実行されることがプロセッサによって許可されます。これが許可されるのは、各命令が独立している限り、結果として得られる値が正しいためです。OOO実行では、このような実行時の依存性が自動的に追跡され、プロセッサ内で透過的に処理されます。

これを単純な例を使用して示します。下のリストは、疑似命令を使用した浮動小数点命令シーケンスです。"r<n>"を使用してレジスタ<n>を示しています。下の各命令は、最初の命令から開始され、上から下に実行されます。

```
fadd   r1,r2,r3      #      r3 =   r1+r2
fsub   r1,r2,r4      #      r4 =   r1-r2
load   @a,r1  #      load   the value at memory address "a" into register r1
load   @b,r2  #      load   the value at memory address "b" into register r2
fdiv   r3,r4,r5      #      r5 =   r3/r4 = (r1+r2)/(r1-r2)
fmul   r1,r2,r6      #      r6 =   r1*r2
```

上の命令スケジュールは、独立した命令が互いをブロックするため、最適ではありません。これらのストールの原因になっている命令は、この場合、コストも（潜在的に）高くなっています。

イン・オーダーアーキテクチャの場合、除算を実行している"fdiv"命令は、2つのメモリ・ロード命令の完了を、それらの値を必要としていないにもかかわらず待つ必要があります。このデータが近くにキャッシュされている場合でも、その取得には数サイクルかかります。これにより、パイプライン・ストールが発生します。"fmul"命令は、その入力値が"fdiv"命令に依存していないにもかかわらず、レイテンシの長いその命令の完了を待つ必要があります。これらの遅延が真の依存性のためではなく、命令スケジュールのために発生していることに注意してください。

また、ロード命令での"r1"と"r2"の使用により、"fadd"命令と"fsub"命令でも"r1"と"r2"が使用されているため、これらの命令に対する人為的な依存も発生しています。しかし、この重複した使用は必要ありません。正確性に影響を与えることなく、ロード命令の結果を"r7"と"r8"に移すことができます。したがって、これは正しいけれども、最適ではないスケジュールであり、削減または回避できる可能性のあるストールを含んでいます。

OOOアーキテクチャでは、実行パターンはより動的であり、これらの2つのデメリットを解消します。"fdiv"命令が待つ必要があるのは、"fadd"命令と"fsub"命令の完了だけです。同様に、"fmul"命令は"fdiv"命令には依存せず、2つのメモリ・ロードの結果が使用可能になるとすぐに開始されます。

OOO実行を実装するときは、別の機能が必要なことに注意してください。2つのメモリ・ロード命令を"fadd"命令と"fsub"命令が完了する前に発行できるようにするには、レジスタ"r1"と"r2"の複数の使用について考慮する必要があります。先ほど説明したように、この再利用は正確性のためには不要であり、それによって実行速度が低下しないようにする必要があります。プロセッサの"レジスタ名の変更"の部分がこの状況を処理します。

OOO実行は多くのアプリケーションのパフォーマンスを向上させるだけでなく、パフォーマンスをより滑らかなものにします。命令がスケジュールされている順序の正確な詳細に対する依存性が低くなっているため、最適ではない命令のスケジューリングに関する耐性が向上しています。別の利点として、キャッシュ内に存在するかどうか分からないメモリ・ロード命令などの可変の待機時間を持つ命令を、関連のない作業とオーバラップできることが挙げられます。

OOO実行のおかげで、これらの依存性はハードウェアで処理され、命令のストールは、これらの命令に必要なサイクル数に応じて削減されるか、場合によっては解消されることもあります。

SPARC S3コア

SPARC S3コア[9、10、11、12]には、OOO実行が実装されています。これは、この機能を実装する最初のSPARCプロセッサであり、T4、T5、およびM5プロセッサで使用されています。図1は、このコアのブロック図を示しています。

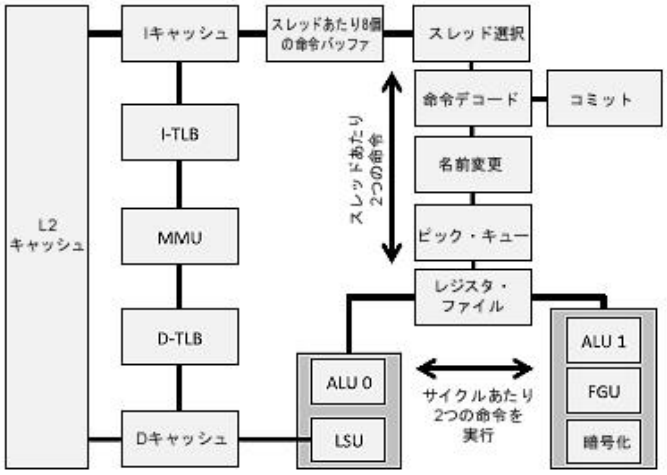

図1：SPARC S3コアのブロック図

これはコアだけであることに注意してください。T4、T5、およびM5プロセッサは複数のコアを含んでいるだけでなく、メモリ・コントローラ、I/O、コヒーレンシ・インタフェースなどの追加のオンチップ機能も備えています。

S3コアでは、OOO実行が命令レベルのパラレル化（ILP）と組み合わされています。ハードウェアは、アプリケーションに対して透過的に、コア内のさまざまな実行パイプラインに最大2つの命令を発行します。たとえば、2つの整数加算命令、または整数加算命令とメモリ・ロード命令を同時に発行できます。

このメカニズムは、"2ウェイ・スーパースカラ"とも呼ばれます。一見すると、さらに多くの命令をパラレルで発行できるようにしたくなりますが、実環境の多くのアプリケーションでは、命令レベルのパラレル化がごく控えめなレベルでしか示されないことが分かっています。クロック・サイクルあたり2つの命令を発行する機能をサポートすることは、多様なアプリケーションでのシングルスレッド・パフォーマンスを高速化することが実証されています。

その他のプロセッサ機能

もう1つの重要なパフォーマンス機能として、S3コア内での8個のハードウェア・スレッド（HWT、"ストランド"とも呼ばれます）の非常に効率的な使用があります。

ハードウェア・スレッドの基本的な概念は、実行中のアプリケーションでパイプライン内の遅延が発生した場合にプロセッサ・サイクルの浪費を避けることです。この遅延は、ロング・レイテンシ命令やキャッシュ・ミスで発生することがあります。以前の設計では、遅延が解決されるまで実行を続行できないため、このようなすべてのイベントのためにコアはサイクルを浪費することになります。

S3コアでは、アイドル・サイクルは他のハードウェア・スレッドによって自動的に使用されます。これは、アプリケーション、Solaris、およびユーザーに対して透過的に行われます。

このスレッド選択メカニズムは非常にきめ細かく、ハードウェアによって実行されます。プロセッサ・サイクルごとに、8個の候補スレッドから1つが選択されます。実行の準備ができているスレッドのみが対象になります。この選択は、各スレッドによるパイプラインの公平な共有を保証するとともに、スレッドの枯渇を回避する、変更されたLRU（Least Recently Used）アルゴリズムに基づいて行われます。

これを非常に効率的に実行するために、十分な注意が払われました。たとえば、各スレッドには独自の命令バッファがあります。選択されたスレッドの命令はこのバッファからフェッチされ、コアのOOOの部分に送信されます。

S3コアでのハードウェア・スレッディングの実装によって、システム・スループットとパラレル・アプリケーションのパフォーマンスの向上を目的とした、使用可能なサイクルの効率的で、適応性があり、かつきめ細かな使用が可能になります。

これらの機能の組み合わせにより、非常に強力で、柔軟なコアが生成されます。次のセクションでは、このコアがSPARC T4、T5、およびM5の共有メモリ・システム内の基本構成要素としてどのように使用されているかについて説明します。

SPARC T4、T5、およびM5システム：点の接続

今見てきたように、コア・レベルにはすでにパラレル化の2つのレベルが存在します。もっとも低いレベルでは、2ウェイ・スーパースカラ機能によって、最大2つの命令が同時に発行されます。これが、コアあたり8個のハードウェア・スレッドのハードウェア・サポートと組み合わされます。

ここでは、SPARC T4、T5、およびM5システムの特定の機能がどのように使用されて、パラレル化の*3番目*のレベルが追加されるかについて説明します。

先に説明したように、これらのシステムはすべて、コアあたり8個のハードウェア・スレッドを含む同じSPARC S3コアを使用しています。ただし、T4、T5、およびM5*プロセッサ*には、コア・クロック速度、1つのプロセッサ内のコアの数、およびL3キャッシュのサイズに関して違いがあります。[1]これらのプロセッサは、T4、T5、およびM5*システム*で使用されています。システムの名前はプロセッサ名に基づいており、サポートされているプロセッサ/ソケットの数がそれに追加されます。たとえば、T5-4システムには、4つのT5プロセッサが含まれています。

表1は、さまざまなプロセッサ機能の概要を示しています。

表1：SPARC T4、T5、およびM5システムのプロセッサ・レベルのさまざまな機能

システム	プロセッサの最大数	コア/プロセッサの数	HWT/コア	HWTの合計	クロック速度（MHZ）	L3キャッシュ（MB）
T4	4	8	8	256	2998	4
T5	8	16	8	1024	3600	8
M5	32	6	8	1536	3600	48

3つのすべてのシステムで同じコアが使用されているため、コア・レベルで実行された最適化はすべて、3つのすべてのシステムに役立ちます。

すべてのシステムが、システム内のすべてのプロセッサ、メモリ、およびI/Oを接続している、キャッシュ・コヒーレンシのあるネットワークを使用します。ネットワークのトポロジには違いが存在しますが、後で説明するように、ネットワークの詳細がアプリケーション開発やパフォーマンス・チューニングに影響することはありません。

T4およびT5システムは、シングル・ホップの直接接続ネットワークを使用しています。T4-4とT5-8で使用されているインターコネクト・ネットワークの図を、それぞれ図2と図3に示します。

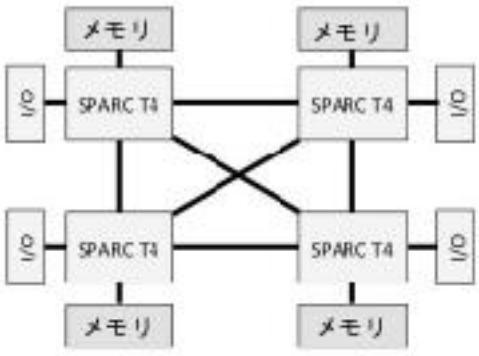

図2：SPARC T4-4システムのインターコネクト・ネットワーク

[11] このホワイト・ペーパーの範囲を超えるその他の低レベルの違いがいくつか存在します。

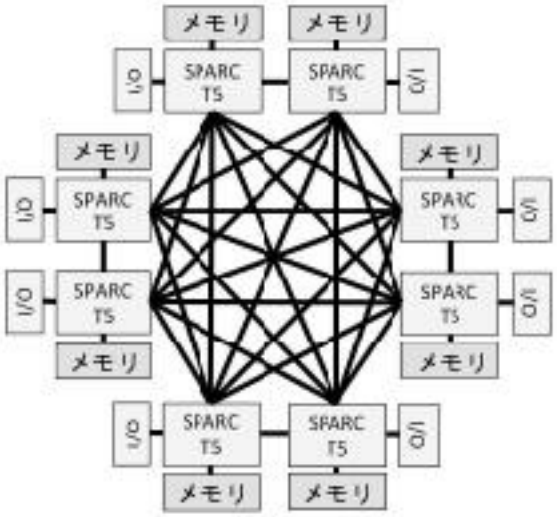

図3：SPARC T5-8システムのインターコネクト・ネットワーク

図2と図3に示すように、プロセッサ、メモリ、およびI/Oはサーバー全体にわたって物理的に分散していますが、キャッシュ・コヒーレンシ回路のおかげで、サーバーは単一システム・イメージ・アーキテクチャを備えています。つまり、これはクラスタとは異なります。システム全体に対して1つのアドレス空間が存在し、各メモリ・ロケーションやI/Oデバイスにはシステム内のどこからでもアクセスできます。

クラスタとは異なり、メモリやI/Oなどのリソースが断片化されないため、これは非常に有利な特性です。アプリケーションには、これらのリソースの使用に関する制限はありません。たとえば、単一アプリケーションが、1つのコアしか使用していない場合でもすべての共有メモリを使用できる一方で、パラレル・アプリケーションも、すべてのメモリとすべてのコアを容易に使用できます。I/Oにも同じ透過性が適用されます。

この種のアーキテクチャはまた、スケーラブルなメモリ帯域幅も提供します。各プロセッサは共有メモリの一部に接続し、そのメモリへの独自の直接接続が与えられます。そのため、プロセッサを追加すると、メモリ帯域幅も追加されます。これは、帯域幅が固定されている、古いバス・ベースの設計に対する大幅な改善です。この場合は、プロセッサを追加すると、各プロセッサで使用可能なメモリ帯域幅が削減されます。

任意のプロセッサ上の各コアが、他のプロセッサに接続されているすべてのメモリに透過的にアクセスできます。唯一の違いは、そのコアが物理的に存在しているプロセッサに接続された"ローカル"メモリへのアクセスに比べて、"リモート"メモリへのアクセスには時間がかかる点です。

