並列アプリケーションを指向した
分散オペレーティングシステム Colonia

Colonia: a Distributed Operating System
for Efficient Execution of Parallel Applications

青木 秀貴 (*Hidetaka AOKI*)

平成 9 年 2 月 (*Feb. 1997*)

内容梗概

分散システムと集中システムを比較した時、分散システムには構成が柔軟で拡張性に優れるという特徴がある。しかし今日の分散システムではノード間の通信レイテンシが大きく、並列アプリケーションの効率的な実行を阻害している。

本論文では、並列アプリケーションの効率的な実行を支援する分散システムとして、コンピュータ・コロニーを提案する。コンピュータ・コロニーとは共有メモリをベースとした分散システムであり、OS を含めた総合的な環境である。コンピュータ・コロニーは (1) 物理的には分散していても構わないが、(2) 機能的には完全にネットワーク透過であり、(3) 並列アプリケーションの実行を支援する、という特徴を持つ。コンピュータ・コロニーでは、従来の分散システムにおける通信レイテンシの問題を、共有メモリベースの通信によって解決する。共有メモリをベースとした通信では、MMU によるメモリ保護を利用することで、通信機構をユーザに対して安全に解放することができる。個々の通信に対しては OS が介在しないため、高速・低遅延な通信が実現可能となり、その上で処理ノードが緊密に協調動作することができる。

コンピュータ・コロニーは、ハードウェアおよびソフトウェアによって達成される環境である。本論文では、コンピュータ・コロニーのハードウェア環境の実現例として、共有メモリベースのワークステーションクラスタを示す。共有メモリベースのワークステーションクラスタは、従来のワークステーションのプロセッサバスに、高速なリモートメモリアクセスを提供する通信ボードを付加することで実現される。これにより、従来からのハードウェア資産を生かして、高性能な並列アプリケーション実行のためのハードウェア基盤を安価に実現することができる。

コンピュータ・コロニーの計算機資源を管理し、並列アプリケーションの実行を支援するためのソフトウェア環境として、分散 OS Colonia を提案する。Colonia はハードウェアの提供する共有メモリベースの通信機構を OS レベルから積極的に利用し、並列アプリケーションの高速な実行と計算機資源の有効利用を目指す。その実現のため、プログラムの位置透過な実行と動的な負荷分散を重視している。

Colonia はマイクロカーネル方式の構成をとり、Colonia カーネルでプログラムの実行と通信のための基本機能を提供する。Colonia はミッション-ユニットモデルと呼ぶプログラミングモデルを採用し、それを可能にするための通信メカニズムとして、共有メモリとシグナルを提供する。

ミッション-ユニットモデルでは、ひとつのプログラムをミッションとし、その中で並列実行可能な処理単位をユニットに割り当てる。ユニットは、それぞれが独立したアドレス空間を持つアクティビティである。そのため、他のユニットによる資源の破壊の心配がなく、また不要なデータの共有による性能低下を避けることができる。ただしミッション内のユニットは、アドレス空間の一部を、そのアドレス位置も含めて共有し、そこにプログラムコードやその他の共有可能なデータを配置する。共有データはすべて同じアドレスに配置されるためポインタを使うことができ、またアドレス空間が単純になるためそれを生かしたユニットの高速な生成と移送が実現される。

ユニット間の通信プリミティブとしては、共有メモリとシグナルを提供する。通常のデータの授受はハードウェアに支援された高速・低遅延な共有メモリを介しておこなわれ、非同期イベントの伝達手段としてシグナルが用いられる。メッセージなどその他の通信機構は、サーバやライブラリの中で共有メモリとシグナルを組み合わせることで実現される。

Abstract

Distributed systems have the advantage of flexible and scalable formation to centralized systems. But distributed systems today have great communication latency, which blocks efficient execution of parallel applications.

In this thesis, we propose a computer colony, a distributed system which supports efficient execution of parallel applications. A computer colony is an integrated computer environment including shared-memory-based distributed system hardware and its operating system. It has 3 features: (1) physically distributed, (2) functionally network-transparent, and (3) support for parallel applications. In a computer colony, fast and low-latency communication in user level is achieved by its shared-memory-based communication mechanism where memory protection mechanism releases the operating system from its miscellaneous works of the shared resource protection. Thus, there are very few interventions of operating systems during a user level communication. Such a high-performance communication enables tight cooperation of each nodes and network-transparent execution of parallel applications.

As an implementation of computer colony hardware, this thesis presents a shared-memory-based workstation cluster, which is realized by adding communication boards that enable fast remote memory access to processor busses of ordinary workstations. You can get a hardware environment for high-performance execution of parallel applications at low price by utilizing hardware property of today.

This thesis also proposes a distributed operating system, Colonia, which manages hardware resources of computer colony and helps efficient execution of parallel applications. Colonia makes active use of shared-memory-based communication mechanism offered by hardware in order to realize efficient execution of parallel applications and to make good use of computer resources. Location-transparent execution of programs and dynamic load balancing are the keys to them.

Colonia adopts a microkernel approach. The Colonia kernel provides the basic functions for program execution and communication. Colonia adopts a programming model called the Mission-Unit model and provides the shared memory and the signal as communication mechanisms.

In the Mission-Unit model, a program is assigned to a mission, and execution units which can run in parallel are assigned to units. The unit is an activity that has its own address space. The separated address space protects data from destroyed by other units and alleviates performance degradation incurred by unnecessary data sharing. Several parts of the address space are assigned as shared region and each of them has the same address in any of units' spaces. Since a shared data is located at the same address in all units' spaces, the pointer can be used among units in a mission to point the same data. The simple address space image also enables fast creation and migration of a unit.

Colonia provides shared memory and signals for communication primitives. Ordinary data are transferred by fast and low-latency shared memory supported by hardware and asynchronous events is transferred by signal. Other communication mechanism, such as messages, are realized by combination of shared memory and signals in servers and libraries.

目 次

1 はじめに

近年、ワークステーションやパーソナルコンピュータの低価格化、高性能化、そして小型化が進み、オフィスにおいて個人がこれらのコンピュータを持つことが一般的になってきている。今や、一昔前の大型計算機に匹敵する計算パワーを、一人一人が持てるようになっているのである。

そして、ワークステーションやパーソナルコンピュータの普及に伴い、それらのコンピュータをLAN (Local Area Network)で結合した分散システムもまた広く利用されるようになってきている。物理的に分散した計算機資源をネットワークで結合することで、マシン間あるいはユーザ間で、データを共有したり、プリンタ等の周辺機器を共用したりすることが可能となる。正当なアクセス権限を有していれば、利用マシンに依らず誰もが自由にデータや周辺機器を利用できることの有効性は広く認められ、分散システムの普及に拍車をかけている。そして分散システムの普及は、さらなるワークステーションやパーソナルコンピュータの普及を促す形となっている。

このような形で分散システムは広く普及してきており、その規模も大きなものでコンピュータ数百台から数千台にも及ぶ。そこにはトータルで莫大な計算パワーが存在するにもかかわらず、現状ではそれらが有効に利用されているとは言えない。具体的には、ネットワーク上の計算機資源、特にプロセッサ資源やメモリ資源の利用率は一般にかなり低い。分散システム上のすべての計算機に負荷がかかっているような状況はあまり存在せず、また時間帯によってはほとんどの計算機資源がまったく利用されないままとなっているのである。

これらの遊んでいる計算機資源を有効に利用し、分散システムの持つ巨大な計算パワーを引き出すことによって、次の2点の達成が期待される。

- 並列処理による個々のアプリケーションの高速化 $\cdots$ 近い将来、高速化を目指して様々な分野のアプリケーションが並列化されると考えられる。その際、並列アプリケーション実行のプラットホームとして、分散システムの持つ大量のプロセッサ資源は十分に魅力的である。
- システム全体としての処理能力の向上 $\cdots$ システムワイドな負荷分散をおこなうことで、システム全体としての処理能力を向上することができる。個々のプログラムが並列化されていれば、並列化された処理の流れをシステム中に分散させることで、一層の負荷分散を図ることができる。また空き資源を利用することで、実行に何日もかかるような科学技術計算などの大規模ジョブを、通常アプリケー

ションの実行を阻害することなく処理させることも可能となる。

しかしこれらを実現するには、現状の分散システムではハードウェア, ソフトウェアとも問題がある。とりわけ通信の遅さは、その大きな障害となるものである。分散システム用に現在利用可能なネットワークでは、どれだけデータ転送速度の高いものであっても、通信時のレイテンシが非常に大きいという問題がある。そのため、要素マシン間での細粒度の協調動作は現実問題として不可能であり、さらには分散システムの利用形態そのものに大きな制約が課せられることになる。例えば、並列計算のプラットホームとして現状の分散システムはあまりに非力であり、マシン間をまたがる細粒度並列計算をおこなうことはできない。

そこで本論文では、分散システムを拡張しその利用範囲を広げた環境として、コンピュータ・コロニー (computer colony) を提案する。コンピュータ・コロニーとは並列アプリケーションを指向した共有メモリベースの分散システムであり、オペレーティングシステム (Operating System, OS) を含めた総合的な環境を指す。コンピュータ・コロニーでは、システムを構成する物理的には分散したノードが緊密に協調動作し、全体としてひとつの巨大な計算パワーを提供する。コンピュータ・コロニーはマルチユーザで利用され、各ユーザはその上で並列アプリケーションを効率よく実行することができる。

コンピュータ・コロニーの実現には、ハードウェアによる高速通信の支援と、その通信機構を積極的に活用する OS が不可欠である。本論文では、コンピュータ・コロニーのハードウェア環境の実現例として、共有メモリベースのワークステーションクラスタによるものを示す。また、コンピュータ・コロニーを構築するための分散オペレーティングシステム **Colonia** を提案する。

共有メモリベースのワークステーションクラスタとは、処理ノード間を共有メモリという汎用性の高い通信プロトコルによって結合したクラスタシステムである。本研究ではこの共有メモリベースのワークステーションクラスタを、分散システムと集中システムの中間に位置するものと考えている。ただし、コンピュータ・コロニーのハードウェア環境は様々な実現が可能であり、共有メモリベースのワークステーションクラスタはそのひとつに過ぎない。

コンピュータ・コロニーのソフトウェア環境として、ユーザレベルの共有メモリを実現し、マルチユーザ環境における並列アプリケーションの実行を支援するのが Colonia の役割である。本研究では、ハードウェアの提供する通信機構に対して OS レベルから積極的な支援をおこなう点が、ひとつの大きな特徴となっている。

本論文の構成は以下の通りである。まず第 2 章で集中システムと分散システムというふたつのマルチユーザ並列計算環境を比較し、第 3 章で並列アプリケーションを指向した分散システムとしてコンピュータ・コロニーを提案する。第 4 章でコンピュータ・コロニーのハードウェア環境の実現例として共有メモリベースのワークステーションクラスタについて述べ、第 5 章でコンピュータ・コロニーのソフトウェア環境である分散オペレーティングシステム Colonia について説明する。最後に第 6 章でまとめる。

2 背景

マルチユーザを想定した計算機環境は、大きく分散システムと集中システムに分けることができる。本章では将来の計算機環境として両者を比較するため、まずソフトウェアの流れについて述べる。続いてハードウェア環境としての両者を比較する。

2.1 ソフトウェアの潮流

今日のソフトウェアの潮流として、ふたつの大きなキーワードを挙げることができる。ひとつは「並列化」であり、もうひとつは「マルチメディア」である。

ソフトウェアの並列化　単一プロセッサにおける処理能力の限界を打ち破るべく、マルチプロセッサあるいはマルチコンピュータによる並列処理の研究が現在進められている。今日では並列処理技術の適用は、並列計算機 (マルチプロセッサ) の上で、科学技術計算やデータベース処理などの特定のプログラムを実行するといった程度に留まっている。しかしソフトウェアの高機能化・巨大化に伴い、将来的にはワープロソフト, 表計算ソフトなどの日常的なアプリケーションソフトや、CAD, コンパイラなどのエンジニアリング用途のソフトなども並列化され、マルチプロセッサあるいはマルチコンピュータなどの様々なプラットホームの上で並列実行されるようになると期待される。