このメモリ・アクセス時間の違いは、"NUMA"（Non-Uniform Memory Access）と呼ばれます。システムはキャッシュ・コヒーレンシもサポートしているため、"cc-NUMA"アーキテクチャを備えていると見なされます。

T5のメモリ・アーキテクチャに似た、M5-32システムのインターコネクト・ネットワークを図4に示します。

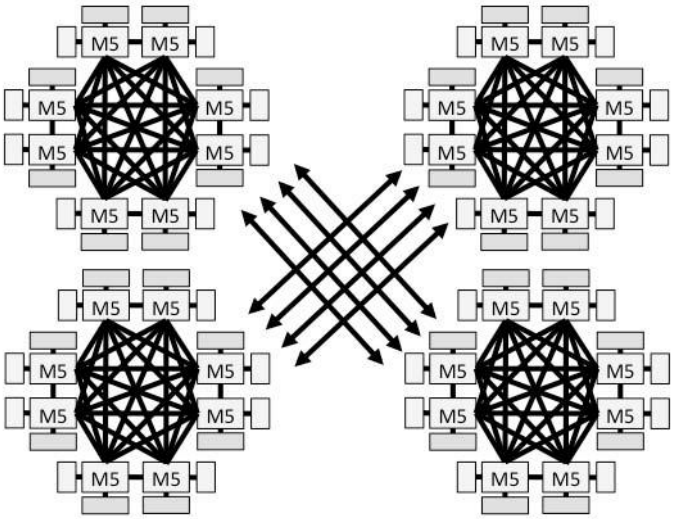

図4：SPARC M5-32システムのインターコネクト・ネットワーク

アプリケーション開発および展開の観点から見ると、cc-NUMAアーキテクチャの使用は非常に簡単です。使用するコアの選択にも、メモリ・アクセスにも制限はありません。アプリケーションは、データの読取りまたは書込みのために、すべてのメモリ・ロケーションに透過的にアクセスできます。

これまでに説明した考慮事項に加えて、「データの局所性のための最適化」では、自動的またはプログラム的にローカル・メモリを優先的に使用する機能について説明します。

マルチコア・システムのためのアプリケーションの開発

ここでは、マルチコア・ベースのハードウェアが提供する共有メモリのパラレル化を利用する方法について説明します。T4、T5、およびM5ベースのサーバーはすべてこれに含まれるため、ここで説明されているすべての内容が、これらのシステムにただちに適用されます。

パラレル化の特定

"方法"について詳細に見ていく前に、"場所"について少し時間を取って説明したいと思います。残念ながら、アプリケーション内のパラレル化の識別についての特効薬はありません。

ここでは、アプリケーションとその機能に関する知識が大きな助けになります。[2]

では、実際にはどうすればよいでしょうか。

[2] Oracle Solaris Studioコンパイラは、コンパイラがループ・レベルで依存分析を実行する-xautoparオプションを通して自動パラレル化をサポートしています。ループの繰り返しに関連した作業が独立していることが分かると、コンパイラは、ユーザーのためにパラレル・コードを自動的に生成します。

理想的なのは、新しいアプリケーションを開発する余裕がある場合です。その場合は、最初からパラレル化を考慮することを強くお勧めします。まだパラレル化を実装する必要はありませんが、後でのパラレル化が容易になるようにコードを記述できます。

新しく開発されたプログラムの場合は、パラレル化の機会を識別することが必要です。オンライン・バンキング・システム、科学的なシミュレーション、電子商取引アプリケーション、デバイス・ドライバ、またはその他の任意のアプリケーションのいずれであっても、パラレル化のレベルを理解することが重要です。

理想的には、事前に理解しておくことで、直接または後の段階でパラレル化を促進するための何らかのソフトウェア設計の選択が左右される可能性があります。

たとえば、アプリケーション内のかなり高いレベルにパラレル化が存在する場合は、ある関数が呼び出す独立した部分の実装を検討できます。これらの関数が極端に単純でない限り、このような関数呼出しによって発生するオーバーヘッドを心配する必要はありません。これらの関数を単純にパラレルで実行するだけで済むため、このメリットは注目に値します。たとえば、OpenMPを使用している場合、同時に実行される"n"個の関数に対して必要なのはn+1個のプラグマだけです。

もう1つの設計の選択として考えられるのは、グローバル・データを使用する場所の慎重な決定です。一般に、ローカライズされたデータをできるだけ使用し、実際に必要でない限りグローバル・データを使用しないようにすることは良いパフォーマンスの原則ですが、パラレル・アプリケーションでは、一般に変更されるグローバル・データには特別な注意が必要です。

しかし、多くの場合、アプリケーションを最初から記述している余裕はありません。ただ、心配する必要はありません。この場合でも、効率的なパラレル・アプリケーションを開発できます。

最初の重要なステップとして、Oracle Solaris Studioパフォーマンス・アナライザなどのプロファイリング・ツールを使用して、実行時間が費やされている場所を見つけます。多くの場合はこれにより、その場所はアプリケーションの中ではコストが高くないと予測されていた部分であることが判明するため、これは一般に重要なステップです。

時間のほとんどが費やされている場所が判明したら、次のおもな問題は、この部分で実行されている作業が特定のレベルの独立性を持っているかどうかという点です。これに該当する場合は、パラレル実行の機会が存在します。

たとえば、任意の順序で実行できる関数呼出しがいくつか存在する場合があります。その場合、適切にロックせずに共有データ構造を変更するといった副作用がないとすると、これらの関数呼出しは同時に実行することもできます。おそらくforループが存在し、さまざまな繰り返しにわたって実行される作業は独立しています。その場合は、そのループをパラレルで実行できます。

もちろん、どこかに問題点がないかどうか十分注意する必要があります。ターゲットのコード片がきわめて単純なものでない限り、より詳細な分析が必要になり、その場合はアプリケーションの知識が大きな助けになる可能性があります。

もっとも長い時間がかかる部分がパラレル化されたら、同様の方法で、2番目にコストの高い部分に取り組む必要があります。このようにして、パフォーマンス・プロファイルを処理しながら、アプリケーションを徐々にパラレル化していきます。

このプロセスをすぐにやめることはせず、常にアムダールの法則[18]を念頭に置いてください。

共有メモリのためのパラレル・プログラミング・モデル

マルチコア・システムを対象としたアプリケーション内のパラレル化を表すために使用できる共有メモリ・プログラミング・モデルが多数存在します。[3]

従来から開発者は、C/C++でのPOSIXスレッド（略して"Pthread"）[13]や、Javaアプリケーションでの Javaスレッドなどの明示的なスレッディング・モデルを使用してきました。このようなモデルでは、共有メモリ環境でパラレル化を実装するための強力ではあるが、かなり低レベルの関数については比較的少数しか提供されていません。

これらのモデルとこのホワイト・ペーパーの範囲との概念的な類似性を考慮し、明示的なスレッディング・モデルの例としてはPthreadだけについて説明することを選択しましたが、ここで説明されているトピックの多くはその他のこのようなモデルにも適用されます。

この領域の反対側には、OpenMP [14、15]が存在します。大まかに言うと、開発者はプラグマを使用して、アプリケーション内のパラレル化を表します。次に、コンパイラがこれを適切な基盤インフラストラクチャに変換して、パラレル・アプリケーションを作成します。

このセクションの残りの部分では、OpenMPとPthreadの両方について説明します。

OpenMPパラレル・プログラミング・モデル

最初のOpenMP 1.0の仕様は、1997年にリリースされました。それ以降大幅に進化し、2013年7月にリリースされた4.0の仕様では、さらに大きな前進を遂げています。

では、OpenMPとはどのようなものでしょうか。

それは、C、C++、およびFortranに使用可能な共有メモリのパラレル・プログラミング・モデルです。ユーザーは、ソース内にディレクティブを追加することによってパラレル化を指定します。C/C++では、これらはプラグマです。一方、Fortranでは、固有の言語コメント構文が使用されます。どちらの場合も、その利点は移植性です。OpenMPをサポートしていないコンパイラでは、そのプラグマや特殊なコメント行が単純に無視されます。

パラレル・バージョンの"Hello World"の単純な例を次に示します。OpenMPに固有のソース・コードの変更が青で色分けされています。

[3] プログラムを"パラレル"または"マルチスレッド"と呼ぶのは、同時に実行できる複数の独立した命令ストリームが存在する場合です。これらのストリームを"スレッド"として大まかに定義することもできます。

```
#include <stdio.h>                                                          (1)
#include <stdlib.h>                                                         (2)

#ifdef _OPENMP                                                              (3)
#include <omp.h>                                                            (4)
#endif                                                                      (5)

int main()                                                                  (6)
{                                                                           (7)
 #pragma omp parallel                                                       (8)
 {                                                                          (9)
  printf ("Hello OpenMP World, I am thread %d\n",omp_get_thread_num());     (10)
 } // End of parallel region                                               (11)
}                                                                           (12)
```

このように単純なコードでも、どのOpenMPアプリケーションにもあるおもな機能のいくつかをすでに備えています。

最初の新しい要素は、行3～5にあります。ここでは、すべてのOpenMPランタイム関数を定義したOpenMP固有のファイル（"omp.h"）をインクルードしています。このような関数が使用されない場合、このファイルをインクルードする必要はありません。このインクルードが、_OPENMPが定義された条件の下で行われていることに注意してください。これは、アプリケーションがOpenMP用にコンパイルされる場合であることが保証されています。そのため、このファイルをインクルードしない場合は、OpenMPを有効にせずにプログラムをコンパイルすることもできます。

行8～11は、パラレル・コードの中心部分を形成しています。ここでは、"パラレル・リージョン"と呼ばれるものが定義されます。これは、OpenMPにおける主要な構文です。それは、#pragma omp parallelキーワードによって定義されます。すべてのスレッドがパラレル・リージョン内のコードを実行します。パラレル・リージョンの外では、メインの"マスター"スレッドがコードのシリアル部分を実行します。

この場合、パラレル・リージョンは行9にある開始の中かっこで始まり、行11にある終了の中かっこで終わります。必須ではありませんが、パラレル化の終了を強調するために、この行にコメント文字列を付加するようにしています。

行10にある"printf"文はパラレル・リージョンの一部であるため、各スレッドがこの文を実行します。この行を出力したスレッドを識別するために、出力に一意のスレッド番号を含めています。この値は、omp_get_thread_num()ランタイム関数によって返されます。

これでパラレル・コードが完成したので、次にこれをコンパイルする必要があります。これは、特定のコンパイラ・オプション（Oracle Solaris Studio C、C++、およびFortranコンパイラでは-xopenmp）を使用して容易に達成されます。このプログラムをファイル"hello.omp.c"に格納したと仮定すると、単純に次のようにコンパイルします。

```
$ cc -o hello -fast -xopenmp hello.omp.c
```

-fastオプションは、多くの高度なシリアル最適化をアクティブ化します。先ほど説明したように、-xopenmpオプションは、OracleコンパイラでのOpenMPサポートをアクティブ化します。

実際にコンパイラでどのような処理が行われているかについて、少し時間を取って説明します。-xopenmpオプションにより、コンパイラは適切な構文を持つプラグマ（Fortranではディレクティブ）を認識します。それが、C/C++では#pragma omp <keyword(s)>です。

行8に示す単純なプラグマが、実行時にパラレルで実行されるコードを生成するためのかなり複雑なライブラリ呼出し構造に変換されます。スレッドの作成、管理、および実行はすべて、ユーザーに対して透過的に処理されます。開発者が行う必要があるのは、プラグマを使用して、パラレル化が存在する場所および作業を分散する方法をコンパイラに通知することだけです。

後で説明するように、OpenMPではこれだけでなく、さらに豊富な機能が提供されますが、一般的な理念は変わりません。抽象化のレベルが非常に高いため、開発者は、あらゆる種類の低レベルの詳細事項に関する心配から解放されます。

これで、今コンパイルおよびリンクしたプログラムを通常どおりに実行できます。唯一の違いは、OMP_NUM_THREADSと呼ばれるOpenMP固有の環境変数を使用して、使用するスレッドの数をランタイム・システムに通知する点です。この変数が設定されていない場合は、システムに依存した値が使用されます。

次に、2つおよび4つのスレッドを使用してこのコードを実行する方法を示します。

```
$ export OMP_NUM_THREADS=2              (1)
$ ./hello                               (2)
Hello OpenMP World, I am thread 0       (3)
Hello OpenMP World, I am thread 1       (4)
$ export OMP_NUM_THREADS=4              (5)
$ ./hello                               (6)
Hello OpenMP World, I am thread 0       (7)
Hello OpenMP World, I am thread 3       (8)
Hello OpenMP World, I am thread 2       (9)
Hello OpenMP World, I am thread 1       (10)
$
```

行1でスレッド数を2に設定し、通常どおりにプログラムを実行します（行2）。行3と4は、スレッドごとの出力を示しています。OpenMPでは、メインの"マスター"スレッドのスレッドIDは0であることが保証されています。その他のスレッドは、1から始まる連続した整数です。