マルチメディア　情報伝達の手段としてあるいはユーザインタフェースとして、マルチメディアはもはや必要不可欠な技術となっている。マルチメディア処理の本質は、リアルタイム処理である。そのため、コンピュータ利用のあらゆる局面において、リアルタイム性の確保が求められるようになってきたと言える。また、マルチメディアでは大量のデータを扱うため、十分なバンド幅を持ったネットワークも欠かすことができない。

このように、並列化とマルチメディアへの対応は今後のソフトウェアに欠かすことのできない流れであり、コンピュータ環境の構築の上で見逃すことのできない存在である。

2.2 分散システムと集中システム

第 2.1 節で述べた通り、ソフトウェアの並列化は今後一般的なものになると考えられる。そのため、並列化されたアプリケーションの処理基盤として、ハードウェアの提供する並列処理能力は重要になってくる。

並列処理のプラットホームとしては、SMP (Symmetrical Multiprocessor, 対称型マルチプロセッサ) 機やクラスタシステム、そして超並列機がある。ここで、ユーザにとって真に使いやすい並列処理環境を考える場合、マルチユーザへの対応は欠かすことができない。マルチユーザのための計算機環境は、分散システムと集中システムに分けることができる。ここでは、それぞれ次のようなものを想定している。

- 分散システム … 1 台 1 台の性能はさほど高くない複数のワークステーションやパーソナルコンピュータを、LAN などのネットワークで結合した環境。各要素マシン (ノード) は、シングルプロセッサでも SMP でも構わない。
- 集中システム … 巨大な計算パワーを有する 1 台の大規模 SMP 機や超並列機を中心に構成され、端末を通じて複数ユーザが利用する環境。なお本論文では以下、大規模 SMP 機と超並列機をまとめて単に並列計算機と呼ぶ。

以下、分散システムと集中システムというふたつの計算機環境の比較をおこなう。

分散システムの利点　分散システムの最大の利点は、その柔軟な構成と優れた拡張性にある。並列計算機でもプロセッサ数の増減を可能にすることはできるが、分散システムは構成がより自由であり、要素マシン 1 台からの利用も可能である。ユーザは必要に応じてマシンを増設し、それによって段階的にシステ

ムの高性能化を図ることが可能である。また、要素マシンが十分に互換性を意識して設計されていれば、要素マシンを少しずつ最新の上位機種に交換していくことも可能である。

また分散システムでは、製品価格を抑えることが可能である。分散システムを構成する要素マシンは基本的にすべて同じ構造をしており、要素マシンの大量販売が見込まれる。さらには、システムを拡張するためのモジュールとして、あるいは単体でも動作可能な通常のワークステーションやパーソナルコンピュータとして、要素マシンのさらなる販売を見込むことができる。この結果、量産によって製品価格を抑えることが可能となる。

ユーザから見ても、分散システムの場合、最初は小規模なシステムを低価格で導入し、必要に応じてシステムを拡張することが可能である。その価格も含めて、分散システムはユーザにとって魅力的である。

これに対し、並列計算機ではどうしても価格が高くなり、最小構成のものであっても購入できるユーザは非常に限られてしまう。

集中システムの利点　一方、並列計算機を中心とした集中システムの利点としては、プロセッサ間の通信が非常に高速であることが挙げられる。

分散システムにおける要素マシン間の通信をどれだけ高速にしても、並列計算機におけるプロセッサ間通信はさらに高速である。並列処理やその他の協調処理をおこなう上で、通信の性能は重要なポイントである。通信性能は並列計算機の方が優れているため、並列化されたアプリケーションの実行については、集中システムは分散システムよりも高い処理能力を発揮する。これは特に、細粒度並列処理において顕著となる。

分散システムと集中システムを比較した時、両者の違いの本質は「拡張性」と「高速通信」のトレードオフにあるといえる。すなわち、分散システムは高速通信を犠牲にして拡張性を得たものであり、集中システムはその逆に拡張性を犠牲にして高速通信を達成するシステムである。

2.3　現状の分散システムの問題点

以上、分散システムと集中システムを比較してきたが、将来の計算機環境の構築にあたり、分散システムをベースとするのは、非常に現実的であるし、また悪くない選択肢である。価格と拡張性は、ユーザが決して無視することのできない問題である。並列計算機を中心とした集中システムでは導入の初期コストが高く、しかも後々の拡張性にも問題があるため、ユーザがなかなか購入に踏み切れないという状況は、将来も続くだろう。それに対して分散システムならば、安価な要素マシン1台からの利用が可能であり、必要に応じてシステムを拡張・更新していくことが可能である。分散システムによってユーザは、並列アプリケーションの実行に必要な大量のプロセッサ資源を、手軽に導入することができるのである。

分散システムは現在既に普及してきてはいるのだが、それらはネットワークOSの管理下でデータや周辺機器の共有に利用されているに過ぎず、巨大な計算パワーを生かした「並列アプリケーションの効率よい実行」や「システム全体としての処理能力の向上」は達成されていない。その原因としては、大きく次のふたつを挙げることができる。

- 現状の分散システムでは、通信のコストが高い。
- 現状の分散システムでは、計算パワーの位置透過性が実現されていない。

通信のコストが高い　現状の分散システムでは通信の遅さがボトルネックとなり、並列処理などノード間での緊密な協調を必要とする処理を効率よく実行することは難しい。これは、並列計算機と比べた際の分散システム最大の弱点である。現状の分散システムにおける通信の遅さは、以下の理由による。

現在一般に利用されている分散システムでは、各マシンのネットワークインタフェースが低速なI/Oバスに接続されている。これにより、

- I/Oバスがデータ転送のボトルネックとなる。そのため、ネットワーク自体のデータ転送能力がどれだけ高くても、その性能が十分に利用できない。
- 資源の保護のため、通信時には必ずOSが介在することになる。そのため、データコピーやコンテキストスイッチングのオーバヘッドが生じる。

という大きな問題が生じることになる。

また汎用のネットワークとして用いるために、通信プロトコルとして従来からのTCP/IPを採用する場合が多い。OSの介在によるデータコピーの他に、TCP/IPを実現するためのデータコピーも必要となり、そのオーバヘッドは無視できないものとなっている。

これらの要因により、現状の分散システムでは通信のコストが非常に高くなっており、ノード間の協調動作を大きく阻害している。

計算パワーの位置透過性が実現されていない　これは分散システムの管理に利用されるネットワークOSの問題であるが、プロセッサ資源やメモリ資源などの計算パワーに関して、位置透過性が実現されていない。ユーザは常にそれら資源の物理的・論理的な位置を意識する必要があり、その結果、計算パワーの有効利用が不可能となっている。計算パワーの位置透過性を実現し、さらにその上に負荷分散機能を提供して初めて、計算パワーの有効利用が達成される。

現状の分散システムの持つこれらの問題点をハードウェアおよびソフトウェアの両面から解決し、将来のコンピュータ利用において十分な性能を持つマルチユーザ並列計算環境として、本論文はコンピュータ・コロニーを提唱する。

3　コンピュータ・コロニー

本章では、並列アプリケーションを指向した分散システム「コンピュータ・コロニー」について述べる。

3.1　コンピュータ・コロニー

コンピュータ・コロニーとは並列アプリケーションを指向した共有メモリベースの分散システムであり、OSを含めた総合的な環境を指す。

コロニー(colony)とは「群体」であり、多数の個体が集まった上で相互に協調し、全体であたかもひとつの個体であるかのようにふるまう生物をいう。コンピュータ・コロニーでは、システムを構成するノードが緊密に協調動作し、全体を構成する。ノードは物理的に分散していても構わないが、機能的には完全にネットワーク透過であり、ユーザの視点からはコンピュータ・コロニーはひとつの巨大な計算機資源としてのみ捉えられる。コンピュータ・コロニーはマルチユーザで利用され、各ユーザはその上で並列アプリケーションを効率よく実行することができる。

コンピュータ・コロニーでは、従来の分散システムにおける通信レイテンシの問題を、共有メモリベースの通信によって解決する。共有メモリをベースとした通信では、MMU (Memory Management Unit, メモリ保護ユニット) によるメモリ保護を利用することで、通信機構をユーザに対して安全に解放できる。これにより、OS の介在しない高速・低遅延な通信を実現することができるのである。

3.2 コンピュータ・コロニーの位置付け

コンピュータ・コロニーは、分散システムを発展させたマルチユーザ環境である。その上で、今日の分散システムの欠点である通信コストの大きさを克服し、並列アプリケーションの実行能力を実用レベルにまで高めることを目指すものである。分散システムの持つ導入の手軽さや優れた拡張性といった利点を生かしつつ、従来の分散システムでは困難とされてきた並列処理における高性能を目指す。

コンピュータ・コロニーはさらに、OS の働きによって計算機資源の位置透過性と高性能な負荷分散の実現を目指す。これが達成されると、ユーザがコンピュータ・コロニーに投入したプログラムは自動的に最適なノードに配置・実行され、ユーザは処理ノードを一切意識する必要がなくなる。プログラムが並列化されている場合も同様に、並列実行可能な処理単位がコンピュータ・コロニー内部の最適ノードに配置・実行される。マルチメディア処理などのリアルタイム性の要求されるプログラムについてもまったく同様である。なお、ここでいう最適な配置とは、各ユーザの各プログラムが各種資源に対して要求する QoS (Quality of Service) を出来る限り満足させた上で、システム全体として出来る限り高い性能を得られるような配置であり、動的に変化するものである。

ただし、コンピュータ・コロニーが並列アプリケーションの効率的な実行を可能にするといっても、それでもなお、並列処理能力は並列計算機の方が上である。しかし、科学技術計算などの大規模並列処理自体が目的ではない場合、すなわち並列化された日常的なアプリケーションソフト (ワープロソフトや表計算ソフトなど) や並列化されたエンジニアリング用途のソフト (CAD やコンパイラなど) の実行が主目的である場合には、手軽に利用できる並列処理実行環境として、コンピュータ・コロニーは魅力ある環境であると考える。また大規模ジョブの場合であっても、結果をそれほど急がないのであれば、簡易並列計算機という感覚でコンピュータ・コロニーを利用することができる。

3.3 コンピュータ・コロニーの要件

コンピュータ・コロニーの実現には、いくつかの要件がある。

高速・低遅延な通信機構をユーザに解放　並列アプリケーションの実行やその他のノード間協調処理を効率よくおこなうためには、高速・低遅延な通信をユーザが利用できなければならない。これは、コンピュータ・コロニーとして様々な機能を提供するための、いちばんの基盤となるものである。

ただしここで、コンピュータ・コロニーはマルチユーザで利用されるため、通信資源の保護の問題を避けて通ることはできない。今日の分散システムでは、ネットワークインタフェースが各マシンの I/O バスに接続されている。通信資源の保護は OS の介在によって実現されるため、その結果、OS 介在のオーバヘッドが通信レイテンシを増大させるとして問題となっている。

コンピュータ・コロニーではこの問題を、ハードウェアに支援された共有メモリベースの通信によって解決する。共有メモリをベースとした通信では、MMU によるメモリ保護機構を利用することで、通信機構をユーザに対して安全に解放することができる。通信資源の保護は MMU によって達成されるため、通信機構が一度ユーザに解放されると、個々の通信に対しては OS の介在を必要とせず、高速・低遅延な通信が達成される。

プログラムのネットワーク透過な実行　ひとつの並列アプリケーションを高速に実行するにしても、あるいは複数のプログラムを効率よく実行するにしても、システム全体の資源の利用率を上げることが重要である。そのためには、プログラムのネットワーク透過な実行を支援しなければならない。ここで、位置透過なアクティビティの実行機構や、アクティビティの動的な移送機構が必要となる。アクティビティから利用される資源は、アクティビティの実行位置によらずアクセスできなければならない。また、並列アプリケーションにおける複数アクティビティ間の通信も、動的な実行位置の変化に柔軟に対処し得るものでなければならない。

並列アプリケーションのためのプログラミングモデル　並列アプリケーションの作成・実行に適したシステムイメージやプログラミングモデルの提供も、プログラム作成を支援するものとして欠かせない。柔軟で容易な並列プログラミングを可能とするため、自然な並列プログラムの記述を支援するための通信機構や、処理の大小を気にすることなく自由な並列化をおこなうための高速なアクティビティ生成機構が重要である。

高性能な動的負荷分散のためのメカニズムとポリシー
資源の利用率を高め、それによって個々のアプリケーションの効率よい実行やシステム全体としての処理能力の向上を達成するためには、高性能な動的負荷分散が必要である。負荷分散をおこなうためのメカニズムを提供し、さらにコンピュータ・コロニーでは運用のための基本ポリシーを提供しなければならない。

これらの要件のうちのいくつかは、共有メモリをベースにすることで自然に達成することができる。ユーザレベルでの高速・低遅延な通信機構は、ハードウェアに支援された共有メモリベースの通信がその解となる。また、ネットワーク透過な資源へのアクセスや、ネットワーク透過なアクティビティ間の通信は、共有メモリをベースとした資源やプログラミングモデルによって達成される。

本論文では、コンピュータ・コロニーのハードウェア環境の実現例として、共有メモリベースのワークステーションクラスタを示す (第 4 章)。また、コンピュータ・コロニーを実現するためのオペレーティングシステム Colonia を提案する (第 5 章)。

4 コンピュータ・コロニーのハードウェア環境

コンピュータ・コロニーは並列アプリケーション実行のための分散システムの拡張環境であり、共有メモリベースの通信を利用する。本章ではコンピュータ・コロニーのハードウェア環境の実現例として、共有メモリベースのワークステーションクラスタを示す。さらに、共有メモリベースのワークステーションクラスタの拡張についても述べる。

4.1 共有メモリベースのワークステーションクラスタ

コンピュータ・コロニーのハードウェア環境の実現のひとつとして、従来のマシン (ワークステーションおよびパーソナルコン

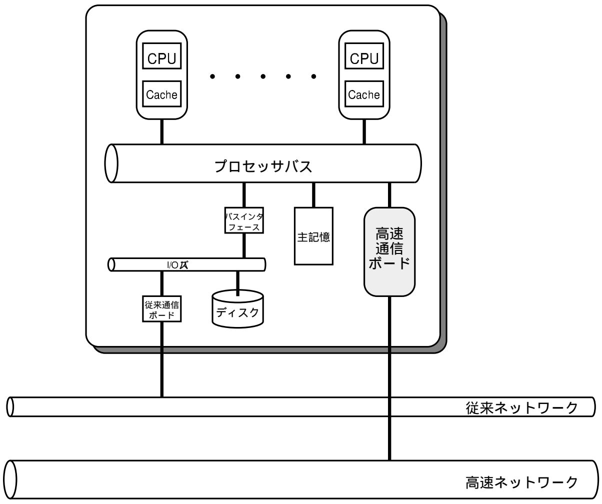

図 1: 要素マシンの構成

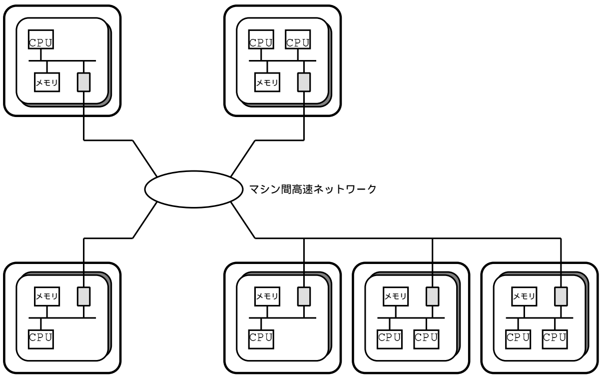

図 2: 共有メモリベースのワークステーションクラスタ

ピュータ) に高速・低遅延な通信を提供する専用のハードウェアを付加し、それによって共有メモリベースのワークステーションクラスタ (および PC クラスタ) を構築することを提案する。

構成　具体的には、クラスタを構成する各要素マシン (ノード) のプロセッサバスに通信ボードを接続する。要素マシンの大まかな構成を図 1 に示す。なお、各要素マシンはシングルプロセッサであっても SMP であっても構わない。また、すべての要素マシンがディスクや従来ネットワークへのインタフェースを有する必要はない。要素マシンを接続して構成された共有メモリベースのワークステーションクラスタの構成を図 2 に示す。

この通信ボードの基本的な機能は、ローカルマシンの物理アドレスとリモートマシンの主記憶を対応付けることである。すなわち、メモリアクセス命令発行時にプロセッサバスを流れる物理アドレスを拾い、必要に応じてリモートメモリアクセスという形の通信を実現するものである。通信効率を向上させるため、単なるリモートメモリアクセスだけでなく、データのキャッシングおよび整合性制御をサポートしてもよい。

ここで、ユーザプログラムにおけるメモリアクセスは、仮想アドレスに対しておこなわれるという点に注意する。そのため、プログラムから本通信ボードを利用してリモートマシンとの通信をおこなうには、仮想アドレスから物理アドレスへの対応付けと、物理アドレスからリモートマシンの主記憶への対応付けを、あらかじめ設定しておく必要がある。

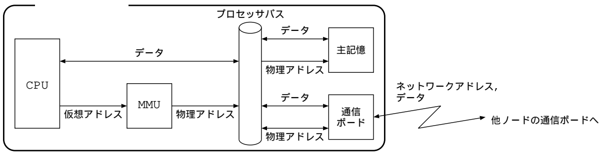

図 3: データおよびアドレスの流れ

しかし一度そのコネクションを確立しさえすれば、あとは通常のメモリアクセスによって他ノードと通信をおこなうことができる。具体的には、次のような手順となる。

1. ユーザは仮想アドレス空間にアクセスする。
2. MMU が仮想アドレスを物理アドレスに変換し、プロセッサバス上に流す。
3. 通信ボードが物理アドレスを拾い、他ノードとの通信の必要性を検知すると、実際の通信処理をおこなう。

データおよびアドレスの流れを、図 3 に示す。図 3 では MMU がプロセッサの外部に描かれているが、プロセッサと同一のチップ上に実装されていても構わない。

個々の通信には MMU による仮想アドレスから物理アドレスへの変換が含まれており、これにより通信ボードは、通常の主記憶と同じ形で保護されている。保護の問題が MMU によって解決されるため、通信機構をマルチユーザに対して安全に解放することができる。その結果ユーザは、OS の介在しない高速・低遅延な通信を利用することが可能となっている。

高速通信用の通信プロトコルも、従来との互換性を意識した TCP/IP ベースのものである必要はなく、またそれほど高機能である必要もない。なぜなら高速ネットワークはローカルな環境内で閉じたものであり、外部との TCP/IP による通信は、I/O バスを経由した従来ネットワーク上で提供すればよいからである。単純な通信プロトコルを採用することで、通信時のオーバヘッドをさらに軽減することが可能となる。

ネットワークアドレス　共有メモリベースのワークステーションクラスタを実現するための通信ボードは、プロセッサバスを流れる物理アドレスを拾い、必要に応じてリモートメモリアクセスをおこなうものである。このリモートメモリアクセスを実現する際、通信ボードは相手ノードの通信ボードに対してネットワークアドレスを使ってアクセス要求を送る (図 3)。そして、それを受けた通信ボードで、ネットワークアドレスから物理アドレスへの変換をおこなう。

ネットワークアドレスとしては様々な形式のものが考えられるが、物理アドレスを直接指示するものでなく、何らかの仮想化をおこなったものを利用することが望ましい。ネットワークアドレスに対して仮想化を施さず、物理アドレスをそのまま使った場合には、受信側でのアドレス変換は不要となる。しかしこの方式では、ページングの際の処理が繁雑になる。なぜなら、ページを参照しているすべてのノードに対して、ページアウトあるいはページインされた旨通知する必要があるからである。ネットワークアドレスという中間形式にいったん変換することで、ノード内における物理アドレスの変更を、他ノードに対して隠蔽することが可能となる。その結果、ノードの独立性が高くなるとともに、柔軟なシステム構築が可能となる。

ネットワークアドレスによるハードウェアの仮想化は、コンピュータ・コロニーにおけるシステムの柔軟な構成と優れた拡張性を保証するための重要なポイントであると思われる。

位置付け　共有メモリベースのワークステーションクラスタが、従来の分散環境に通信ボードを追加するだけで実現できるのは、大きな利点である。これにより、高性能な並列アプリケーションの実行環境であるコンピュータ・コロニーのハードウェア環境を、従来からのハードウェア資産を生かして低価格で実現することが可能となる。

ただし共有メモリベースのワークステーションクラスタは、コンピュータ・コロニー実現のためのハードウェア基盤に過ぎない。システム中の大量の計算機資源を管理し、システムをコンピュータ・コロニーにまで高めるのは、ソフトウェア環境たる分散オペレーティングシステム Colonia の役割である。

関連研究　分散共有メモリ機構をハードウェアでサポートしたワークステーションクラスタとして、キヤノンの Euphoria [1] がある。しかし Euphoria では、すべてのノードが同じアドレスマップを持ち、その中にシステム中のすべてのメモリ資源が配置されている。

4.2　ハードウェア環境の拡張

第 4.1 節では、コンピュータ・コロニーのハードウェア環境の実現例として、共有メモリベースのワークステーションクラスタについて述べた。しかしコンピュータ・コロニーのハードウェア環境としては、共有メモリベースのワークステーションクラスタという分散型の実装の他に、集中型の実装も可能である。

コンピュータ・コロニー並列計算機　並列計算機をクラスタ方式で構築する。処理ノードはシングルプロセッサまたは SMP で、主記憶と通信インタフェースを有する。通信インタフェースは各処理ノードのプロセッサバスに接続されており、通常のメモリアクセス命令によるリモートメモリアクセス機構を提供する。その上で処理ノード毎に OS を稼働させ、ノード間に共有メモリベースの通信を実現する。このような構成の並列計算機を、コンピュータ・コロニー並列計算機と呼ぶ。

コンピュータ・コロニー並列計算機では全処理ノードがひとつの筐体に納まっているため、各処理ノードが別々の筐体に納まっている分散型のコンピュータ・コロニーハードウェア (共有メモリベースのワークステーションクラスタ) と比べてより高速なノード間通信が可能であり、より並列処理に適した環境であると言える。コンピュータ・コロニー並列計算機は、ノードの独立性が高い点、およびノード間が汎用性の高い共有メモリベースのプロトコルで接続されている点から、従来の SMP 機や超並列機と比べて、柔軟でスケーラブルな構成をとることが可能となる。

コンピュータ・コロニー方式の並列計算機を中心に据えた集中システムを構築する場合、ユーザ端末は次のようなものにすることができる。

コンピュータ・コロニー端末装置　端末装置は通常の処理ノードと同様な構造を持ち、通常の処理ノードと同様に、共有メモリベースの通信によってシステムに組み込まれる。ただし、中心となるコンピュータ・コロニー並列計算機とは物理的に離れた場所に配置されるため、並列計算機内部よりは通信が遅くなる。なお、端末装置にはコンソール入出力装置は必須となる。

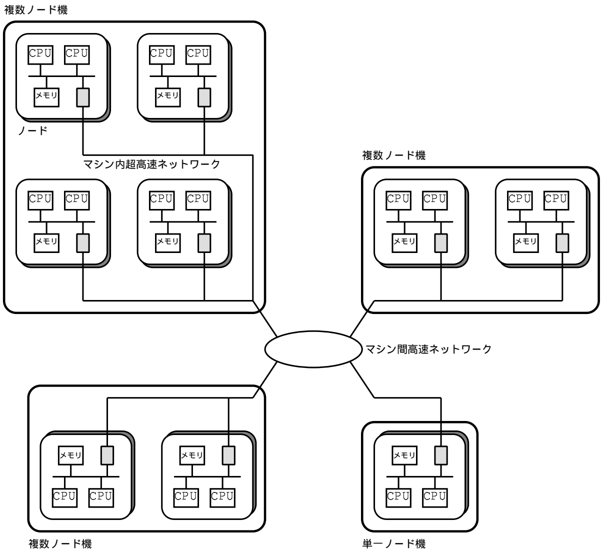

図 4: コンピュータ・コロニー並列計算機クラスタ

このようにして構築された集中システムは、見方を変えれば「処理ノード間の通信速度が非均質なコンピュータ・コロニー」と見なすことができる。この考えをさらに推し進めると、「コンピュータ・コロニー方式の並列計算機を複数台接続したシステム」にたどり付く。