行5でスレッド数を4に変更し、同じプログラムをまったく同じ方法で実行しますが、今度はスレッドごとに1行ずつで4行が出力されます（行7～10）。

出力がスレッドIDでソートされていないことに注意してください。これは、パラレル・コンピューティングの1つの側面を示しています。さまざまなスレッドが異なる時間に実行されるため、各実行にわたって同じ順序にならない可能性が高くなります。コード・ブロックが本当の意味でパラレルであれば、これは結果の正確性に影響しないはずです。

まだ説明していないことが1つあります。どの共有メモリのパラレル・プログラムでも、データがすべてのスレッドで共有される必要があるだけでなく、各スレッドにはプライベート・データも必要になります。プライベート・データは、それを所有するスレッドにのみ表示され、他のスレッドからの読取りや書込みはできません。forループ内のループ・カウンタは、この1つの例です。あるスレッドが別のスレッドによって使用されているループ変数を変更できたとしたら、実際に不都合が生じます。

OpenMPでは、shared句とprivate句を使用して、データにそれに応じたラベルを付けます。

さらに、C/C++では、コード・ブロックに対してローカルな変数はすべて、自動的にプライベート変数になります。次の単純な例は、先のサンプル・プログラムを書き換えたバージョンです。

```
#pragma omp parallel                                                    (1)
 {                                                                      (2)
  int thread_id = omp_get_thread_num();                                 (3)
  printf ("Hello OpenMP World, I am thread %d\n",thread_id);            (4)
 } // End of parallel region                                           (5)
```

変数"thread_id"は行3でローカルに宣言されているため、OpenMPでは自動的にプライベート変数になります。

これでOpenMPの最初で、かつ見たとおり単純な例の説明が終わったので、次は大局的に見て、使用可能なおもな機能について説明します。次の3つの主要コンポーネントが存在しますが、それぞれの例についてはすでに見てきました。

- **ディレクティブ** – 基本的な概念はパラレル・リージョンです。1つのアプリケーションが任意の数のパラレル・リージョンを持つことができます。各リージョン内で、開発者は、作業をスレッドにわたって分散する方法を選択できます。

  たとえば、`#pragma omp for`構文を使用すると、それが適用されるループの繰り返しはパラレルで実行され、各スレッドがすべての繰り返しのサブセット上で動作します。ただし、それだけではありません。

  パラレル・セクション（`#pragma omp`セクション）を通して、複数の独立したコード・ブロックを識別できます。これらのブロックは任意の種類のコードを含むことができ、パラレルで実行されます。これを使用すると、たとえば、I/Oと処理がオーバラップしたパイプラインを設定できます。

  OpenMP 3.0では、タスクが導入されました。ソース内のタスクは、`#pragma omp task`構文を使用して識別します。タスクとして識別されたコード・ブロックは独立して実行できることが前提になっています。タスクの作成とパラレル実行は、コンパイラとランタイム・システムによって処理されます。タスクは、より動的で、不規則なタイプのパラレル化に最適です。

- **ランタイム関数** – ランタイム環境への問合せや変更を行うために、さまざまなランタイム関数が使用できます。先に説明した`omp_get_thread_num()`関数が、その1つの例です。この関数は、スレッドIDを返します。ただし、これ以外にも多数の関数が存在し、実行を特定の状況に適応させたり、変更したりするための高い柔軟性が提供されます。

- **環境変数** – いくつかの環境変数がサポートされています。これらを使用して初期の環境を定義できます。それについてはすでに説明しました。変数`OMP_NUM_THREADS`を使用すると、パラレル・リージョンで使用される初期のスレッド数を設定できます。すべての環境変数にはデフォルト値があります。設定を変更しない限り、環境変数を設定する必要はありません。

OpenMPには、高レベルの抽象化や使いやすさの他、明示的なライブラリ呼出しに依存するプログラミング・モデルに対するもう1つの大きな利点があります。元のシリアル・バージョンを引き続き"組込み"の状態にしたまま、OpenMPを起動するためのオプション（たとえば、Solaris Studioコンパイラでの`-xopenmp`）でコンパイルせずに使用できるようにパラレル・アプリケーションを記述できます。

これは、開発フェーズや、土壇場でのリグレッションの発生時に使用できる適切な安全策です。`-xopenmp`オプションが省略された場合、コードは引き続きシリアルで実行されます。`-xopenmp`でコンパイルされていない部分は明らかにシリアルで実行されますが、少なくともアプリケーションは実行されます。

OpenMPの別の重要な特徴として、その仕様が、マイクロプロセッサおよびコンピュータ・システムのアーキテクチャの急激な変化に引き続き適応できることが挙げられます。その代表例が、2013年7月のOpenMP 4.0のリリースです。追加されたおもな新機能には、異機種システム（接続されたアクセラレータなど）のサポート、cc-NUMAアーキテクチャ、スレッドの取消し、タスキング・モデルの拡張などがあります。

ここでは、OpenMPについてのみ説明しました。詳しくは、[16]を参照してください。『Oracle Solaris Studio User's Guide』は、[17]に掲載されています。

POSIXスレッド

長年にわたって、POSIXスレッド（"Pthread"とも呼ばれます）は、共有メモリ・システムのための移植可能な、標準化された唯一のパラレル・プログラミング・モデルを提供してきました。

その移植性と柔軟性のおかげで、POSIXスレッドは、アプリケーションをパラレル化するための非常に一般的な方法になっています。

これは明示的なプログラミング・モデルです。開発者は、すべての詳細を処理する必要があります。Pthread APIは、スレッドの作成および管理やスレッドの同期化に使用できる関数を提供しますが、それ以上のレベルの機能は存在しません。

よく生じる疑問として、OpenMPとPthreadのどちらを使用したらよいかという問題があります。

OpenMPは、パラレル・プログラムを開発するための使いやすい、柔軟な方法を提供しますが、スレッドやその動作に対する制御が必ずしも明示的であるとは限りません。これは設計によるものであり、多くの場合はこれで十分ですが、開発者がスレッディング・モデルを非常に詳細な方法で制御したいときもあります。この場合は、Pthreadによりを使用すると、自然で、移植可能なソリューションが提供されます。

Pthread APIには、多数の関数と機能が含まれています。あまりに多すぎて、このホワイト・ペーパーでは説明しきれません。代わりに、このセクションの残りの部分では、スレッド作成と相互排除のコア・サブセットについて説明します。

Pthreadでの新しいスレッドの作成方法は、関数`pthread_create()`の呼出しです。この関数の主要なパラメータは、新しいスレッドが実行するルーチンへのポインタです。スレッドは作業を完了すると、戻り値を返すことができるように、"結合される"のを待機します。

APIコール`pthread_join()`は、特定のスレッドの完了を待機し、子スレッドからの戻り値を呼出し元のスレッドから使用できるようにします。次のコードでは、スレッドが作成され、実行されて、値を返しています。Pthread固有のコード片が青で色分けされています。

```
#include <pthread.h>
#include <stdio.h>

void * work(void * param)
{
   printf("In child thread\n");
   return (void*)8;
}

int main()
{
   pthread_t childthread;
   pthread_create(&childthread, 0, work, 0);

   printf("In parent thread\n");

   long value;
   pthread_join(childthread,(void*)&value);

   printf("Child returned %i\n",value);
}
```

Pthreadを使用するデメリットは、開発者がさまざまなスレッド間のやり取りのすべての側面の管理に責任を負うことです。たとえば、同じ変数を操作する2つのスレッドが存在する場合は、これらの2つのスレッドがその変数を同時に更新できないようにする必要があります。これを行うためのもっとも簡単な（ただし、もっとも非効率的な）方法は、変数への更新を相互に排他的なロック、つまり"mutex"（ロック）の制御下に置くことです。

このmutexロックは、一度に1つのスレッドだけが取得できます。共有変数を更新するには、スレッドがmutexロックを取得し、変数を更新してから、mutexロックを解放する必要があります。APIによって、ロックの取得と解放の機能が提供されます。それを行うことができるのは一度に1つのスレッドだけであることが保証されています。

mutexロックにはライフ・サイクルがあります。使用する前には初期化する必要があり、必要がなくなったら破棄する必要があります。次の例は、2つのスレッドがmutexロックを使用して共有変数を増分する様子を示しています。Pthread固有のコード片が青で色分けされています。

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t mutex;
volatile int value=0;

void *work(void * param)
{
   for(int i=0;i<200; i++)
   {
      pthread_mutex_lock(&mutex);
         value++;
      pthread_mutex_unlock(&mutex);
   }
}

int main()
{
   pthread_t childthread[2];

   pthread_mutex_init(&mutex,0);

   for (int i=0;i<2; i++)
   {
      pthread_create(&childthread[i], 0, &work, 0);
   }
   for (int i=0;i<2; i++)
   {
      pthread_join(childthread[i],0);
   }

   printf("Value of shared variable = %i\n",value);

   pthread_mutex_destroy(&mutex);
}
```

この例では、スレッドが共有変数を増分します。このタイプの更新は、かなり多くの状況で発生します。たとえば、完了した作業量、または作業を完了するために要した時間の統計が保持されている場合です。

これらの状況に対してmutexロックを使用する代わりに、不可分操作を使用する方が効率的な場合があります。これらは一般に非常に効率的ですが、移植は不可能です。Solarisでは、これらはatomic.hヘッダー・ファイルで宣言されます。前の例は、次に示すように、不可分操作を使用するようにコーディングし直すことができます。不可分操作の使用に固有のコードが青で色分けされています。

```
#include <pthread.h>
#include <stdio.h>
#include <atomic.h>

volatile unsigned int value=0;

void *work(void * param)
{
   for(int i=0;i<200; i++)
   {
      atomic_add_int(&value,1);
   }
}

int main()
{
   pthread_t childthread[2];
   for (int i=0;i<2; i++)
   {
      pthread_create(&childthread[i], 0, &work, 0);
   }
   for (int i=0;i<2; i++)
   {
      pthread_join(childthread[i],0);
   }
   printf("Value of shared variable = %i\n",value);
}
```

この例では、変数"value"を*volatile*として宣言しました。これは、この変数をレジスタ内にキャッシュしてはいけないことをコンパイラに通知します。代わりに、使用される場合は常にメモリからフェッチされ、変更されたらただちに元のメモリに格納されるようにします。volatileキーワードは、この状況では必ずしも必要ではありません。ループ内で関数呼出しを行っているため、この変数はmutexロックが解放される前に必ず元のメモリに格納されますが、その他の状況では、共有変数をこの方法で宣言することが必要になる場合があります。

次のコードについて考えてみます。

```
#include <pthread.h>
#include <stdio.h>

int ready = 0;

void *work(void * param)
{
   while (!ready) {}
}

int main()
{
   pthread_t childthread;

   printf("Started\n");

   pthread_create(&childthread, 0, &work, 0);
   printf("Continue\n");

   ready=1;

   pthread_join(childthread,0);