コンピュータ・コロニー並列計算機クラスタ　複数のコンピュータ・コロニー並列計算機を、共有メモリベースの通信で接続する (図 4)。各コンピュータ・コロニー並列計算機のマシン内ネットワークに比べ、マシン間ネットワークは遅いものとなる。

分散型のコンピュータ・コロニー (共有メモリベースのワークステーションクラスタ)、コンピュータ・コロニー並列計算機、コンピュータ・コロニー並列計算機と端末装置による集中システム、コンピュータ・コロニー並列計算機クラスタはいずれも、ノードを基本とした構成やノード間の通信方式など、その基本アーキテクチャは同じである。これらの違いは、処理ノードを納める筐体の違いと、それに伴う処理ノード間の通信距離の違いに過ぎない。すべての環境は統一的に扱うことができ、ユーザはシステムを利用するにあたり、分散/集中などの構成を意識する必要がない。ユーザが唯一注意すべきことは、システムを構築あるいは拡張する際、より高い並列計算能力を重視して並列計算機を導入するか、価格や可用性, 構成の柔軟さを重視してワークステーション群を導入するかということだけである。

5　分散 OS Colonia

コンピュータ・コロニーのハードウェア環境としては、様々な実装が可能である。しかしそれらは共有メモリベースの高速・低遅延な通信を提供する基盤に過ぎない。むしろ重要なのは、ハードウェア環境内の各種計算機資源を管理し、コンピュータ・コロニーとしての種々の機能を実現するソフトウェア環境である。本章では、コンピュータ・コロニーのソフトウェア環境である分散 OS Colonia について述べる。

Colonia は、メモリアクセス命令によって利用される高速・低遅延なリモートメモリアクセス機構を前提とした OS である。

その通信機構を生かして、並列アプリケーションの効率的な実行に重きを置いているという点で、本研究は従来の分散OS研究とは異なるものであり、むしろ並列OSのそれに近いと言える。しかし、マルチユーザによる本格的な並列処理と、マルチメディアにも対応し得る柔軟な利用を目指しているという点で、本研究は新しいものであると言える。

5.1 対象とする環境

Coloniaは、第4.1節で述べた共有メモリベースのワークステーションクラスタ、および第4.2節で述べた拡張環境に対して適用可能なOSである。

Coloniaは、以下のハードウェア環境を想定している。

- 処理ノードの持つ計算機資源は、一般に非均質である。具体的には、
  - プロセッサの計算能力および搭載プロセッサ数
  - 主記憶量
  - ディスクの有無およびディスク量

  などに多様性があるとする。
- ノード間の通信距離も、一般には非均質である。例えば、コンピュータ・コロニー並列計算機を中心に構築された集中システムやコンピュータ・コロニー並列計算機クラスタでは、マシン内とマシン間では通信速度にそれなりの差異があると考えられる。また共有メモリベースのワークステーションクラスタや単体のコンピュータ・コロニー並列計算機であっても、処理ノードを繋ぐネットワークのトポロジによっては、ノード間の通信距離は完全に均質であるとは言えない。
- 一般には、命令セットの異なるプロセッサが混在したコンピュータ・コロニーを考えることもできる。しかしその場合、プログラムのコード領域を共有できないことを主な理由として、システムの運用に様々な制約が生じてくる。そこでColoniaではさしあたって、オブジェクトコードレベルでの互換性を有するプロセッサで構成されたシステムを想定する。

さらに、各ノードの持つ通信ボードに関して、以下を想定している。

- 通信ボードでは、任意の物理アドレスへのアクセスに対して、リモートメモリアクセスをおこなう/おこなわないを設定することができる。この設定は、プロセッサの規定するページ単位でおこなわれるとする。
- 通信ボードはプロセッサバスを流れる物理アドレスへのアクセスをスヌープし、当該物理アドレスに対するメモリアクセスに通信の必要性を検知すると、他ノードの通信ボードと通信し、データの授受をおこなう。
- 上記の通信機構を利用して、処理ノード間の共有メモリ(分散共有メモリ)を実現することができる。ここで、自ノードおよび他ノードからのアクセスによって参照されるページの実体をホームページと呼び、ホームページを持つノードをそのページのホームノードと呼ぶ。
- 分散共有メモリにおいて、アクセス効率を向上させるためにデータのキャッシングをおこなっても構わない。またキャッシングをおこなう際、ノード間のキャッシュ整合性制御はハードウェアでおこなってもソフトウェアでおこなっても構わない。ただし、厳密に速度を要求する環境では、ハードウェアによる支援がある方が望ましい。

実際には、Coloniaの対象は、第4.1節の共有メモリベースのワークステーションクラスタや第4.2節の拡張環境にとどまるものではない。例えば、並列計算機JUMP-1 [2]を考える。JUMP-1はクラスタ構成をとった共有メモリ型並列計算機であり、JUMP-1クラスタを処理ノードと見なすことで、JUMP-1全体をコンピュータ・コロニー並列計算機とまったく同様に扱うことができる。

一般に、通常のメモリアクセス命令によってリモートメモリアクセスができるものであれば、それは共有メモリベースのコンピュータ・コロニーにおける処理ノードと見なすことができる。各処理ノードは、シングルプロセッサであってもSMPであっても、さらには1台の共有メモリ型並列計算機であっても構わない。

5.2 現状のOSの問題点

分散システム向けに利用可能なOSは、大きく次のふたつに分けることができる。

- **ネットワークOS**
  個々のマシン上で動作していた従来のOSに、分散処理機能を追加したもの。ノード毎に独立したOSが存在し、ユーザが分散をある程度意識しなければならない。
- **分散OS**
  統合化された分散処理をおこなうために、最初から分散処理を目的として開発されたOS。ネットワーク透過性の実現に重きを置き、全体をひとつの計算機システムとして利用させることを目的とする。

商用のシステムとして今日利用可能なOSのほとんどは、前者に分類されるものである。既存のOSをベースに、現実的な要請に応じた機能拡張を施しているため、その利用価値は高い。しかし、逆にそのベースとなるOSに引きずられ、分散システムを真に有効利用するものとはなり得ない。

分散OSに関しては、主に研究レベルで様々なシステムが発表されている。しかしまだ、決定打となるシステムは生まれていない。我々はこの原因を、現在の分散OS研究が、今日の分散システムという環境に束縛されているためと考えている。現状の分散システムでは、ノード間の距離、すなわちノード間の通信遅延が非常に大きい。そのような環境でも、ノード間通信の粒度に気を付けさえすればある程度の協調処理は可能であるが、その粒度はあまりに粗く、例えば並列数値計算などは実用になり得ない。そこで我々は、現在の貧弱な分散システム上でOSを研究するのではなく、5年、10年先の高性能な分散システムを見据えた研究をおこなうことに大きな意義があると見ている。第4.1節に述べた共有メモリベースのワークステーションクラスタとは、まさに5年、10年先の分散システムを想定したものである。

本論文の提案する共有メモリベースのワークステーションクラスタは、その高速な通信機構から、今日の分散システムよりはむしろ並列計算機に近いと言える。我々は、将来の分散システムは性能面で並列計算機に近付き、並列計算機もまたその構成法の面で分散システムに近付き得ると考えている。そこでOS研究のプラットホームとしては、共有メモリベースのワークステーションクラスタという完全な分散型のコンピュータ・コロニーにとどまらず、第4.2節に述べた拡張環境も含めた、より一般的な構成のシステムを想定している。

ここで、今日の並列計算機向けOSに目を向けてみる。今日の並列計算機の利用は、ベクトル計算機の延長として、ひとつのジョブをいかに高速に処理するかに重きを置く場合がほとんどである。単一ユーザによる排他的な利用を前提とすれば、資

源の保護について考える必要がなくなる。またこれは、マルチユーザであっても、パーティショニングやギャングスケジューリングによる利用の場合はほぼ同様である。しかし、このようなマルチユーザ環境は柔軟性を欠く上、並列計算機の持つ全資源を効率よく管理しているとは言えない。

最近では、IBM の RS/6000 SP や富士通の AP3000 のように、ワークステーションクラスタとほぼ同様の構成をとった並列サーバと呼ばれる計算機も出ている。これらの運用は、計算機全体をワークステーションの集合体と見なし、各ノード上に UNIX などの従来 OS を載せる (結果として、システム全体がネットワーク OS によって運用される) ことでおこなわれている。しかしこれも、並列計算機の持つパワーを真に引き出すものではない。

以上のように、今日の並列計算機向け OS はマルチユーザ環境として貧弱であり、また資源の効率よい活用を実現するものでもない。

これらの現状を踏まえた上で、コンピュータ・コロニーのソフトウェア環境としての分散 OS Colonia を提案する。

5.3 設計目標

Colonia の果たすべき役割のうち最も基本となるのは、共有メモリベースの並列計算環境を実現するための次の 2 点である。

- 物理的にメモリを共有していないノード間で、共有メモリを実現する。
- その上で、任意のアクティビティ(Colonia ではユニットと呼ぶ) 間でのメモリ空間の共有を可能にする。

これらを実現した上で、システムをコンピュータ・コロニーたらしめる次の 2 点の達成が、Colonia の最終的な目標である。

- 並列アプリケーションの効率的な実行
  個々の並列アプリケーションを、できる限り高速に実行する。これは、個々のプログラムに対して最大の性能を提供しようというものである。
- **QoS** を満たすためのシステム全体に渡るスケジューリング
  個々のプログラムが各種資源に対して要求する QoS を満足するよう、システム全体に渡ったスケジューリングをおこなう。これは、個々のプログラムが要求する最低ラインの性能を確保しようというものである。

ただし、これらはトレードオフの関係にあるものである。どちらをより優先するかは、システム運用上のポリシーによって定まるものである。最も穏健なポリシーは、個々のプログラムが要求する QoS を満足させた上で、余った資源を個々のプログラムのさらなる高速化に利用しようというものである。しかし場合によっては、他の何にも増して最大性能が必要となるプログラムがあるかもしれない。コンピュータ・コロニーはマルチユーザ環境であることも考慮に入れた上で、システム運用のポリシーが決定されることになる。

上記のふたつの目標を達成するために、以下の技術項目を実現しなければならない。

- 位置透過性の実現 ··· ユーザに計算機資源の物理的な位置を意識させない。さらには、プログラムの実行位置さえも意識させない。
- 並列アプリケーションの支援 ··· 並列アプリケーションの効率的な実行を支援するための、プログラミングモデルと実行メカニズムを提供する。
- マルチユーザ環境の実現 ··· システム内のすべての計算機資源を管理し、マルチユーザに提供する。
- **QoS** を考えた資源割り当て ··· システム内の計算機資源をマルチユーザに割り当てるにあたり、各ユーザが各種資源に対して要求する QoS (Quality of Service) を出来る限り満足させる。ただしマルチユーザ環境である以上、不当であると判断された要求は受理されない。
- マルチメディア処理への対応 ··· マルチメディア処理への対応は、今後のコンピュータ環境の重要な課題である。マルチメディア処理は、QoS を要求するプログラムの典型である。
- システム全体としての性能の向上 ··· マルチユーザの利用において、システム全体の資源の利用率を上げ、それによってシステム全体として出来る限り高い性能を得る。
- 動的な負荷分散 ··· システム内の計算機資源の有効利用をおこなうため、アクティビティの移送も含めた様々な負荷分散の機構が必要である。
- アクティビティの高速な移送 ··· 動的な負荷分散を支援するため、アクティビティの移送コストは出来る限り低い方がよい。
- アクティビティの高速な生成 ··· あるアプリケーションがどれだけ並列化されており、そしてどれだけアクティビティを生成するかは、問題の性質とプログラマ次第である。並列アプリケーションの実行を支援する上で、アクティビティの生成コストは出来る限り低い方がよい。

これらを実現するために我々が採用したアプローチが、「マイクロカーネル方式の OS 構成」と「ミッション-ユニットモデルと呼ぶプログラミングモデル」である。

5.4 システム構成

Colonia は、いわゆるマイクロカーネル方式の構成をとる。すなわち、ノード毎に稼働するマイクロカーネルがごく基本的な機能のみを提供し、それ以外の OS 機能は複数のユーザレベルのサーバによって実現される。

マイクロカーネル方式の構成を採用したのは、マイクロカーネル方式の持つ次のような利点による。

- カーネルおよびサーバが高度にモジュール化されており、設計、作成および保守などが容易である ··· カーネルそのものが小さく、また機能的にモジュール化されているため、設計や作成、保守の面でモノリシックカーネルより有利である。またマイクロカーネルの上に構築される各種サーバも、機能的にモジュール化されており、カーネルとの切り分けがはっきりしているため、新しいサービスの実装が容易である。
- システムを停止することなく、サービスを追加・削除・変更することができる ··· サービスを追加、削除あるいは変更する際、モノリシックカーネルを使った場合にはシステムを一時停止したり新しいカーネルのブートをしたりする必要があるのに対し、マイクロカーネル方式ではその必要がない。これにより、新しいサービスの実現、インストール、デバッグが容易になる。
- **OS** のシステム構成を、ノード毎に調整することができる ··· OS の提供すべきサービスは、ノードの持つ計算機資源によって異なる。ディスクの有無、従来ネットワークへ

のインタフェースの有無など、コンピュータ・コロニーにおけるノードの構成は多様であるが、必要に応じて必要な機能を持つサーバを追加できるマイクロカーネル方式ならば、この多様性に柔軟に対処することができる。

- カーネル部分の実行時間を束縛 (**bound**) できる ··· あるイベントに対し、そのイベントに興味を示しているユーザレベルのプロセスがスケジュールされるまでにカーネル内部で費やされる時間を、カーネルのオーバヘッドという。リアルタイムシステムでは、この時間を束縛することが本質的である。ここで要求されているのは、イベントに対して可能な限り高速に応答することではなく、応答に要する時間の最悪値が予測できるということであり、マイクロカーネルではこれが可能である。
- メカニズムとポリシーを分離できる ··· カーネルは効率的な基本メカニズムを提供し、ポリシーはユーザレベルで柔軟に表現することができる。また、プログラマはユーザレベルのソフトウェア間の相対的な優先度を制御することが可能である。

この中で特に重要なのは、リアルタイム処理に対応するための「カーネル部分の実行時間を束縛 (bound) できる」という性質と、資源割り当てや負荷分散のための種々のポリシーを実装するための「メカニズムとポリシーを分離できる」という性質である。

マイクロカーネルに基づいたシステムの欠点としては、あるサービスがカーネル内部ではなくユーザレベルにおいて実装された時、応答が遅くなるという点を挙げることができる。しかし、モノリシックカーネルを用いたもの (Sprite) とマイクロカーネルを用いたもの (Amoeba) を詳細に比較してみた結果、これがそれほど大きな問題とはならないという報告もあり [3]、Colonia におけるマイクロカーネル方式の導入は問題ないと判断した。

なお Colonia では、物理的な資源の位置を次の単位で管理する。

- プロセッサ ··· アクティビティのディスパッチはプロセッサに対してなされるが、それ以外に個々のプロセッサが意識されることはない。
- ノード ··· ノードとは、物理的にメモリを共有する、密結合された物理的資源の集合体である。具体的には、コンピュータ・コロニーにおける処理ノードである。各ノードはプロセッサ (ひとつまたは複数), 主記憶, 通信インタフェースからなり、必要に応じて I/O 装置を持つ。
- テリトリ ··· テリトリとは、Colonia によって管理されるノードの全集合である。

5.4.1　マイクロカーネル

Colonia カーネルは特権モードで実行され、プログラムの実行とアクティビティ間通信に必要となるごく基本的な機能のみを提供する。具体的には、次のような機能である。

- ハードウェアインタフェースの管理
- アクティビティの管理・実行
- アクティビティ間の通信機構
- 最小のメモリ管理

Colonia カーネルで重要なのはシステムを動作させるのに必要なメカニズムの提供であり、そのポリシーはユーザレベルのサーバに委ねられている。

Colonia カーネルは以下の基本概念を提供する。

- ミッション
  実行中のプログラム。あるいは、ユーザがコンピュータシステムの中に立てる代理である。ミッションはひとつまたは複数のユニットから成る。
- ユニット
  アドレス空間の単位であると同時に、逐次実行の単位である。ひとつのユニットは、必ずひとつのミッションに属する。ユニットはノードに対して割り付けられ、その中で実行される。
- 共有メモリ
  同一ミッションあるいは異ミッションを問わず、任意のユニット間での共有メモリを提供する。
- シグナル
  任意のユニットに対して非同期事象を伝達する。

5.4.2　システムサーバ

カーネル上に構築され、種々のサービスを提供するのがサーバの働きである。サーバは通常、サービス毎に分割して構築される。一般にサーバには、ローカルサーバとグローバルサーバがある。

- ローカルサーバ ··· ノードに固有なサービスを提供するサーバ。ノードを管理する上でそのノードで動作することが本質であるものや、実行効率などからそのノードで動作することが厳密に望まれるもの。
- グローバルサーバ ··· テリトリ全域に渡るサービスを提供し、その実行位置を特定のノードに限定する必要のないサーバ。

サーバのうち、Colonia がシステムの一部として標準的に提供するものをシステムサーバと呼ぶ。システムサーバとして、以下のものを考えている。

- グローバルスケジューラ ··· システム全体の計算機資源の利用状況を監視し、ユニットの移送や共有メモリの管理などをおこなう。ページャや CPU スケジューラと密接な関係にある。
- ページャ ··· ページフォールトを解決するために、物理ページの退避・復帰をおこなう。ページの退避先としては、自ノードのディスクの他に、他ノードの主記憶やディスクという選択肢もある。ノードの中にはディスクを持たないもの (ディスクレスノード) もあるため、他ノードへのページアウト (ネットワークページアウト) は必須機能である。
- プロセッサスケジューラ ··· ノード内のプロセッサ群に対して、CPU スケジューリングをおこなう。
- ネームサーバ ··· システムに名前空間を形成する。
- 認証サーバ ··· 資源に対するアクセスの認証をおこなう。
- ファイルサーバ ··· ファイルとは、永続的なアドレス空間である。ファイルサーバは、ファイルの実体の位置を管理する。
- デバイスサーバ (デバイスドライバ) ··· I/O デバイスの制御をおこなう。

5.5 プログラミングモデル

UNIX では、プログラムとプロセッサを一体化してプロセスとしていた。これは、ひとつのメモリ空間とひとつのプロセッサしかない基本的な計算機をモデル化したものであり、複数のプロセッサが同一のプログラムを実行する共有メモリ型マルチプロセッサを上手く取り扱うことができなかった。これに対して Mach では、プロセスをタスクとスレッドに分解し、複数のスレッドがひとつのタスク内で走ることを許すことで、マルチプロセッサを取り扱うことに成功した。

Colonia ではさらに、共有メモリを利用可能な分散環境の中で並列プログラムの効率よい実行を実現するモデルとして、ミッション-ユニットモデルを採用している。

ミッション-ユニットモデルでは、プログラムをひとつのミッションとし、その中の並列性をユニットによって記述することを基本とする。コードなどの共有可能なデータをミッション内で共有し、コンテクストはユニット毎に独立して持つことで、ユニットの高速な生成および移送を実現するものである。

5.5.1 ミッション-ユニットモデル

Colonia におけるプログラムの実行は、ミッション-ユニットモデルに基づいておこなわれる。これは次のようなものである。

- ユーザはシステムにミッションを投入する。ミッションとはユーザがコンピュータシステム内に立てる代理であり、システムに投入された一連の仕事をおこなうプログラムである。ひとりのユーザは、同時に複数のミッションを投入することが可能である。
- ミッションは、協調動作する複数のユニットよりなる。ユニットとはミッションの完了に不可欠な存在であり、それぞれが固有のアドレス空間を持ったアクティビティである。各ユニットはノードに対して割り付けられ、ノード内のプロセッサで実行される。
- ひとつのミッションに属するすべてのユニットは、ミッション生成時に指定したアドレス空間の固定領域を、アドレス位置も含めて完全に共有する。また必要に応じて、ミッションの同異に依らない任意のユニット間に共有メモリを張ることも可能である。
- アクティビティたるユニットは、システムコールを発行することによりミッションおよびユニットを生成することができる。ミッションが生成されると、そのミッション内にルートユニットと呼ぶユニットが自動的に生成される。ユニットがミッションやユニットを生成していくことにより、ミッションおよびユニットの木が構成される。この様子を図 5 に示す。図中のログインミッションとは、ユーザがコンピュータ・コロニーを利用する際に、システムからまず最初に割り当てられるミッションである。
- 必要に応じて、ユニット内で軽量プロセスを利用することもできる。ただし、軽量プロセスはユーザの責任で利用されるものであり、カーネルからその存在が意識されることはない。

以下で、ミッション-ユニットモデルについて詳述する。

5.5.2 アドレス空間

ミッション-ユニットモデルでは、アクティビティであるユニットがそれぞれ固有のアドレス空間を有している。ただしそのアドレス空間の一部は、同一ミッションに属するすべてのユニットとメモリを共有するのに利用される。この領域をミッション内共有領域と呼び、それ以外の領域をユニット固有領域と呼ぶ。ユニットのアドレス空間を図 6 に示す。

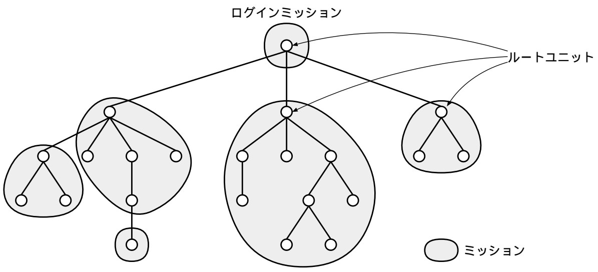

図 5: ミッション-ユニット木

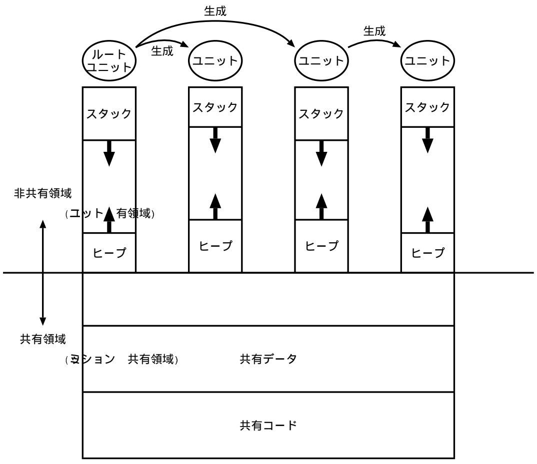

図 6: ユニットのアドレス空間

ミッション内共有領域 ミッション内共有領域は、ミッションの生成時に指定される。ミッション内に生成されたすべてのユニットは、そのミッション内共有領域を自身のアドレス空間の同じアドレス位置にマッピングする。ミッション内のユニットはその領域を完全に共有し、このメモリ共有領域ではポインタをそのまま使うことができる。この共有領域は、コードやデータの共有に利用される。

ユニット固有領域 ユニット固有領域は、ユニットが完全に自由に利用できる領域であり、スタックやユニットに固有なデータ空間として利用される。ユニット固有領域は、他ユニットから完全に保護される。そのため、他ユニットによるスタックの破壊の心配がないといった利点がある。

この領域にプログラムをロードし、ミッション内の他ユニットとは別のプログラムを実行することもできる。これについては、第 5.5.6 項に詳述する。

ここで、Mach のタスク-スレッドモデルを考えてみる。Mach におけるタスクとはアドレス空間の割り当て単位であり、その中の並列実行可能な処理単位がスレッドである。Mach においては、アクティビティたるスレッドが、タスクの持つアドレス

空間を完全に共有している。そのため、別スレッドによってスタックが破壊されるなどの危険があった。

またスレッド間でアドレス空間を完全に共有するため、実際には共有する必要のないデータ (例えばスタック) まで共有されていた。この点は SMP 機などの物理的にメモリを共有する計算機環境では問題にならないのだが、コンピュータ・コロニーのようにノード間では物理的にメモリを共有しない環境では、性能低下の原因となり得るものである。これは次のような理由による。