   printf("Finished\n");
}
```

ルーチン"work()"を実行しているスレッドは、変数"ready"が0以外になるのを待機します。変数"ready"がvolatileとして宣言されていないため、コンパイラは、この変数が1回だけ読み取られるようにループを自由に最適化できます。

これは完全に有効な最適化ですが、パラレル・プログラムでは不要で、かつ好ましくない副作用が生じます。

この変数が1に設定される前に読み取られた場合、子スレッドはそれを二度と読み取ることができないため、永久にループします。この変数が、子スレッドが起動されるまでに1に設定された場合、子スレッドはただちに復帰します。

このコードの問題は、ほとんどの場合は変数"ready"が正しく設定され、期待したとおりにコードが動作することにあります。ただし、まれなケースとして、この変数が"適切な"時間に設定されないと、プログラムは無限ループに陥ります。この種の予測不可能な動作が、マルチスレッド・アプリケーションのデバッグを困難にしています。[4]

[4] これは、OpenMPにより多くの安全策が存在する領域です。"フラッシュ"の概念によって、システム全体にわたる変数値の一貫性を強制するための適切に定義された、移植可能な方法が提供されます。flush構文がすべての主要な構文（バリアなど）に組み込まれているため、通常、共有変数をvolatileとして宣言する必要はありません。

その他の同期メカニズム

mutexや不可分操作に加えて、スレッド間の調整を行うためのその他のメカニズムがいくつか存在します。

リーダー/ライター・ロックはmutexの変形であり、複数のリーダー・スレッドがロックを保持し、保護された変数を読み取ることができるか、または1つのライターがロックを保持し、保護された変数を変更できます。

セマフォはカウンタであり、"post"がカウンタを増分し、"wait"はカウンタが0を超えるまで待機します。このメカニズムを使用すると、作業の準備ができるまでスレッドを待機させることができます。

条件変数は、あるスレッドが他のスレッドに信号を送信できるようにし、他のスレッドがその信号を受信するまで待機できるようにするという点で、セマフォに似たメカニズムです。ただし、条件変数では、共有変数への保護されたアクセスも許可されます。そのため、データを通信したり、使用可能なデータが存在することを示したりすることが容易です。

スケーラブルなアプリケーションの開発方法

シリアル実行だけでなく、マルチスレッド化された実行についても、パフォーマンスの高いアプリケーションの記述に関しては多くの誤解があります。

ハードウェアは引き続き進化し、コンパイラはますます高度化しているため、これは実際に驚くべきことではありません。結果として、最近まで有効だったガイドラインの多くはすでに必要なくなっているか、または裏目に出ることさえあり、最適ではないコードが生成される原因になっています。

そのため、ここでは、長続きするパフォーマンスの利点を備えたコーディング手法に焦点を当てたいと思います。ときには、コンパイラがこれらのパフォーマンスの問題を解決することは困難なだけでなく、それに対処することさえ基本的に不可能である場合もあります。これは特に、データ・レイアウトやcc-NUMAに関連したパフォーマンス・ボトルネックに当てはまります。

シリアル・パフォーマンス

大規模スレッド化システムに関するホワイト・ペーパーでこの件について書くことは奇妙にも思えますが、それがまさに、ここでそれについて触れる理由です。ほとんどの場合、開発者はシリアル・パフォーマンスをあたり前と見なし、ただちにパラレル化に着手します。理解できることではありますが、シリアル・パフォーマンスを飛ばして進むと、たいていはいずれ、スケーラビリティが妨げられる可能性があります。

その理由はきわめて単純です。アプリケーションがシングルスレッドを使用して高パフォーマンスで実行されない場合、そのコードをパラレルで実行したらどうなるでしょうか。その場合に高パフォーマンスが得られると想定することは、かなり幼稚な考えと言えます。

それどころか、スケーラビリティが低下する可能性が高くなります。これは、シリアル・パフォーマンスが低い場合はほぼ常に、キャッシュやメモリ・システムの最適でない使用が原因になっているためです。これにより、メモリ・トランザクションが必要以上に発生します。これらに余分な時間がかかるだけでなく、必要以上のデータをフェッチしたり、書き戻したりする必要があるため、帯域幅（場合によってはレイテンシも）の要件もより厳しくなります。

同じアプリケーションのパラレル・バージョンでは、複数のスレッドがインターコネクトを同時に使用するようになるため、帯域幅にただちに負荷が加わります。使用されるスレッド数が増えるにしたがって、この問題がより目立つようになり、帯域幅も必要以上に早く飽和します。

そのため、まず、シリアル・アプリケーションが妥当なパフォーマンスで確実に実行されるようにすることを強くお勧めします。この作業は、多くの場合、追加のコンパイラ・オプションをいくつか調べるだけで済みます。

Oracle Solaris Studioコンパイラなどの最新のコンパイラは、アプリケーションのチューニング時に大きな威力を発揮します。この鍵となるのは、コードの目的を判定するコンパイラの機能です。それはどのような意味でしょうか。人間が読みにくいだけでなく、コンパイラが実際に処理されている内容を判定することも困難な、まるで暗号のようなコードを記述することはきわめて容易です。その結果、最適なコードを生成できなくなる可能性があります。

コンパイラが最高のパフォーマンスを実現することが困難であると判定する可能性のあるもう1つの原因は、データ構造の領域にあります。これについての詳しい説明はこのホワイト・ペーパーの範囲を超えていますが、ここでは、この問題の概要のみを示します。

データ構造（構造体の配列など）はすべて、特定の方法でメモリに格納されます。最高のパフォーマンスは、データ・アクセスがメモリ内の格納順序にスムーズに従っている場合に得られます。キャッシュが最適な方法で使用されている場合や、データ・プリフェッチなどの手法がキャッシュやメモリ・レイテンシを非常にうまく隠すことができる場合はこれに該当します。

ただし、これに当てはまらない場合は、パフォーマンスが急速に低下することがあります。これにより、メモリ・アクセス時間が長くなり、帯域幅の要件が増加します。

標準的な例として、配列へのアクセスがあります。次のコード片は、2次元配列"a"を1行ずつ読み取ります。配列はメモリ内で行ごとに格納されているため、これはC/C++でのパフォーマンスのための正しい方法です。

```
for (int i=0; i<nRows; i++)
       for (int j=0; j<nColumns; j++)
              sum += a[i][j];
```

これらの2つのループが逆になっていると、パフォーマンスははるかに低下します。特に、大きなマトリックスの場合は、さらに多くのデータ・キャッシュが存在し、データのTLB（Translation Look-Aside Buffer）ミスがはるかに多く発生します。

一般に、コンパイラの最適化では、コードを再構築してメモリ・アクセスを改善しようとします。たとえば、上に示したネストされたループでは、ループの順序を交換します。ただし、特定の最適化を検証するために必要な情報が不足しているため、コンパイラでできることは制限されることがあります。たとえば、レコードの配列内の特定の要素が使用されていない場合はどうなるでしょうか。これにより、キャッシュ内に必要以上のデータがロードされますが、これを修正するには、メモリ内のデータ構造の格納場所を並べ替える必要があります。それには、おそらく、その設定方法が変わります。コンパイラがコード全体にわたってこれを完全に把握していない限り、制限は、コンパイラでこれが処理できないほど果てしなく拡大されます。

これは、開発者がデータ・レイアウトやデータ構造の定義を考え直すことによって解決できる領域です。

要約すると、コードのシリアル・パフォーマンスを最適化する機会は数多く存在し、コンパイラのオプションは行動を起こすための非常に優れた場所です。プログラムがシリアルで十分高速に実行されるようになったら、次はパラレル・パフォーマンスについて検討します。これが、このセクションの残りの部分のトピックです。

アムダールの法則

パラレル・アプリケーションの高速化を予測するために使用できるルールが存在します。それは、有名なコンピュータ・アーキテクトであるGene Amdahl（ジーン・アムダール）にちなんで名付けられ、一般には"アムダールの法則" [18]と呼ばれています。

この法則が基本的に述べているのは、プログラムのシリアル（つまり、非パラレル）部分が存在すると、それがすぐにでもパフォーマンスに影響するということです。

計算式で表すと、この法則は、予測される高速化を"P"と"f"の2つのパラメータの関数として次のように示します。

```
SP(f) = 1/(1-f + f/P)        （アムダールの法則）
```

この計算式で、SP（f）は、"P"と"f"の関数としてのパラレル高速化です。ここで、"P"は使用されるスレッド数を示します。一方、"f"は、アプリケーションのパラレル化できる部分に費やされる時間の割合です。明らかに、"f"は[0,1]の間にあり、両端がコーナー・ケースです。ケースf=0は純粋にシーケンシャルなプログラムを示し、パラレル化が実行できることによる高速化はありません。f=1の値は、アプリケーション全体を最初から最後までパラレル化できるため、使用可能なスレッドの数に基づいて直線的にスケーラブルであることを示します。このタイプのアプリケーションは多くの場合、"極端なパラレル"と呼ばれます。

より現実的なシナリオでは、"f"の値は0と1の間のどこかの値であり、後者に近い方が望まれます。

まだ説明していない重要なことがあります。それは、アムダールの法則にはパラレル・オーバーヘッドがまったく含まれていないということです。これは現実的な仮定ではありませんが、この法則が実際には最良の場合のシナリオを示すことを意味しています。

この計算式は単純に見えますが、その意味を無視してはいけません。

図5は、割合"f"のさまざまな値の関数としての予測される高速化を示しています。最大16個のスレッドが使用されると仮定しています。このグラフでは、"f"が割合（%）として表されていることに注意してください。

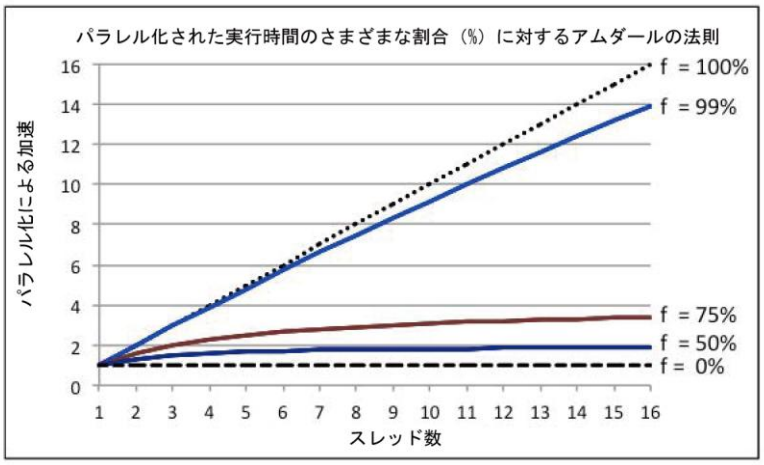

図5：割合"f"のさまざまな値の関数としての予測されるパラレル高速化

2本の破線は、f = 0%とf = 100%のコーナー・ケースに対するパラレル高速化の値です。予測されるように、f = 0%の曲線はまったく高速化を示していません。反対の状況であるf = 100%は、完全に直線的な高速化を示しています。ただし、先ほど説明したように、"f"はこのどちらでもなく、その間のどこかの値であると仮定する方が現実的です。

アムダールの法則が示す現実が、3本の実線の曲線によって示されています。見て分かるように、シリアル・アプリケーション内の実行時間の99%をパラレル化できる場合、16個のスレッドを使用した高速化は16ではなく、14です。これはまだ、まったく悪い結果ではありませんが、さらに多くのスレッドを追加するとすぐに収穫逓減に達します。

他の2本の曲線は、それぞれf = 75%とf = 50%に対するものです。見て分かるように、どれだけ多くのスレッドを使用しても、高速化はそれぞれ4と2を超えていません。

これが、アムダールの法則の一般的な特性です。この計算式から分かるように、パラレル高速化は常に1/(1-f)未満になります。

このセクションの最後に、実際的なヒントを示します。2つのタイミングだけを使用して、任意のアプリケーションに対する割合"f"を見積もることができます。

アムダールの法則は、次のように、すでに分かっているか、または測定できる値で"f"を表すように書き換えることができます。

```
f = (1-1/SP)/(1-1/P)
```

この計算式では、"P"（使用されるスレッド数）は明らかに分かっており、パラレル化による加速が測定できます。

たとえば、あるジョブのシングルスレッド時間が100秒であると仮定します。また、2つのスレッドを使用した場合、57秒の経過時間を測定したと仮定します。上の計算式を適用すると、次の値が得られます。