ひとつのタスクに属するスレッドを複数ノードで実行する場合、この不要な領域に対してもノード間での共有が設定されることになる。ノード内での共有メモリとは異なり、ノード間の共有メモリのコストはゼロではない。例えば更新型のプロトコルによってノード間共有メモリの整合性制御をおこなう場合には、あるスレッドでスタックの操作をおこなうたびに、他ノードのデータを更新しなければならないのである。

この問題があるため、Mach のタスク-スレッドモデルは、物理的にメモリを共有しない環境でアクティビティを分散させるのに適したプログラミングモデルとは言えない。

それに対しミッション-ユニットモデルでは、スタックなどアクティビティ間で共有される必要のないデータ (ユニットに固有なデータ) はユニット固有領域に配置され、基本的に他ユニットと共有されることがない[1]。そのため、不要なデータ共有による性能低下を避けることができる。

5.5.3 ユニット間の共有メモリ

Colonia ではユニット間の共有メモリとして、次の 2 種類を提供する。

- ミッション内共有領域による共有メモリ
- エクスポート/マウント方式による任意のユニット間の共有メモリ

以下、それぞれについて述べる。

ミッション内共有領域による共有メモリ　Colonia の提供する共有メモリで最も基本となるのが、ミッション内共有領域によるミッション内ユニット間の共有メモリである。この共有メモリは、コードやデータの共有に利用される (図 6)。ミッション内共有領域はすべてのユニットの同じアドレス位置にマッピングされているため、この方式による共有メモリでは、ユニット間でポインタをそのまま渡すことができるという特徴を持つ。

共有領域中のコード領域はミッション生成時に初期化され、ミッション内のユニットは通常このコードを利用する。コードを共有することで、Colonia ではユニットの高速な生成や移送が可能となっている。ユニット生成の詳細は第 5.5.4 項に、ユニット移送の詳細は第 5.5.5 項に述べる。

ミッション内共有領域中のデータは、ミッション内のすべてのユニットから自由にアクセスされる。ここで、共有領域にアクセスするすべてのユニットはアドレス空間に関する完全な合意があるはずであり、その利用は完全にユーザに任される。

なおミッション内共有領域による共有メモリの解放は、その領域がエクスポート/マウント方式の共有メモリ (後述) によって他ミッションに属するユニットから利用されていない限り、ミッションの終了時におこなわれる。

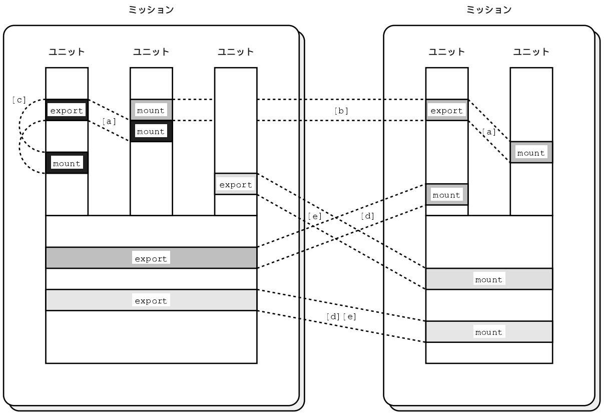

図 7: エクスポート/マウント方式の共有メモリ

エクスポート/マウント方式の共有メモリ　Colonia では上述のミッション内共有領域によるミッション内ユニット間の共有メモリに加えて、エクスポート/マウント方式 (export/mount 方式) による任意のユニット間の共有メモリを提供する。このエクスポート/マウント方式の共有メモリでは、ミッションの同異に依らない任意のユニット間で共有メモリを張ることができる。

エクスポート/マウント方式による共有メモリの設定の手順は、次の通りである。

1. あるミッションのあるユニットが、自身のアドレス空間の一部をエクスポートするシステムコールを発行し、他ユニットからアクセス可能な領域として登録する。この際、どのユニットにどのようなアクセス (読み書き可能, 読み出しのみ可能など) を許可するか、併せて登録しておく。エクスポートをおこなったユニットをエクスポータ・ユニット (exporter unit) と呼ぶ。
2. 他ユニットがマウントシステムコールを発行し、エクスポータ・ユニットによってエクスポートされている領域を自身のアドレス空間の任意のアドレス位置にマッピングする。ただしこの時、マウントシステムコールを発行したユニットがエクスポートされている領域をマウントする権利を有するか、認証がおこなわれる。マウントをおこなったユニットをマウンタ・ユニット (mounter unit) と呼ぶ。
3. 一度エクスポート/マウントがおこなわれると、その共有メモリ領域のアクセスにおいて、エクスポータ・ユニットとマウンタ・ユニットにまったく違いはなくなる。

エクスポータ・ユニットとマウンタ・ユニットとの関係に制約はない。エクスポータ・ユニットとマウンタ・ユニットは、同じミッションに属していても (図 7[a])、違うミッションに属していても (図 7[b]) 構わない。さらには、エクスポータ・ユニットとマウンタ・ユニットが同一であっても (図 7[c]) 構わない。

また、エクスポートする領域とそれをマウントする領域についても特に規定はない。よって、ミッション内共有領域をエクスポートしても (図 7[d])、ミッション内共有領域にマウントしても (図 7[e]) 構わない。

なお、エクスポート/マウント方式で共有された領域は、すべてのユニットで同じアドレス位置にマッピングされているとは限らない。そのため、ミッション内共有領域による共有メモリと違い、ポインタは一般に利用することができない。

---

[1] 第 5.5.3 項に述べるエクスポート/マウント方式のユニット間共有メモリを使えば、この領域を共有することも可能である。

エクスポート/マウント方式による共有メモリの解放は、その共有メモリを利用する最後のユニットの終了時におこなわれる。

Colonia は、上記の 2 種類のユニット間共有メモリを提供する。しかし一口にユニット間の共有メモリといっても、コンピュータ・コロニーはノード間では物理的にメモリを共有しないため、その実現においてはノードの同異が問題となる。

共有メモリを張ったユニットが同一ノード内に存在する場合は、各ユニットの仮想アドレスを同一の物理アドレスにマッピングすることで共有メモリが実現される。これは、ページテーブルの操作によって達成される。

これに対し、共有メモリを張ったユニットが異なるノードに存在する場合は、ネットワークを介したノード間の通信が必要となる。これは、ページテーブルの操作の他、通信ボードの利用する送信テーブル (物理アドレスからネットワークアドレスへの変換テーブル) および受信テーブル (ネットワークアドレスから物理アドレスへの変換テーブル) の設定が必要となる。なお、ネットワークアドレスとは通信ボード間で利用されるアドレスで、主記憶の物理的な位置を指示するものでなく、仮想化されたアドレスである。ネットワークアドレスという中間形式にいったん変換することで、ノード内における物理アドレスの変更を他ノードに対して隠蔽することが可能となる。例えば、ホームページ (第 5.1 項) がページアウトされても、そのホームページを参照していたノードにその旨通知する必要がなくなる。

5.5.4 ユニットの高速な生成

ミッション-ユニットモデルにおけるアプリケーションの並列化は、並列動作可能な処理の流れをユニットに割り当てることでおこなわれる。プログラマがプログラムから抽出した並列性を生かすには、高速なユニットの生成がひとつのポイントとなる。もしユニットの生成が遅ければ、プログラマはユニット生成にかかるオーバヘッドを見積もって並列プログラムを記述しなければならない。純粋に並列性を引き出したプログラムを記述するには、ユニットの高速な生成機構が重要となる。

ミッション内に新しいユニットを生成する手順は、次の通りである。

1. あるユニットが、新規ユニットを生成するシステムコールを発行する。
2. ユニット生成のシステムコールを受けると、Colonia はまずユニットを生成・実行するノードを決定する。この決定は、システムサーバがおこなう。システムサーバは何らかのポリシーに従ってユニットの生成ノードを決定し、そのノードに実際にユニットを生成するよう、そのノードのカーネルに対して低レベルのシステムコールを発行する。
3. ユニットを生成するノードではまず、そのノードから当該ミッションのミッション内共有領域を利用できるよう、ノード間の共有メモリを設定する。
4. ミッション内共有領域を共有するアクティビティを生成し、実行可能キューに登録する。この際、コードはミッション内共有領域にあるものが利用される。

新規ユニットはミッション内共有領域にあるコードを利用するため、各ノードでプログラムをロードする手間を省くことができる。

また、あるノードで既にそのミッションに属するユニットが動いているなど、ミッション内共有領域が利用可能な状態にあれば、そのノードにおけるユニット生成はさらに高速におこなうことができる。これを生かすため、あらかじめミッション内共有領域を利用可能な状態にしておくシステムコールを提供する。Colonia で適宜このシステムコールを発行しておくことで、新たなノードでのミッション生成を高速化することができる。

5.5.5 ユニットの高速な移送

動的な負荷分散をおこなうには、ユニットの高速な移送機構が望まれる。

ユニットの移送では、ユニットの新規生成時に必要なミッション内共有領域の準備に加え、移送元ノードから移送先ノードにユニット固有領域を移動する必要がある。

ユニットの移送時は、まず最初にミッション内共有領域の準備をおこなう。ミッション生成時と同様、移送先ノードでミッション内共有領域が既に利用可能な状態にあれば、この手順を省くことができる。

ユニット固有領域の転送方式としては、次のふたつがある。

- 移送先へすぐに移動する方式
- 移送元と移送先で共有を設定し、アクセス発生時に適宜コピーや移動をおこなう方式

前者はプリページング方式、後者はデマンドページング方式として知られている方法である [4]。プリページング方式では、無駄な転送をおこなう危険性があるものの、ワーキングセットを考慮して一括した転送をおこなうことにより、性能向上が期待される。デマンドページング方式では、移送の開始から実行の再開までの応答時間を短縮することができ、また無駄なページを転送せずに済ますことができるが、ユニット移送後も移送元ノードからページをフェッチし続けることになり、負荷の移動が緩慢になる。

Colonia でどちらを採用すべきかは、今後の研究課題のひとつである。

5.5.6 ミッション内での別プログラムの実行

ミッション内のユニットは、ミッション内共有領域にロードされたコードを実行するのが基本である。しかし、別プログラムを実行することも可能である。同一ミッション内で他のプログラムを実行したい場合には、ユニット固有領域に適宜プログラムをロードし、実行する (図 8)。

さらに、別プログラムをロードしたユニットが自身と同じプログラムを実行する子ユニットを生成するために、fork システムコールを用意する。これは新しいユニットを生成した上で、ユニット固有空間内のコードを共有、データおよびヒープをコピーするものである (図 8)。この時のコピーには、書き込み時コピー (copy-on-write) の手法を使うことができる。

ただし、別プログラムのロードや fork は、ユニットの高速な生成を阻害するものであり、ユーザはあまり利用すべきでない。

5.5.7 ユニットの配置に関するプリファレンス

ユニットの配置は、プロセッサ資源, メモリ資源, 通信量などを基に、Colonia によって最適化される。例えば、頻繁に通信がおこなわれるユニット同士は、できる限り同じノードに配置される。ユーザがシステムにミッションを投入すると、ルートユニットを基点として次々にユニットが生成される。ユニットは Colonia が選択した最適なノードで実行され、また最適なノードへと動的に移送される。

OS によるユニット配置の最適化の前提があるため、下手にユーザが特定の位置にユニットを縛るべきではない。ユーザが

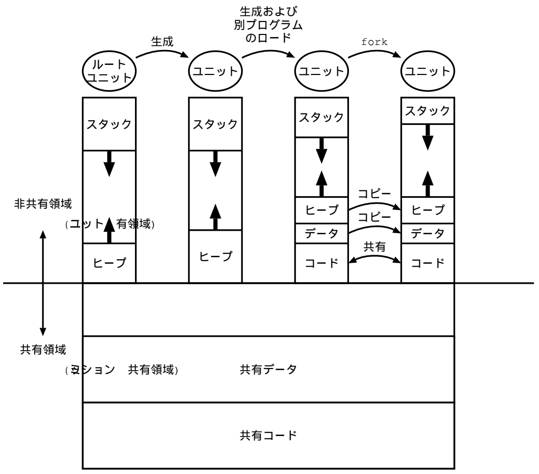

図 8: 別プログラムのロードとユニットの fork

ユニットの位置を厳密に定めてしまうと、負荷の不均衡、テリトリ全体の計算機資源の利用率の低下、そしてプログラムの実行性能の低下などを引き起こす原因となる。

そこで、ユニットの位置やミッション内ユニット同士の距離に関して、プリファレンスという形でユーザが設定できるようにする。これはユニットの配置に関するユーザからの要望・提案であり、OS がどの程度それに従うべきかという優先度付きで指定される。

これは、例えば次のように利用される。

- 特定のデバイスを利用するため、できる限りそのデバイスを持つ特定のノードにユニットを置きたい。それが不可能な場合でも、できる限りそのノードとの通信コストが低いノードに置きたい。
- あるユニットとあるユニットでは頻繁に通信が発生するため、通信コストの低さに重きを置いた配置をして欲しい。すなわち、もし可能ならば両者を同じノードに置いて欲しい。それが不可能な場合でも、できる限り通信コストが抑えられるような配置にして欲しい。
- 上記とは逆に、あるユニットとあるユニットではほとんど通信が起こらない。両者の通信距離は気にせず、それぞれに対してできるだけ多くのプロセッサ資源 (CPU タイム) を割り当てるように配置して欲しい。

ユニットの配置に関するプリファレンスは、OS による計算機資源割り当てのヒントとして利用される。他ユーザを含めたシステム全体の性能を阻害しない限りにおいては、OS は極力このプリファレンスに沿った資源割り当てをおこなうようにする。これにより、プログラムに関するユーザの知識を、OS が積極的に利用できるようになる。

なお、システムの提供するユニットなどで、位置を厳密に固定する必要があるもの (ローカルサーバ) については、最高優先度のプリファレンスを設定することで対処する。

5.5.8 プログラミング上の注意

Colonia は基本的に、ひとつの並列アプリケーションをミッションに対応させ、その中の並行実行可能な単位をユニットに対応させたプログラミングを想定している。Colonia は、同一ミッションに属する複数のユニットの効率的な並列実行に重きを置いて種々の最適化をおこなうため、その結果、異ミッションに属するユニットでは、効率的な並列実行が保証されない。ユーザはこの点を念頭に置いてプログラミングをおこなうべきである。

5.6 通信

Colonia カーネルはユニット間通信のプリミティブとして、共有メモリとシグナルのふたつを提供する。

通常のデータの授受は、共有メモリを介しておこなう。ハードウェアによる支援により、コンピュータ・コロニーでは高速・低遅延な共有メモリが利用可能である。

しかし、共有メモリでは同期的な通信しかおこなうことができない。共有メモリ上で非同期に発生するイベントを伝達するには、ポーリングを使わざるを得ないという問題がある。そこで非同期イベントの効率のよい伝達手段として、シグナルを導入している。

5.6.1 共有メモリ

共有メモリによる効率のよいユニット間通信は、本システムの大きな特徴である。メモリアクセスによる通信は MMU によって完全に保護され、高速・低遅延の通信をユーザに対して安全に解放することができる。

ユニット間で共有メモリを介した通信をおこなうための手順は、次のようになる。

1. まず最初に、通信をおこなうユニットのアドレス空間 (仮想アドレス空間) の特定領域でメモリを共有するよう、カーネルに要求を出す。ユニット内共有領域による共有メモリの場合は、ユニットの生成時に自動的に共有の設定がなされる。エクスポート/マウント方式による共有メモリの場合は、明示的に共有の設定がおこなわれる。

2. ユニット間の共有メモリの設定要求を受けると、カーネルはメモリ空間の共有を実現するよう、以下のテーブルを適宜設定する。

    - ページテーブル (仮想アドレス → 物理アドレスの変換テーブル)
    - 逆引きページテーブル (物理アドレス → 仮想アドレスの変換テーブル)
    - 送信テーブル (物理アドレス → ネットワークアドレスの変換テーブル)
    - 受信テーブル (ネットワークアドレス → 物理アドレスの変換テーブル)

    送信テーブルと受信テーブルは、ノード間の共有メモリを実現するためのテーブルである。そのため、メモリを共有するユニットがひとつのノードに集まっている場合には、送信テーブルと受信テーブルの設定は不要である。

3. ユニット間に共有メモリによるコネクションが確立した後は、ユーザがそのアドレス領域に対してメモリアクセスをおこなうことで、ユニット間の通信が実現される。

    この段階をより細かく見ると、

    (a) ユーザが仮想アドレス空間内の共有メモリ領域にアクセスする。
    (b) MMU が仮想アドレスを物理アドレスに変換し、プロセッサバス上に流す。
    (c) 通信ボードが物理アドレスを拾い、他ノードとの通信の必要性を検知すると、実際の通信処理をおこなう。

となっている。

個々の通信の中には MMU による仮想アドレスから物理アドレスへの変換が含まれており、これにより通信ボードは、通常の主記憶装置と同じ形で保護されている。保護の問題が MMU によって解決されるため、通信をユーザに対して安全に解放することができ、その結果ユーザは、OS を介在しない高速・低遅延の通信機構を利用することが可能となっている。

4. それ以上の通信が必要なくなった場合は、ユニット間の共有メモリを解除するようカーネルに要求を出す。ただし、ミッション内共有領域の共有メモリは解除されない。

本方式で通信をおこなうには、最初に仮想アドレス空間の間に共有メモリを張る必要があり、このセットアップの段階で多少コストがかかる。しかし一度共有メモリが張られた後は、仮想アドレス空間に対して通常のメモリアクセスをおこなうことで、高速・低遅延のユニット間通信をおこなうことが可能となる。

なお、ユニット間通信がミッションをまたがるか否かは、最初にコネクションを確立する際の手順およびコストには効いてくるが、個々の通信には一切影響を与えない。

5.6.2 シグナル

上記のユニット間共有メモリを利用することで、データの授受を達成することができる。しかし、共有メモリによる通信では、非同期に発生するイベントを効率よく伝達することができないという大きな問題がある。これに対処するため、ユニット間通信のプリミティブとしてシグナルの提供もおこなう。

シグナルはそれほど大きくない固定長のデータを伴い、相手ユニットに対してネットワーク透過に伝達される。各ユニットはあらかじめ、自分に送られるシグナルの種類によってそれを受信するか無視するか登録しておく。シグナルを受信する場合にはさらに、受信後に制御を移すシグナルハンドラを登録することができる。シグナルハンドラ内でおこなう処理は、ユーザの自由であり、例えば別ユニットを生成し、処理の並列性を高めることも可能である。

5.6.3 その他の通信機構

カーネルが提供する通信のプリミティブは、共有メモリとシグナルだけである。その他の通信機構を利用したい場合は、サーバやライブラリの中で共有メモリとシグナルを組み合わせることで実現する。

例えば、あるユニットとの間でメッセージ風の通信をしたい場合には、次のような手順で実現される。

1. 送信側ユニットで、通信バッファ用のメモリ領域を確保しておく。
2. 送信側ユニットが受信側ユニットに対し、コネクションを張るための接続要求シグナルを送る。シグナルには、通信バッファのアドレスが含まれている。
3. 受信側ユニットでは、接続要求シグナルに従って共有メモリを張る。
4. 実際のデータの授受をおこなうための通信要求シグナルを送り、ユニット間で実際のデータの授受をおこなう。
5. 通信終了後、ユニット間のコネクションを切断する。コネクション切断までに、データの授受を何度おこなっても構わない。

また、不特定ユニット間でのメッセージを実現するサーバは、次のように実現される。

- メッセージの送信
    1. 送信側ユニットで送信バッファを確保し、メッセージを書き込んでおく。
    2. データの書き込みが完了したら、メッセージサーバに送信シグナルを送る。シグナルには、送信バッファのアドレスが含まれている。
    3. メッセージサーバは送信バッファとの間に共有メモリを張り、そこからメッセージをコピーしておく。
    4. メッセージのコピーが完了すると、送信バッファを解放していい旨、送信側ユニットにシグナルを送る。
- メッセージの受信
    1. 受信側ユニットで共有メモリ用アドレス空間上に受信バッファを確保する。
    2. メッセージサーバに受信シグナルを送る。シグナルには、受信バッファの共有メモリアドレスが含まれている。
    3. メッセージサーバは受信バッファとの間に共有メモリを張り、そこに対してメッセージをコピーする。
    4. メッセージのコピーが完了すると、受信バッファにメッセージが書き込まれている旨、受信側ユニットにシグナルを送る。

メッセージの送信・受信のどちらの場合でも、メッセージサーバとの間であらかじめ共有メモリを張っておけば、処理を軽減することができる。また、上記では送信側ユニットからメッセージサーバへのデータのコピー、およびメッセージサーバから受信側ユニットへのデータのコピーをおこなっているが、送信バッファと受信バッファにメモリの共有を設定することもできる。ただしその場合は、エクスポート/マウント時の認証に関する注意が必要となる。

5.7 I/O, ファイル

5.7.1 I/O

Colonia では、I/O デバイスを 2 種類に分類する。

ランダムアクセスデバイス ディスク装置など。メモリにマッピングされ、load, store でアクセスされる。通常ランダムアクセスデバイスは、ファイルという概念を通して利用される。

シーケンシャルアクセスデバイス コンソール入出力やテープデバイス等。read, write でアクセスされる。

5.7.2 ファイル

ファイルとは永続的に存在するアドレス空間であり、ユニットのアドレス空間にマウントできるよう、全空間がエクスポートされている。ファイルはその実体をメモリおよびランダムアクセスデバイス上に持ち、その物理的な位置はファイルサーバによって管理されている。ユーザにはただアドレス空間が見えるだけであり、実体の物理的な位置はまったく意識されない(図 9)。

ファイルはひとつのアドレス空間であるが、全アドレス領域に対して実体が存在する訳ではない。ファイルは生成された時点では実体を持たず、ユーザミッションにマウントされた時

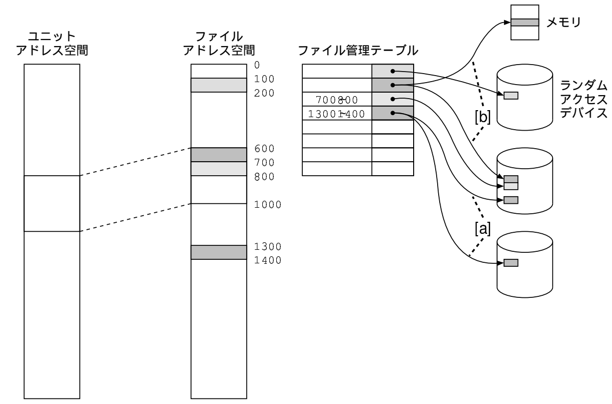

図 9: ファイル

点でもまだ実体は存在しない。アクセスを受けた時点で初めて、アクセスを受けた領域に対する実体が自動的に生成される。なお、ファイル中のある領域に対する実体が不要になった場合には、ユーザが明示的に解放する。

ファイルの実体の物理的な位置を、アクセス状況に応じて最適化するのはファイルサーバの役目である。

- ファイルの実体は、基本的にはランダムアクセスデバイス上に存在する。入出力の並列性を高めるために、ひとつのファイルを複数のランダムアクセスデバイスに分割して格納しても構わない。また、ファイルのある領域に対する実体を、複数作ることも可能である (図 9[a])。
- アクセスを受けると、その領域に対する実体のコピーをメモリ上に作成してやる (図 9[b])。内容の変更を受けた場合は、その内容を適宜ランダムアクセスデバイスに書き戻してやる。この際、元のランダムアクセスデバイスに書き戻さずに、別のランダムアクセスデバイスに新しい実体を作ることも可能である。

このようにファイルの空間と実体の位置を分離することで、並列I/O などによる効率的な入出力が可能となる。

5.8 テリトリ・スケジューリング

第 5.3 節に述べた Colonia の達成目標、すなわち、

- 並列アプリケーションの効率的な実行をサポートし、個々の並列アプリケーションに対して高性能を提供する。
- マルチユーザによる利用の中で、それぞれのユーザが各種資源に対して要求する QoS を出来る限り満足させる。