```
f = (1-1/(100/57))/(1-1/2) = 0.86 = 86%.
```

今度は、"f"にこの値を使用し、アムダールの法則を適用します。これに従うと、4つのスレッドでの高速化の見積もりは、S4(0.86) = 1/(1-0.86 + 0.86/4) = 2.82になります。

同様の方法で、8つのスレッドでの高速化を4.04と見積もることができます。まだ4つのスレッドを使用した実行よりは高速ですが、収穫逓減が発生しています。

アムダールの法則を、アプリケーションをパラレル化しない理由としては絶対に使用しないでください。ますます多くの開発者のグループが、パラレル・コンピューティングの実行を通して大きな成果を上げています。アムダールの法則が述べているのは、非常にスケーラブルなアプリケーションを目標にしている場合は、アプリケーション内のシリアル部分を認識し、それをパラレル化する方法を見つける必要があるということです。多くの場合、これは可能であり、かつ非常にやりがいがあります。

その点に関しては、従来の"問題点"として、ロックやバリアなどの同期プリミティブがあります。パラレル・インフラストラクチャの一部ではありますが、これらは、システム内部の深い特定の地点ではシリアル操作です。さらに悪いことに、スレッドが追加されるに従って、これらのプリミティブのコストは増えていきます。そのため、次に説明するように、これらは十分注意して使用する必要があります。

同期

先ほど説明したように、アプリケーションを適切に拡張するには、コードのシリアルの（シングルスレッドの）部分に費やされる時間を少なくする必要があります。

同期は通常、シリアル実行になります。たとえば、すべてのスレッドが一部の作業を完了した後、それぞれの作業を再開するには、その前にmutexロックによって保護されている共通の状態を更新する必要があるとします。

mutexロックによって保護されているコードを実行できるのは、一度に1つのスレッドだけです。このコードをまだ実行していないその他のすべてのスレッドは待つ必要がありますが、それは、実行がシリアル化されていることを示します。各スレッドが同時にではなく、他のスレッドの後に実行されます。

同期が必要なスレッド数が多くなればなるほど、同期にかかる時間が長くなるため、同期に費やされる実行時間の比率が増えます。

拡張可能なコードを生成するには、スレッドが同期に費やす時間をできるだけ少なくし、独立した作業を実行する時間をできるだけ多くする必要があります。

同期が挿入される段階は複数存在し、次のように分類できます。

- **同期なし** すべてのスレッドが、他のスレッドとやり取りすることなく独自の作業を完了します。結果として、アプリケーションは同期のコストを必要とせずに多数のスレッドに拡張できるため、これは理想的なアプローチです。
- **共有データのローカル・コピー** 保持すべき何らかの共有状態が存在する場合がありますが、各スレッドはこのデータの独自のコピーを更新でき、最終的な値は必要になった場合にのみ計算できます。たとえば、数値の配列の合計値を計算するには、各スレッドに、配列のセグメントの加算から得られた部分的な合計を保持するローカル変数を割り当てることができます。すべてのスレッドがそれぞれの作業を完了したら、これらの部分的な合計を1つの値に結合できます。部分的な合計を結合する操作のコストはかなり高くなりますが、結果の合計を1つの変数に保持し、ロックによって各更新を保護しようとするコストに比べると低くなります。
- **軽量な同期** 不可分操作などの一部の同期プリミティブは低コストですが、同期*なし*に比べるとまだ明らかに高くなります。

- **Pthreadの同期プリミティブ** Pthread APIによって提供される同期プリミティブは移植可能ですが、不可分操作に比べると負荷は高くなります。多くの場合は、オペレーティング・システム・カーネルの呼出しが必要であり、スレッドがイベントの待機中にループするのではなく、スリープ状態になる場合があります。スレッドを再び起動する高コストになります。
- **"停止を伴う同期"** 最悪のタイプの同期は、すべてのスレッドが、ロックで保護されたコード・ブロックを完了するまで作業を停止するタイプです。実行がシリアル化される（アムダールの法則）だけでなく、スレッドが追加されるに従って同期のコストが増加するため、これは非常に非効率的です。

OpenMPなどのより高いレベルのパラレル化アプローチを使用する利点の1つは、サポート・ライブラリによって、共通データの共有の問題を解決するための効率的なアルゴリズムが提供されることです。

たとえば、OpenMPでは、上で説明した配列の要素の合計を、*reduction*句を使用して簡単に処理できます。次に、コンパイラが、余分な同期を回避するためにローカル・ストレージを利用しながら、もっとも効率的な方法でこれを処理します。

データの局所性のための最適化

最新のパラレル・コンピュータ・システムはすべて、cc-NUMAアーキテクチャを備えています。その理由は単純です。スケーラブルなメモリ帯域幅を提供するために、最新の汎用マイクロプロセッサはすべて、チップ上にメモリ・コントローラを搭載しています。先に説明したように、キャッシュ・コヒーレンシを通して、すべてのメモリがシステム全体にわたって見えるようになります。

これが、アプリケーションにとって非常に適切に機能します。シリアル・アプリケーションでも、システム内のすべてのメモリを使用できます。

ただし、問題が1つあります。別のプロセッサのメモリへのアクセスには時間がかかります。データへのアクセスは、特定のアドレスを含む単純なロード（または格納）命令であるため、アプリケーションに対して完全に透過的に行われます。ハードウェアがデータの場所を解決し、そのデータを必要とするスレッドに移動しますが、データを別のソケット上のメモリから読み取る必要がある場合は時間がかかります。

T4、T5、およびM5ベースのシステムでは、ローカル・アクセスとリモート・アクセスの違いは古いcc-NUMAシステムほど大きくありませんが、高いレベルのスケーラブルなパフォーマンス、つまり多数のスレッドへの拡張を実現したい場合は無視できません。

シリアル・アプリケーションのためのもっとも高速なメモリ・アクセスを実現するために、Solarisは、"ファースト・タッチ"ポリシーを使用してメモリを割り当てます。つまり、初めてデータにタッチしたプロセスまたはスレッドがそのデータを所有します。そのプロセスは、データがタッチされた時点で実行されていたプロセッサに接続されているメモリ内にデータを保持します。"タッチ"は、TLBエントリの初めての作成として定義されます。

シリアル・アプリケーションでは実行中のスレッドは1つしか存在せず、そのスレッドがそのローカル・メモリ内にすべてのデータを保持するため、このデフォルトの選択は非常に適切に機能します。データが収まらない場合は、隣接するメモリにオーバーフローします。Solarisでは、長年にわたってcc-NUMAへの対応がますます強化されてきており、システムは、このようなオーバーフローをもっとも効率的な方法で処理するだけの十分な機能を備えています。

パラレル・アプリケーションの場合は、1つのスレッドがすべてのデータを所有するかどうかが明らかでなくなるため、状況は少し異なります。逆に、別のプロセッサ上で実行されているスレッドがデータを使用するためにリモート・メモリにアクセスすることが必要になる可能性があるため、これはおそらく望ましいことではありません。

幸いにも、多くの場合はファースト・タッチ・ポリシーがここでも役立ちますが、データ初期化の部分でソース・コードの変更が必要になる可能性があります。

ファースト・タッチの動作を考慮に入れて、データが初めて初期化される場所を見つけ、この部分をどのようにパラレル化できるかを確認する必要があります。最適な戦略は、スレッドとデータの関係を判定し、スレッドがその部分のデータを確実に初期化するようにすることです。

これを少し具体的にするために、OpenMPを使用した非常に単純な例を示します。スレッドが、"a"という名前のベクトルのサブセットを使用していると仮定します。"a"がシングルスレッドによって初期化されるようにしたとすると、他のスレッドはすべて、"a"が存在するメモリにアクセスする必要があります。代わりに、OpenMPの"#pragma omp parallel for"構文を使用して、データの初期化をパラレルで実行します。次のようにします。

```
#pragma omp parallel for schedule(static)

     for (int i=0; i<n; i++)

         a[i] = 0;
```

OpenMPランタイム・システムは、ループの繰り返しをスレッドにわたって分散します。schedule(static)句によって、最初のスレッドが"a"の最初のセクションを初期化し、2番目のスレッドが次のチャンクをというように、以降も同様に処理されます。結果として、各スレッドがこの配列の1つのセクションを"所有する"ようになり、それをそのローカル・メモリ内に（それがシステム上のどこにあろうとも）保持します。

このアプローチによって、データ配置を最適化するための移植可能で、容易な方法が提供されますが、それをすべての状況およびすべてのアプリケーションに適用できるわけではありません。たとえば、データ・アクセスのパターンが実行中に変化する場合があります。このような場合のため、および最高の柔軟性のために、Solarisは"madvise"（3C）システム・コールを提供しています。

この関数は、引数として開始アドレス、長さ、およびキーワードを受け取ります。このキーワードは、データの取り扱い方に関するアドバイスです。たとえば、データを、それを初期化した最初のスレッドではなく、そのデータを使用する次のスレッドに移行するようシステムに通知することもできます。

最後に重要なこととして、データ配置がページ・レベルで制御されることに触れておく必要があります。

偽共有

偽共有は、スケーラビリティに悪影響を与える、もっとも理解されていない要因の1つです。これは、偽共有があまり一般的でないためかもしれませんが、発生した場合は、パフォーマンスが著しく低下します。

偽共有について説明するには、データ・キャッシュのレベルで何が発生しているかを少し詳しく見てみる必要があります。このレベルのデータは、"キャッシュ・ライン"に体系化されています。キャッシュ・ラインのサイズはアーキテクチャに依存し、すべてのキャッシュ・レベルにわたって同じである必要もありません。標準的なサイズは、32または64バイトです。

プロセッサであるデータ要素が必要になると、この要素がフェッチされるだけでなく、必要なデータを含むライン全体も（レベル1の）キャッシュにロードされます。このように処理される理由は、転送のコストを償却するためです。パフォーマンスを向上させるには、新しいラインがロードされる前に、同じキャッシュ・ライン内のすべての要素を使用する必要があります。

パラレル・アプリケーションでは、同じラインの複数のコピーが存在するという状況が容易に発生します。これが発生する1つの理由は、複数のスレッドが、すべてが同じラインに含まれる異なる変数を読み取るためです。

ここまでは何も心配することはありませんが、これらのスレッドのいずれかが、このような共有されているライン内の変数を変更した場合はどうすればよいのでしょうか。その時点で、新しく変更されたラインとその他のコピーとの間に非一貫性が生じます。システムは、これらのその他のコピーを"無効にする"ことによって、これを処理します。このようにして、スレッドはこのラインがもう使用できなくなり、システム内の別の場所から最新のコピーを取得する必要があることを認識します。

これは、実際には行き過ぎになることがあります。それは、あるスレッドが対象としているキャッシュ・ラインの部分は変更されていない可能性があるが、このステータスがバイト・レベルではなく、ライン単位で保持されているため、スレッドがこれを認識する方法は存在せず、新しいコピーをフェッチするためです。

この"偽共有"の結果として、コヒーレンシ・トラフィックは増加し、キャッシュ・ラインはシステム全体を移動します。この状況をできるだけ効率的に処理するために多大な努力が払われているにもかかわらず、このコストは高くなる場合があります。

偽共有は常に発生しますが、実際にはそれほど目立つわけではありません。アプリケーションの中心部分に見られるように、同じキャッシュ・ラインへの複数の書込みが非常に頻繁に、かつ比較的短期間で発生する場合は、問題になります。[5]

これは回避すべき状況であり、たいていの場合、回避できます。多くの場合、もっとも簡単な解決策は、更新を"局所的"にして、最終的な結果だけを共有変数に書き込むようにすることです。

OpenMPでは、スレッドに対してプライベートな変数を使用して、これを非常に容易に実現できます。

ヒントと技

このセクションの最後に、高いパフォーマンスを得るための追加のアドバイスを示します。

- **高いパフォーマンスは、適切なコンパイラを使用することから始まります** これは、シリアル・パフォーマンスとパラレル・パフォーマンスの両方に当てはまります。Oracle Solaris Studioコンパイラは、最適なコードを生成するように継続的にチューニングされています。チームは、考えられる最高のパフォーマンスを保証するために、SPARCエンジニアと緊密に協力しています。
- これらのコンパイラにバンドルされている**Oracle Solaris Studioパフォーマンス・アナライザ**を使用します。このツールは、C、C++、Java、またはFortranで記述された任意のシリアルまたはパラレル・アプリケーションに含まれているパフォーマンスのホット・スポットを容易に識別します。このツールは、ユーザーが最適化の機会に優先順位を付けるのに役立つだけでなく、通信のボトルネックや、同期に費やされた時間などもすばやく表示します。

[5] 偽共有は、読取り専用データでは発生しないことに注意してください。

- **コンパイラ・オプションを試してみてください** コード内の"ホットな"セクションの識別に加えて、コンパイラが実行した最適化も示されます。これにより、その状況に合わせて調整されたその他のいくつかのオプションを調査する必要性が指摘される可能性があります。
- **明確なコードを記述します** 自分がコンパイラだとしたらどうするかを考えるようにしてください。ソースだけを見て、コードのこの部分の目的が何であるかを実際に把握できるでしょうか。特定のコード・ブロックの意図がコンパイラにとって明確であれば、コンパイラは、より効率的なコードを生成できます。不明瞭なコーディングにより、これがはるかに困難になる場合があります。その点に関しては、チューニングのヒントが記載された古い書籍に対して別の目的を見つけることもできます。これらの多くはすでに必要なくなっており、考えられる最高のコードがコンパイラによって生成されない程度まで、意図を実際に隠す場合があります。
- **データ構造を確認します** コンパイラは多くのことを自動的に実行できますが、データ構造に関する問題をすべて修正することはできません。データ構造を、キャッシュ・ラインが最適な方法で使用されるように設定するようにしてください。これは、連続したデータ・アクセスがメモリ内でも連続することを保証することによって達成されます。
- **大局的に見るようにしてください** 低レベルの詳細はコンパイラに任せるのが最適です。コンパイラはプロセッサの特性を詳細に認識しており、どのような種類のコードを生成すべきかを非常に適切に判断できます。ユーザーの責任は、コンパイラが正しいプロセッサを対象としていることを確認することです。デフォルトでは、コンパイルを実行しているプロセッサ用に最適化されます。これが展開に使用されるものとは別のプロセッサである場合は、Oracle Solaris Studioコンパイラで`-xtarget`オプションを使用して、最適化の対象とするプロセッサを特定してください。
- **アムダールの法則を決して忘れないでください** この法則はずっと当てはまってきており、現実を示してもいます。スケーラビリティに関するほとんどの問題はこの法則に戻ります。そのため、プログラム内のシリアル部分をできるだけ回避してください。
- **できるだけプライベート・データを使用します** データの共有は非常に便利であり、パラレル・アプリケーションでは必要な場合もありますが、データは実際に必要な場合にのみ共有するようにしてください。プライベート・データの使用には、パフォーマンス上の利点がいくつかあります。複数のスレッドが同時に同じメモリ位置にアクセスするリスクは存在しないため、偽共有が回避されます。また、プライベート・データを、それを必要とするスレッドのメモリ内に配置することも容易になります。cc-NUMAシステムには、この種のローカル・メモリ・アクセスを強くお勧めします。
- **ロックとバリアの過剰な使用を回避してください** ロックを過剰に使用するアプリケーションが多すぎるように感じます。これらが必要な場合は確実に存在し、実際に便利に使用できますが、これらのコストの高さを認識しているユーザーはほとんどいません。ロックのコストはまた、使用されるスレッド数が増えるに従って増加し、すぐにスケーラビリティのボトルネックになる場合があります。
- **パラレルで思考してください** 最後に間違いなく重要なこととして、新しいコードを開発する場合はパラレルの方法で思考してください。1つのスレッドだけで作業を実行することが実際に必要ですか、またはその作業を分散できますか。1つのスレッドにすべての作業を実行させる一方で、その他のスレッドがロックやバリアなどの潜在的にコストの高い同期構文で待機しているのではなく、各スレッドに同様の作業を実行させるようにしてはどうでしょうか。[20]

多数のコアによる仮想化の実現

多数のスレッドを含むシステムを使用する1つの理由はパラレル・プログラムの展開ですが、おそらくアプリケーションのスケーラビリティは制限され、一部のスレッドを効率的に使用できないか、またはワークロードがシリアル・ジョブとパラレル・ジョブの混在で構成されます。

これは、仮想化、リソースのパーティション化、およびワークロードの分離が姿を現す領域です。

考え方としては、それぞれが*同じサーバー*上で1つのスレッドまたは限られた数のスレッドを使用する複数のアプリケーションを統合します。これらのアプリケーションがすべて同時にピーク時負荷を抱えることはないか、または過剰にプロビジョニングされていた場合は、それらが共有するコンピューティング・リソースの総量を個別のマシンの合計よりはるかに少なくすることができます。これにより、財務コスト、サーバー台数、床面積、ソフトウェア・ライセンス、電気、冷却、および保守作業の大幅な削減を実現できます。

これを従来の方法で行うには、同じマルチプログラミング・オペレーティング・システム・インスタンス上の複数のアプリケーションを結合します。これは、各アプリケーションに互換性のないオペレーティング・システム・バージョンや異なるメンテナンス・ウィンドウが必要な場合や、非常に一般的なこととして、アプリケーション・オーナーに独立性またはパフォーマンスやセキュリティの干渉からの自由が必要な場合は問題になることがあります。これに対する答えは、サーバーの仮想化によって提供されるワークロードの分離という、ますます一般的になっているオプションです。サーバーの仮想化は、一部のプラットフォームでは長年にわたって使用可能ですが、最新のUNIXサーバーでワークロードの分離のために使用される展開方法としてますます一般的になっています。

ワークロードの分離には複数のモデルが存在します。物理パーティション化などの一部の方法では、完全な電気的分離が提供されます。あるパーティション内のすべてのコンポーネントが即座に故障するか、または徐々に機能が低下することがあっても、その他のパーティションは影響を受けません。別の方法である仮想マシン（"VM"）は、ハイパーバイザーを使用して個別のコンピュータのような見せかけた環境を作り出します。これらの仮想マシンは、実際には、HWT、DIMM、メモリ・バス、システム・インターコネクト、I/Oサブシステムなどのコンポーネントを共有します。"オペレーティング・システムの仮想化"または"OSの仮想化"と呼ばれる3番目のモデルは、個別のプロセスの複数のセットが同じOSカーネルを共有している場合でも、複数のオペレーティング・システム・インスタンスという仮想環境を提供します。

仮想化

オラクルのサーバー仮想化製品は、x86およびSPARCアーキテクチャと、Linux、Windows、Oracle Solarisなどのさまざまなオペレーティング・システムをサポートします。オラクルでは、ハイパーバイザーベースのソリューションに加えて、コンピューティング環境全体のためのもっとも完全で、かつ最適化されたソリューションを提供するために、ハードウェアおよびOracleオペレーティング・システムに組み込まれた仮想化も提供しています。

オラクルのSPARC T5およびM5サーバーは、M5、Oracle VM Server for SPARC（ハイパーバイザーベース）、Oracle Solaris Zones（OSベース）を含む、仮想化手法の範囲全体をカバーしています。T5およびM5のきわめて高いレベルのマルチスレッド化の結果として、これらの機能は、サーバー統合およびクラウド・コンピューティング・インフラストラクチャのための従来にはない柔軟性と効率的なスケーラビリティを提供します。

Oracle Solaris Zones

Oracle Solaris Zonesは、業界でもっとも広範囲に認識されている形式のOS仮想化テクノロジーです。Oracle Solaris Zonesは、2005年にSolaris 10で初めてリリースされ、あるゾーン内のワークロードが別のゾーン内のプロセスと対話できないようにするセキュリティ境界（分離）から開始されました。それ以降は定期的に機能が追加され、Solarisでのその他の技術革新との統合により、この形式の仮想化が非常に一般的になりました。オラクルは、SPARC Solarisシステムの40%がOracle Solaris Zonesを使用すると見積もっています。

ゾーンは、Oracle Solarisシステムの1つのインスタンス内に作成された仮想オペレーティング・システム環境であり、*完全なOSカーネルではありません*。ゾーンを作成すると、プロセスがシステムの残りの部分から分離されたアプリケーション実行環境が生成されます。この分離により、あるゾーン内で実行されているプロセスが他のゾーン内で実行されているプロセスを監視したり、影響を与えたりすることができなくなります。スーパーユーザーの資格情報で実行されているプロセスでさえ、他のゾーン内のアクティビティを表示したり、影響を与えたりすることができません。

ゾーンはまた、アプリケーションを、それが展開されているマシンの物理属性から分離する抽象化レイヤーも提供します。これらの属性の例には、物理デバイスのパスが含まれます。

ゾーンは、少なくともOracle Solaris 10リリースを実行している任意のマシンで使用できます。システム上のゾーンの数の上限は8192です。単一システム上で効率的にホストできるゾーンの数は、すべてのゾーン内で実行されているアプリケーション・ソフトウェアの合計のリソース要件によって決定されます。

統合された関連機能により、Oracle Solaris Zonesに次の利点が提供されます。[22]

- 構成可能なセキュリティ境界。機能を微調整して、ゾーンの機能やセキュリティを増やしたり減らしたりできるようになります。
- ゾーンのリソース使用率の管理制御。これには、HWTの使用、ゾーンから使用できるHWTの量、ゾーンから使用できるRAMや仮想または共有メモリの量、消費されるネットワーク帯域幅、およびその他多数が含まれます。
- ストレージにアクセスするための各種の方法。これには、さまざまなファイル・システムや、RAWストレージへのアクセスが含まれます。
- 以前のバージョンのSolarisの構成し、そのバイナリを実行する"ブランド・ゾーン"を作成する機能。

OSの仮想化は、複数の独立したプロセス・セットが共通カーネルを共有するという前提条件をその中心に持つ、サーバー仮想化のモデルを使用します。共通カーネルの利点には、仮想マシンと比較した場合、効率と可観測性の向上や、管理の簡素化が含まれます。

ゾーンのパフォーマンス上の利点

OSの仮想化には、ハイパーバイザーや、分離を実装するためのソフトウェアのその他のレイヤーは必要ありません。このようなソフトウェア・ハイパーバイザーは、さまざまな種類のパフォーマンス・オーバーヘッドの原因になる場合があります。これらには、通常は実際のワークロードを実行しているHWT上でハイパーバイザーが実行される時間、キャッシュ・フラッシングによる非効率性、およびハイパーバイザーがVMの代わりに実行する必要があるすべての操作（共有I/Oリソースを使用したI/Oなど）が含まれます。代わりに、Oracle Solaris Zonesでは、個々のプロセスがゾーン内で実行されているために別のコード・パスに従うことがないため、これらのパフォーマンス低下の影響は発生しません。

ゾーンのセキュリティ上の利点

ハイパーバイザーを使用する代わりに、ゾーン間の分離はOSカーネルによって管理されます。このカーネルはゾーンの分離境界を認識し、境界にまたがるアクションを防止します。これらのセキュリティ・チェックは、プロセスがゾーンまたはグローバル・ゾーン（Solarisが最初に起動されたときに実行される環境およびプロセスのセット）のどちらで実行されているかにかかわらず実行されます。この方法により、Oracle Solaris Zonesは、汎用オペレーティング・システムによって達成されるもっとも高いセキュリティ評価を、仮想化によるパフォーマンス低下を招くことなく実現できます。

ゾーンのトラブルシューティング上の利点

すべての（ノン・グローバル・）ゾーンはシステムのグローバル・ゾーンから管理されるため、十分な権限を持つグローバル・ゾーン・ユーザーが各ゾーンとそのコンテンツを制御できる必要があります。ゾーン内のプロセスは、ある意味では1つのカーネルによって管理される1つのプロセス・セットであるため、権限を持つグローバル・ゾーン・ユーザーは、すべてのゾーン内のすべてのプロセスを表示できます。これはゾーンの大きな利点です。システムの問題をトラブルシューティングする段階になったら、通常のSolarisツールを使用して、問題の原因となっている（任意のゾーン内の）プロセスを特定できます。最初に問題の原因となっているゲストを特定してからそのゲストにログインして、問題の原因となっているプロセスを特定する必要はありません。後者の方法を使用する必要があるのは、ハイパーバイザーベースの仮想マシン環境をトラブルシューティングする場合です。

ゾーンは通常、多数のワークロードを1つのSolarisインスタンスに統合するために使用されます（ただし、必ずしもそうとは限りません）。そのインスタンスは、コンピュータ全体（SPARCまたはx86）、エンタープライズ・クラスのSPARCサーバー上のハードウェア・ドメイン、現在のいずれかのSPARCサーバー上の論理ドメインなどである場合があります。このいずれの場合も、各ゾーン内のワークロードのHWTの使用は、Solarisスケジューラおよびカーネルのその他の部分によって管理されます。Solarisスケジューラは、多数のHWTに対応して拡張できるように定期的に改良されてきたコンポーネントの1つです。

デフォルトでは、あるSolarisインスタンスのゾーンのすべてのプロセスを、理論的には、そのSolarisインスタンスから使用できる任意のコア上で実行できます。その構成では、スケジューラによって、HWTやメモリの局所性を維持するという優れたジョブが実行されます。つまり、スケジューラは、プロセスにタイム・スライスが割り当てられるたびに、同じプロセスを同じコア上で実行することによってコアのキャッシュの利点を最大化しようとします。また、そのプロセスが通常実行されるHWTによってアクセスされると、待機時間がもっとも短い物理メモリ・ブロックを割り当てることによってメモリ待機時間も最小限に抑えようとします。

コアの量を増やしたことによる直線的なパフォーマンスの向上や、ゾーンのオーバーヘッドなしの効率性というSolarisの歴史を考慮すると、ゾーンがSolarisのようにリニアにスケールできることは驚くべきことではありません。デフォルトでは、Solarisスケジューラは、異なるゾーン内のすべてのプロセスを均等に処理します。実際、スケジューリングの決定を行うときに、デフォルトでは、プロセスとゾーンの関連付けを考慮しません。一方、選択の対象となるリソース制御がいくつか存在するため、これらのツールの使用によってスケジューリングを予測可能な方法に変更できます。

どのコンピューティング環境でも、予測可能なパフォーマンスが提供されるべきです。あるワークロードで今週、1秒あたり100トランザクションが実行された場合、そのユーザーは、次週も同じパフォーマンスが提供されることを期待します。シングル・ワークロードの環境では、安定したシステムによって、その予測可能なパフォーマンスが提供されます。ただし、マルチテナント環境では、HWTの容量やRAMなどのリソースの消費を管理する必要があります。複数ワークロードの環境の予測可能なパフォーマンスを提供するには、リソース制御が必要です。

Solarisは実行中のワークロードにリソースを公平に割り当てようとしますが、管理されていないワークロードが、他のワークロードが適切なパフォーマンスで実行されないほどのリソースを消費する場合があります。これは新しい問題ではなく、Solarisでは、市場に登場してから20年間にわたって徐々にリソース制御が追加されてきました。

Solarisには、次のリスト内の項目を含む多くのリソース制御があります。Solarisゾーンには、これらのすべてを適用できます。

- HWTの容量：プロセッサ・セット、HWTキャップ、複数のスケジューリング・アルゴリズムおよび優先順位
- RAMの使用：RAMキャップによって、ゾーンのプロセスが使用できるRAMの量が制限される
- 仮想メモリ：VMキャップによって、ゾーンのプロセスが使用できるVMの量（RAMおよびスワップ領域）が制限される

Solarisは、CPUをカウントする場合や、HWTに対してプロセスをスケジュールする場合に、プロセッサ内の各ハードウェア・スレッドを"CPU"と見なします。現行世代のSPARCシステムは、非常に多数のハードウェア・スレッドを備えています。これにより、スケジューラがスケジューリングの決定を行ったり、ユーザーがワークロードに割り当てるHWTの量に関して決定したりするとき（その方法がコンピュータを共有する一連のワークロードに適していると判断した場合）の柔軟性が向上しています。

次の例について考えてみます。T5-4は、4基のプロセッサ、64個のコア、および512個のハードウェア・スレッドを備えています。[6]ユーザーはそのT5-4システム上で数百のSolarisゾーンをきわめて容易に作成して実行でき、さらにリソース制御をまったく使用しなくても、これらのゾーンが非常に適切に実行される可能性があります。ただし、例えば30ワークロードのセット（1つのゾーンに1つずつ）には、特定のパフォーマンス要件を持つ少数のワークロードが含まれる可能性があります。この場合にSolarisのリソース制御を使用すると、1つの本番データベース・ゾーンには8つのコアの1つのセットを、4つの本番アプリケーション・サーバーには8つのコアの4つのセットを割り当てるが、ゾーンの他の部分では残りの24個のコアを共有したままにすることができます。

オラクルは、ターゲット・コンピュータ内のコアの量に基づいて、そのソフトウェアのライセンス費用を請求します。ライセンス・ポリシーにより、"ハード・パーティショニング"が使用されている場合、顧客はライセンスに必要なコアの量を制限できます。このホワイト・ペーパーの執筆時点では、プロセッサ・セット（つまり、リソース・プール）を使用するOracle Solaris Zonesは"ハード・パーティショニング"と見なされるため、これらの"制限付きコンテナ"を使用してライセンス費用を削減できます。ただし、パブリック・ライセンス・ドキュメントで説明されている"制限付きコンテナ"はプロセッサ・セットを使用し、Solarisゾーンの"制限付きCPU"機能は使用していないことに注意してください。[19]

Oracle VM Server for SPARC

Oracle VM Server for SPARC（旧称"Sun Logical Domains"）は、Oracle SPARCサーバー上でサポートされる仮想化テクノロジーです。Oracle VM Server for SPARCでは、サーバー・プロセッサ・リソースをパーティション化して個別の仮想マシン（"ドメイン"とも呼ばれます）に割り当てることができます。

Oracle VM Server for SPARCは、オーバーヘッドを削減する斬新なアーキテクチャを備え、多数のハードウェア・スレッドを含むシステムに最適な機能になっています。従来のハイパーバイザーは、複数の"仮想CPU"をより少数の物理HWTに仮想化します。これにより、ソフトウェアベースのタイム・スライシングや、ゲスト・オペレーティング・システムによる"権限命令"の使用をエミュレートする必要性によるオーバーヘッドが発生します。これは、共有されているHWTの物理状態（メモリ・マッピングや割込みステータスなど）をゲストが変更できないようにするために必要です。ゲストOSは、特定の権限命令を実行しようとするたびに、トラップしてハイパーバイザーにコンテキスト切替えするか、または命令のバイナリ書換えを実行する必要があります。これによって大きなオーバーヘッドが発生しますが、これが40年以上にわたる仮想化の従来のモデルでした。

これに対して、Oracle VM Server for SPARCは、多数のHWTを含むシステム（1,024個のHWTを備えたT5-8など）のために最適に設計されています。従来のハイパーバイザーとは異なり、ハードウェア・スレッドやメモリは、論理ドメインに*直接*割り当てられます。その論理ドメインで実行されているゲストOS（Solaris）は"ベアメタル"で実行され、ハイパーバイザーにコンテキスト切替えしなくても、上記の権限操作を直接実行できます。このおもな差別化要因は、従来のモデルとは異なり、論理ドメイン・モデルにはHWTの"仮想化の重い負担"やメモリのオーバーヘッドが課せられない理由を示しています。ゲスト・ドメインは、独自の専用ハードウェア・スレッド上で、完全な"ベアメタル"パフォーマンスで実行されます。

[6] 大まかな比較として、このようなシステムの処理能力は、1999年の20台のUltraSPARC E10000に勝るとも劣りません。

もう1つの違いは、従来のVMシステムが、管理制御ポイント、リソース・マネージャ、ユーザー・インタフェース、およびデバイス・ドライバを仮想マシン提供のタスクと結合したモノリシック・ハイパーバイザーに依存している点です。これに対して、Oracle VM Server for SPARCのアーキテクチャはモジュラー型であり、次の主要コンポーネントで構成されています。

1. リソース（HWT、メモリ、および必要に応じてI/Oデバイス）を論理ドメインに直接割り当てるファームウェア・ベースのハイパーバイザー。
2. コマンドライン・インタフェース（"CLI"）、またはEnterprise Manager Ops Center 12cやOracle VM Manager 3.2以降などのオプションの管理スイートを通して管理制御ポイントとして機能する"制御ドメイン"。
3. “サービス・ドメイン"内で実行され、ゲスト・ドメインに仮想I/Oサービスを提供する、ソフトウェアベースのI/O仮想化。制御ドメインは通常、この目的に使用されます。冗長性のために、仮想I/Oを提供する追加のドメインが存在する場合があります。I/OデバイスやPCIバスはドメインに直接プロビジョニングできるため、これはオプション・コンポーネントです。

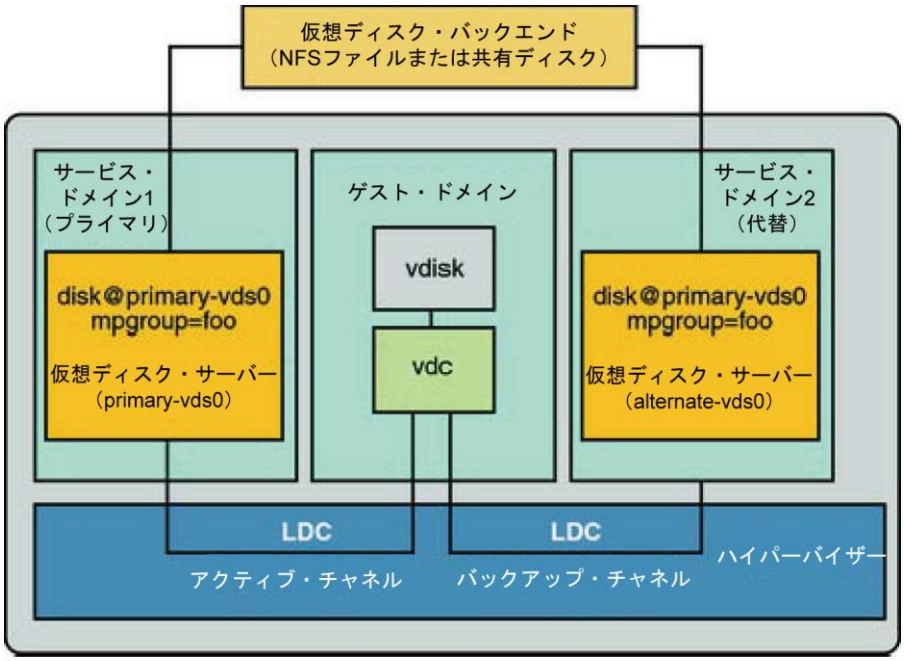

図6：冗長なサービス・ドメインを含むゲスト論理ドメイン

Oracle VM Server for SPARCによって、以下が可能となります。

- **トップクラスの価格性能比** オーバーヘッドを低く抑えたアーキテクチャによって、ワークロードが増加してもライセンス・コストを追加することなく、スケーラブルなパフォーマンスを実現します。
- **セキュアなライブ・マイグレーション** ゲストが引き続き動作し、ユーザーにアプリケーション・サービスを提供できるようにしたまま、アクティブ・ドメインをある物理SPARCサーバーから別の物理SPARCサーバーに移行できます。メモリの内容はソース・サーバーから移行先サーバーに転送される間、常に圧縮および暗号化され、それによってセキュリティとプライバシが保証されます。S3コア内のオンチップ暗号化アクセラレータによって、ハードウェアへの投資を追加しなくても、ライブ・マイグレーション

のためのセキュアな、ワイヤースピードの暗号化が提供されます。

- **シングル・ルートI/O仮想化（SR-IOV）** 認定されたPCIeデバイスを含むゲスト・ドメインに優れたI/Oスループットを提供します。このようなデバイスは、それぞれをドメインに直接割り当てることができる個別のI/Oデバイスであるかのように提供できます。SR-IOVを使用すると、PCIeネットワーク・デバイスを効率的に共有できるため、アプリケーションはネイティブなI/Oパフォーマンスを実現できます。
- **動的再構成（DR）** コンピューティング・リソースをアクティブ・ドメインに動的に追加したり、削除したりすることができます。アクティブ・ドメイン上のプロセッサ、仮想I/O、およびメモリに対するリソース割当ての変更を行うことができます。これらの機能によって、組織では、ITとビジネスの優先順位を効果的に調整できます。
- **高度なRAS** 各論理ドメインは、VM環境では予測される、独自のOSを持つ完全に独立した仮想マシンです。論理ドメインは、仮想ディスクのマルチパスとフェイルオーバーに加えて、IPマルチパス（IPMP）サポートによるネットワーク・フェイルオーバーもサポートしています。また、I/Oドメインとストレージ間のパス障害の処理も可能です。仮想I/Oが使用されている場合は、複数のサービス・ドメインを使用して、クライアント・ゲストに仮想I/Oをプロビジョニングできます。これによってデバイスとパスの冗長性が提供され、またサービス中断のないローリング・アップグレードも可能になります。あるサービス・ドメインが保守のために停止されていると、仮想I/Oはアクティブなサービス・ドメイン経由でルーティングされます。ドメインはSolaris FMA（障害管理アーキテクチャ）と完全に統合されているため、予測的自己修復が可能になります。
- **コア全体の割当てとコアおよびメモリ・アフィニティ** ゲスト・ドメイン間のキャッシュ偽共有を回避することによってパフォーマンスの向上が可能になり、HWTとRAMの間のレイテンシに対するNUMA（Non-Uniform Memory Access）の影響も軽減されます。これにより、組織はすべてのワークロードのより高く、かつ予測可能なパフォーマンスを実現できます。
- **HWTの動的リソース管理（DRM）** リソース管理ポリシーとドメイン・ワークロードにより、ドメインのリソース要件と優先順位に基づいたHWTの自動的な追加および削除を開始できます。ポリシーを設定することによって、指定されたしきい値を超えるHWT使用率が割り当てられている限り、事前に設定された最大数までドメインにHWTを追加するか、または逆に、ドメインからほとんどアイドル状態であるHWTを削除できます。どのドメインにHWTへのアクセスを優先的に与えるかを確立するための優先順位を指定できます。これは、ITとビジネスの優先順位を効果的に調整するのに役立ちます。
- **物理から仮想への（P2V）変換** Solaris 8、Solaris 9、またはOracle Solaris 10 OSを実行する既存のSPARCサーバーを、仮想化されたOracle Solaris 10イメージにすばやく変換します。ドメインはまた、ブランド・ゾーンをホストすることもできます。ここでは、Solaris 11 OSがSolaris 11ゾーンとSolaris 10ゾーンの両方をホストするか、またはSolaris 10 OSがSolaris 10ゾーンをSolaris 9およびSolaris 8ブランド・ゾーンとともにホストします。

- **サーバーの電力管理** すべてのハードウェア・スレッドがアイドル状態にある各コアを無効にするか、またはHWTのクロック速度やメモリの消費電力を調整することによって省電力を実装します。

SPARC Solaris環境の仮想化に関するさらに詳細なガイダンスについては、[22]およびOracleホワイト・ペーパー『*仮想化SPARCコンピューティング環境を構築するためのベスト・プラクティス*』[23]を参照してください。

Oracle Enterprise Manager Ops Center

Oracle Enterprise Managerは、物理および仮想サーバーの完全なライフ・サイクル管理のための、費用効果に優れた統合されたソリューションを提供します。これにより、包括的なプロビジョニング、パッチ適用、監視、管理、および構成管理機能がWebベースのユーザー・インタフェース経由で提供されるとともに、Oracle VM、Linux、Unix、およびWindowsオペレーティング・システム環境の管理に関連した複雑さとコストが大幅に削減されます。また、ITリソースを最適化し、ハードウェア使用率を向上させ、ITプロセスを効率化し、さらにコストを削減するための仮想化およびクラウド・コンピューティングの採用の促進にも役立ちます。

Oracle Enterprise Manager Ops Center 12cは、オペレーティング・システム/ファームウェア/BIOSの構成、ベアメタルと仮想マシンのプロビジョニング、ハードウェアの障害分析、My Oracle Supportサービス・リクエストの自動生成、およびパフォーマンスの管理のための包括的なソリューションを提供します。

Oracle Enterprise Manager Ops Center Virtualization Management Packは、リソース管理やモビリティ・オーケストレーションを含む、仮想ゲストの完全なライフ・サイクル管理機能を提供します。この機能は、顧客が業務を効率化し、停止時間を短縮するのに役立ちます。Virtualization Management PackとOps Center Provisioning and Patch Automation Packが組み合わされて、物理および仮想システムのためのエンド・ツー・エンドの管理ソリューションが単一のWebベース・コンソール経由で提供されます。このソリューションは、仮想システムのライフ・サイクル管理を自動化するものであり、オラクルのSunインフラストラクチャのためのもっとも効果的なシステム管理ソリューションです。

Solaris 11のスケーラビリティ

SPARCマルチコア・プロセッサのどのような高度な機能でも、オペレーティング・システムがそれを利用できなければそれほど役に立ちません。また、プロセッサ・コアやハードウェア・スレッドの数が劇的に増加するに従って、巨大な物理および仮想メモリとページ・サイズを管理し、数千のプロセスとソフトウェア・スレッドをスケジュールし、さらに多数の物理および仮想デバイスを処理するためのOSアルゴリズムをすべて、スケーラブルにする必要があります。

オラクルのSolarisエンジニアはまた、基盤となるハードウェアの進化に伴ってOSのパフォーマンスとスケーラビリティを最大化するために、DTrace [6]という強力なカーネル可観測性ツールを使用してSolarisのすべての側面を検証して改善します。たとえば、相互に作用するソフトウェア・スレッドの数が増加するに従って、ロック競合が大きな問題になる場合があります。Solaris 11のロック管理アルゴリズムでは、ハードウェア・スレッドの数に応じて直線的に拡張できるようになりました。さらに、Solarisの仮想メモリ・システムは、非常に大きなページをより適切に処理したり、ページの置換えを予測して最適化したりするだけでなく、カーネルとユーザー・メモリの両方を関連プロセスが実行されているプロセッサ・コアに近い場所に割り当てることによってメモリ・レイテンシを大幅に短縮するように機能強化されています。

/etc/system内のカーネル・チューニング可能値（特に、maxusers、max_nprocs、およびpidmax）は、使用可能なハードウェア・スレッドの数とシステム上のメモリに応じて自動的に拡張できるようになりました。Solaris 11.1は現在、100万個を超えるソフトウェア・スレッドをサポートしています。

結論

このホワイト・ペーパーでは、パラレル・プログラミングの多くの主要概念と手法について説明しました。アプリケーションのマルチプログラミングおよびマルチスレッド化の大きな価値といくつかの制限事項を強調し、特にオラクルの新しい大規模マルチスレッド化T5およびM5プロセッサの利点に重点を置きました。また、開発者がアプリケーション内のパラレル化を見つけて活用できるように支援する、OpenMP、Solaris Studioコンパイラ、Oracle Solaris Studioパフォーマンス・アナライザなどの重要な開発者ツールについても詳しく説明しました。

SPARCプロセッサによってマルチスレッド・アプリケーションがどのように拡張されるかの説明に加えて、オラクルの仮想化テクノロジーであるOVM for SPARCおよびOracle Solaris Zonesによって、システム・マネージャやITアーキテクトが複数のアプリケーション環境を単一のサーバーにどのように効率的に統合できるようになるかについても説明しました。

要約すると、オラクルの新しいT5およびM5プロセッサとそれらを使用して構築されたサーバーは、アプリケーションとシステムの両方のパラレル化を活用するための、従来にはないスケーラビリティと機会を提供します。この証拠は、これらのシステムが達成した多くのコンピューティングの世界記録[21]だけでなく、さまざまなシナリオでのこれらのシステムの成功した幅広い展開にも見られます。

用語集

API

APIは"アプリケーション・プログラミング・インタフェース"を意味します。これは、ソフトウェア・コンポーネントの対話方法を指定します。APIは通常、（オプションの）特定のデータ構造とオブジェクト・クラスを持つライブラリを提供します。このライブラリ内の関数によって、ユーザーが利用できるツールキットが提供されます。パラレル・プログラミングでの例として、POSIXスレッド、Javaスレッド、OpenMP、およびMPI（"Message Passing Interface"）があります。

不可分操作

これは、パラレル・プログラミングで使用される低レベルの操作です。これが"不可分"と呼ばれるのは、いくつかの関連するアクションが分割できない単位として機能するためです。

残念ながら、不可分操作はPthread APIの一部ではありませんが、不可分操作に使用できる多数の（移植不可能な）APIが存在します。OpenMPでも、不可分操作のサポートが提供されています。

バリア

これは、最後のスレッドが到着するまでどのスレッドも処理を続行できないパラレル・アプリケーション内のポイントです。バリアが非常に役立ち、かつ必要であることは間違いありませんが、注意して使用してください。スレッドが追加されるに従ってコストが増大し、これによってパフォーマンスが急速に低下することがあります。

キャッシュ

これは、アクセス時間がメイン・メモリより短いメモリ・バッファです。データと命令に使用されます。目的は、頻繁に参照されるデータ（または命令）をキャッシュ内に保持することによって、アクセスを大幅に高速化することにあります。最新のマイクロプロセッサには複数レベルのキャッシュがあります（通常、L1、L2などで示されます）。この表記は、L1キャッシュへのアクセスがL2キャッシュより高速であることを示しています。L2キャッシュとL3キャッシュの関係も同様です。統合キャッシュには、データと命令の両方が格納されます。

キャッシュ・コヒーレンシ

これは、共有メモリのパラレル・システムの設計のきわめて重要な部分です。システム内のさまざまなキャッシュ内に同じキャッシュ・ラインの複数のコピーが存在するということが容易に発生するため、同じキャッシュ・ラインのシステム全体にわたる一貫性を維持する必要があります。キャッシュ・コヒーレンシはこれを処理します。この目的のため、各キャッシュ・ラインに状態ビットを割り当て、各ラインのステータスを示すために使用します。キャッシュ・コヒーレンシ・プロトコルは、特定のアクションの結果としてキャッシュ・ラインがどのように状態を変化させるかを規定します。たとえば、あるライン内の要素が変更された場合は、変更されたラインの状態ビットを変更する必要があるだけでなく、同じラインの他のコピーをすべて無効にする必要があります。これは、それらのラインの状態ビットをそれに応じて変更することによっ

て達成されます。

コンテナ

「ゾーン」を参照

CPU

これは、プロセッサの演算コンポーネント（ALU）を指す古い用語である"中央処理装置"を意味します。最新のプロセッサは、それぞれに複数の"ハードウェア・スレッド"（従来の"CPU"に相当します）を含む多数のコアで構成されています

HWT

ハードウェア・スレッドであり、CPUという従来の概念に相当します。最新のプロセッサは、それぞれに複数のHWTを含む複数のコア・ユニットで構成されています。これは、一部のSolarisツール（psrinfoコマンドなど）では、"ストランド"または"仮想プロセッサ"とも呼ばれます。

マルチプログラミング

1つのHWT（またはCPU）上で複数のプロセスを実行する手法。オペレーティング・システムは、各プロセスを限られた時間だけ実行されるようにスケジュールしてから、別のプロセスに切り替えます。

マルチスレッド

これは、パラレル・アプリケーションの別の言い方です。このようなプログラムは、"マルチスレッド"と呼ばれます。つまり、このコードには、複数のスレッドを作成して実行するためのソフトウェア・インフラストラクチャが含まれています。

パラレル化

独立しているため、結果の正確性に影響を与えずに任意の順序で実行できるアプリケーション内の各部分を識別するプロセス。これに該当する場合は、それらの部分を同時に、つまり"パラレルで"実行することもできます。このようなプログラム・セクションが識別されたら、このタイプの同時実行を実装するための適切なパラレル・プログラミング・モデルを選択する必要があります。パラレル化を表す方法は、選択されたモデルに大きく依存します。

パラレル・プログラミング・モデル

これにより、アプリケーション内のパラレル化を実装するためのソフトウェア・ツールボックスが提供されます。また、どのような種類の機能を使用できるかも定義されます。単一システムとクラスタの両方について、使用可能な多くのモデルが存在します。通常、パラレル・プログラミング・モデルはスレッドの作成、作業のパラレルでの実行、データの交換、スレッドの同期化、およびパラレル実行の終了のための方法を提供します。

プラグマ

これにより、C/C++コンパイラに追加情報を渡すための移植可能な方法が提供されます。プラグマは"#pragma"キーワードで始まり、その後に1つ以上の修飾子が続きます。このような修飾子は、コンパイラ固有であるか、またはOpenMPの場合のようにプラットフォーム間で移植可能かのどちらかです。

コンパイラはサポートしていないプラグマを無視するため、プラグマが移植性に影響を与えることはありません。そのため、移植性に違反することなく、コンパイラに特定の情報を渡すことができます。

OpenMPは、このモデルに基づいて構築されています。すべての言語構文が"#pragma omp"で始まります。コンパイラがこれらを認識するよう通知されていない場合でも、OpenMPをサポートしていないコンパイラを使用してソース・コードをそのまま構築できます。これらのプラグマは無視されるため、適切な#ifdefを使用してランタイム関数を処理できます。

プロセッサ・パイプライン

各命令は、さまざまな段階を通過します。たとえば、フェッチ段階では、命令が命令キャッシュからフェッチされます。次に、デコード段階でデコードされます。これらの段階がまとまって、パイプラインが形成されます。ハードウェアは、命令をある段階から次の段階に自動的に進めます。命令はパイプラインを通して流れますが、特定の段階のコストが変動することがあります。データ検索段階のように、データの場所によっては非常に迅速であったり、はるかに長い時間（多くのクロック・サイクル）がかかったりします。パイプライン内のこれらの遅延は、"ストール"または"バブル"とも呼ばれます。

RAS

これは、信頼性、可用性、および保守性を意味します。

共有データとプライベート・データ

共有メモリのパラレル・プログラムでは、共有とプライベートという2つの基本的なデータの種類が存在します。共有データは、すべてのスレッドからアクセス可能です。各スレッドは、このようなデータの読取りまたは書込みを実行中の任意の時点で行うことができます。これが正しい方法で実行されることを保証するのは開発者の責任です。選択されたパラレル・プログラミング・モデルに応じて、そのための特定の方法が存在します。

プライベート・データは、それを所有するスレッドからのみアクセス可能です。変数名が同じ場合でも、各スレッドは互いのプライベート・データの読取りまたは書込みを行うことができません。これが必要な場合は、このようなデータを共有バッファにコピーする必要があります。

OpenMPでは、関連するプラグマで"shared"句と"private"句を使用して、変数にそれに応じたラベルを付けることができます。

スレッド

このホワイト・ペーパーの文脈では、多くの場合、ソフトウェア・スレッドを単に"スレッド"と呼んでいます。これにより、実行のポイントを追跡するための独立したプログラム・カウンタ（PC）を持つ命令のシーケンスを意味しています。同時に複数のスレッドを実行でき、オペレーティング・システムはハードウェアに対して複数のスレッドをスケジュールします。T4、T5、およびM5システムの場合、Solarisはまず実行するコアを選択し、次にこのコア内のハードウェア・スレッド（"仮想プロセッサ"）を選択します。

TLB

これは、"Translation Look-Aside Buffer"を意味します。仮想メモリ・アドレスと物理メモリ・アドレスの間のマッピング情報を格納する特殊な種類のキャッシュです。命令（"I-TLB"）とデータ（"D-TLB"）用に個別のTLBを持つことができます。

ゾーン

ゾーン仮想化テクノロジーは、オペレーティング・システム・サービスを仮想化したり、実行中のアプリケーションのための分離されたセキュアな環境を提供したりするために使用されます。

参考資料

1. "ムーアの法則"、Wikipedia.org
2. "SPARC T5 Deep Dive: An interview with Oracle's Rick Hetherington"、Oracle.com
   "SPARC T5 の何がスゴイのか？　～開発責任者リック・ヘザリントンに聞く～"、 Oracle.com
3. "SPARC T5: 16-core CMT Processor with Glueless 1-Hop Scaling to 8-Sockets"、HotChips 24でのSPARC T5のプレゼンテーション
4. Oracle Engineered Systems、Oracle.com
5. Oracle Optimized Solutions、Oracle.com
6. DTrace、Oracleドキュメント・サイト、Oracle.com
7. Gregg、Brendan、およびJim Mauro、*DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*、Prentice Hall、2011年
8. Oracle Solaris Studio、Oracle.com
9. Combs, Gary、*Oracle's SPARC T5-2, SPARC T5-4, SPARC T5-8, and SPARC T5-1Bサーバーのアーキテクチャ*、Oracleホワイト・ペーパー、2013年3月
10. Combs, Gary、*SPARC M5-32 Server Architecture*、Oracleホワイト・ペーパー、2013年3月
11. *Oracle SPARC T4-1、SPARC T4-2、SPARC T4-4およびSPARC T4-1Bサーバーのアーキテクチャー*、Oracleホワイト・ペーパー、2011年10月
12. van der Pas, RuudおよびJared Smolens、*SPARC T4プロセッサのスループット最適化：ケース・スタディ*、Oracleテクニカル・ホワイト・ペーパー、2012年4月
13. "POSIXスレッド"、Wikipedia.org
14. OpenMP.org
15. Chapman、Barbara、Gabriele Jost、およびRuud van der Pas、*Using OpenMP: Portable Shared Memory Parallel Programming*、MIT Press、2007年10月
16. van der Pas, Ruud、*Parallel Programming with Oracle Developer Tools*、Oracleホワイト・ペーパー、2010年5月
17. *Oracle Solaris Studio OpenMP User's Guide*、Oracle.com
18. Amdahl, Gene M. 、*"Validity of the single processor approach to achieving large scale computing capabilities"*、AFIPS Spring Joint Computer Conferenceでの発表、1967年
19. Oracle Partitioning Policy 、http://www.oracle.com/us/corporate/pricing/partitioning-070609.pdf
20. Gove, Darryl 、*Multicore Application Programming: for Windows, Linux, and Oracle Solaris*、Addison-Wesley Professional、2010年11月9日
21. Oracleプレス・リリース：Oracle Unveils SPARC Servers with the World's Fastest Microprocessor、Oracleプレス・リリース、2013年3月26日
22. Victor, Jeff他、*Oracle Solaris 10 System Virtualization Essentials*、(Boston: Pearson, 2010)、169～226ページ

23. *仮想化SPARCコンピューティング環境を構築するためのベスト・プラクティス*、
http://www.oracle.com/technetwork/jp/oem/host-server-mgmt/hghlyavailldomserverpoolsemoc12cv09-1845483-ja.pdf

READ_ME_FIRST：多数のSPARCスレッドの使用方法
2013年8月

著者：Monica Riccelli、Frances Zhao-Perez
共著者：Ruud van der Pas、Jeff Savit、Jeff Victor、Darryl Gove、およびHarry Foxwell
Redwood Shores, CA 94065
U.S.A.

海外からのお問い合わせ窓口：
電話：+1.650.506.7000
ファクシミリ：+1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

0613

Hardware and Software, Engineered to Work Together