の 2 点は、テリトリ全域に渡るスケジューリングによってもたらされるものである。

システムの性能を引き出しユーザに資源を配分するという重要な役割を持った、この全計算機資源に渡るスケジューリングを、テリトリ・スケジューリングと呼ぶ。これは、例えば以下のような機構の組み合わせによって実現される。

- ノード内の CPU スケジューリング
- ノード間のユニット移送
- ディスクへのページアウト, ネットワークを介した他ノードへのページアウト (ネットワークページアウト)
- 共有メモリページの実体 (ホームページ) を持つノードの変更

ただし、実際にコンピュータ・コロニーの性能を高める上で重要なのは、上記のメカニズムをどのように利用していくかというポリシーである。なお、その際の判断材料としては次のようなものが挙げられる。

- ハードウェアの静的な情報
  - ノードの持つプロセッサの性能や数
  - ノードの持つメモリやディスクの量
  - アイドル時のノード間の通信性能
- ハードウェアの動的な情報
  - プロセッサの負荷
  - メモリの使用量
  - その時点でのノード間の通信性能
- ソフトウェアの特性
  - 要求する QoS (プロセッサ, メモリ, 通信など)
  - ユーザの設定するユニット配置のプリファレンス
  - ユニット移送にかかるコスト

テリトリ・スケジューリングのポリシーを考える際、システムがマルチユーザで利用されるという点を特に考慮しなければならない。そのため、各種資源の QoS の割り当ては、ユーザ単位が基本となる。ひとりのユーザが一度に大量のミッションを投入したり、大量のユニットを生成したとしても、そのユーザに対する QoS の割り当てが不当に向上するべきではない。また、サーバに仕事の一部を実行させる場合も、資源の利用はクライアント側でおこなったものとしてカウントされるべきである。

5.9 プログラム実行の流れ

コンピュータ・コロニーにおけるプログラムの実行は、次のようにしておこなわれる。

5.9.1 ミッションの生成

並列アプリケーションを実行するにあたり、ユーザはコンピュータ・コロニーにミッションを投入する。ミッションの生成は次のようにしておこなわれる。

1. ユーザがコンピュータ・コロニーにミッションを投入する。これは、コマンドラインからプログラムの実行を指示する作業であったり、あるいは既に動いている何らかのプログラムから新しいミッションを生成する作業であったりする。いずれにせよその実体は、あるユニット (例えば前者の場合は、コマンドラインインタプリタを実現するミッションに属するユニット) がミッションを生成するシステムコールを発行するものである。この際、ミッション内共有領域として利用するアドレス領域を指定する。
2. ミッション生成のシステムコールを受けると、OS はまず、ルートユニットを生成するノードを決定する。この際、最初から負荷分散を考慮してルートユニットの配置を決定してもよいし、簡単のためミッションを投入 (生成) したユニットの実行されているノードに配置してもよい。後者の場合でも、動的な負荷分散により、いずれは最適なノードに配置されることが期待できるためである。

3. ミッション内共有領域のうち初期化が必要な領域(共有コードおよび共有データ)の各ページに対して、ホームページを作成し、初期化をおこなう。各ページのホームノードの決定は、やはりOSがおこなう。できる限り多くのノードに分散させる方法もあるし、また初期状態ではルートユニットしかその領域を利用しないため、ルートユニットを生成するノードにすべて集める方法もある。いずれにせよ、動的な負荷分散により、最終的にはホームノードの最適な配置が達成される。
4. 共有コードを利用するアクティビティ(ユニット)を生成し、実行可能キューに登録する。以上でルートユニットの生成が完了し、これを以てミッションの生成が完了したとする。

5.9.2 ユニットの生成

ユニットがユニットを生成していくことにより、ルートユニットを起点としたユニットの木が構築されていく。ユニットの生成は次のようにしておこなわれる。

1. あるユニットが、ユニットを生成するシステムコールを発行する。
2. ユニット生成のシステムコールを受けると、OSはまず、新規ユニットを生成するノードを決定する。この際、ルートユニットの場合と同様に、最初から負荷分散を考慮してユニットの配置を決定してもよいし、簡単のためシステムコールを発行したユニットの実行されているノードに配置してもよい。
3. ユニットを生成するノードで、まだ当該ミッションのミッション内共有領域が利用可能な状態になければ、利用可能な状態にするため各ページのホームノードとの間に共有メモリを設定する。
4. 共有コードを利用するアクティビティ(ユニット)を生成し、実行可能キューに登録する。以上でユニットの生成が完了する。

ユニットを生成するノードで既に当該ミッションに属するユニットが実行されていたり、あるいはミッション内共有領域を利用可能にするシステムコールが発行されていれば、上記3の手順が不要となるため、ユニット生成時のオーバヘッドを軽減することができる。

5.9.3 動的な負荷分散

並列アプリケーションの実行を高速化するため、あるいはQoSを満足するようにするため、OSは動的な負荷分散をおこなう。負荷分散の技術はいくつか考えられるが、ここではユニットの移送, ホームページの分散, ネットワークページングについて述べる。

ユニットの移送　あるノードに負荷(プロセッサの負荷やメモリ使用量)が集中した場合、あるいはノード間で通信が頻発する場合など、ユニットの移送によって処理効率を向上することができる。

1. OSが負荷分散の必要性を検知し、移送するユニットと移送先ノードを決定する。
2. 移送先ノードでまだ当該ミッションのミッション内共有領域が利用可能な状態になければ、利用可能な状態にするため各ページのホームノードとの間に共有メモリを設定する。
3. 移送元ノードから移送先ノードにユニット固有領域を移動する。この際の転送方式としては、プリページング方式とデマンドページング方式がある。
4. 移送ユニットのアドレス空間を移送先で準備できたら、ユニットの実行状態を転送し実行可能キューに登録することで、ユニットの移送が完了する。

ユニットの生成時と同様に、移送先ノードで既に当該ミッションに属するユニットが実行されていたり、あるいはミッション内共有領域を利用可能にするシステムコールが発行されていれば、上記2の手順が不要となるため、ユニット移送時のオーバヘッドを軽減することができる。

ホームページの分散　ノード間でメモリを共有する際、ホームページを管理するノード(ホームノード)が処理のボトルネックとなる可能性がある。そこで、ノード間共有メモリのホームページを多くのノードに分散させることで、ボトルネックを緩和する。ホームページの分散は、静的にも動的にもおこなわれる。

ネットワークページング　あるノードで主記憶が不足した場合、自ノードのディスクにページアウトするよりも、ネットワークを介して他ノードの主記憶にページアウト(ネットワークページアウト)する方が高速である。ネットワークページングにより、通常のページもシステム中に分散されていく。

ルートユニットを起点として展開されたミッション内のユニット群は、並列性を生かしプロセッサ資源を有効に利用するためにテリトリ中に広く分散したり、あるいは通信のコストを最低限に抑えるためにひとつのノードに集まったりしながら、効率よい実行が進められる。その際、ホームページの分散やネットワークページアウトも利用することで、さらに効率的な実行が達成される。

5.9.4 ユニットの終了

ユニットの終了方式については、まだ仕様が確定していない。

まず、ユニットの親子関係を次のように定義する。

> 「新しいユニットが生成された時、そのユニットを生成するシステムコールを発行したユニットを親ユニットと呼び、新しく生成されたユニットを子ユニットと呼ぶ。」

ここでユニット終了の方式としては、次のふたつが考えらえる。

- 親ユニットは子ユニットの終了を待って終了する。
- 親ユニットは子ユニットの終了を待たずに終了できる。

以下、それぞれの方式について説明する。

親ユニットが子ユニットを待つ方式　親ユニットが子ユニットの終了を待ってから自身も終了する場合、ルートユニットを起点として展開したユニットの木は、リーフ(葉)から順々に終了することになる。本方式ではルートユニットが最後に終了し、それをミッションの終了と定めることができる(図10)。

親ユニットが子ユニットを待たない方式　親ユニットが、子ユニットの終了を待たずに終了できる方式である。

この方式の場合では、終了したユニットの子ユニットをどのように管理するかが問題となる。ここでは、次のふたつの方法が考えられる。

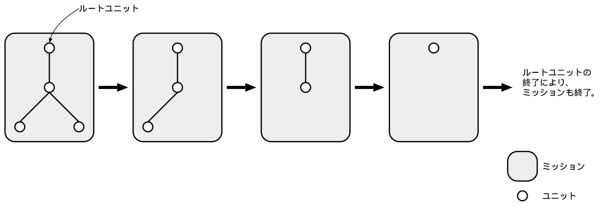

図 10: 親ユニットが子ユニットを待つ方式

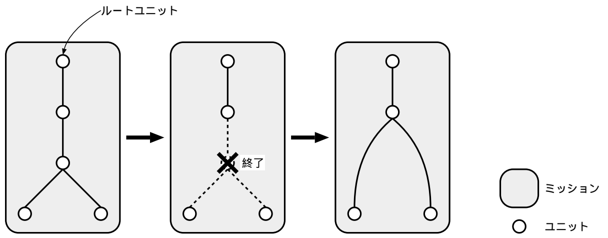

図 11: 終了ユニットの親ユニットの下に登録

- 子ユニットを、終了したユニットの親ユニットの下に登録する (図 11)。
- 子ユニットを、ルートユニットの下に登録する (図 12)。

ただしいずれの場合にせよ、ルートユニットが終了する際には例外的な措置が必要となる。具体的には、次のふたつのうちいずれかが必要となる。

- ルートユニットはミッション内の全ユニットが終了するのを待ち、必ず最後に終了する。
- ルートユニットが先に終了する場合には、ルートユニットとしての役割を他のユニットに引き継ぐ。

両者の比較　「親ユニットが子ユニットを待つ方式」の長所としては、ユニットの管理が容易であることが挙げられる。この方式では、ミッション内ではルートユニットが最後に終了することが保証され、ミッションの管理をルートユニットに任すことができる。しかし、自身のするべき処理は完了していても子ユニットの終了を待たなければならないため、種々の資源を保持したままになるという問題がある。

一方「親ユニットが子ユニットを待たない方式」では、自身の処理が完了すれば子ユニットの終了を待たずに終了できるため、不要な資源はすぐに解放される。しかし、ユニットの木の管理が困難になり、またルートユニットの終了に関して例外が生じるという問題もある。

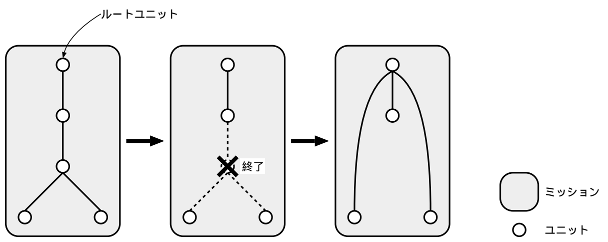

図 12: ルートユニットの下に登録

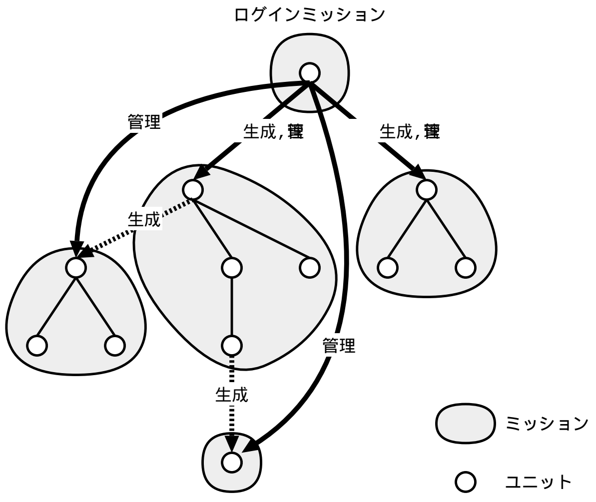

図 13: ログインミッションによる全ミッションの管理

5.9.5　ミッションの終了

ミッションの終了方式についても、まだ仕様が確定していない。

ミッションの親子関係は次のように定義される。

> 「新しいミッションが生成された時、そのミッションを生成するシステムコールを発行したユニットを親ユニットと呼び、新しく生成されたミッションを子ミッションと呼ぶ。また、親ユニットの属するミッションを親ミッションと呼ぶ。」

しかし、ミッションの親子関係はそれほど重要ではない。なぜなら、ミッションは基本的にそれぞれが独立した仕事をおこなうプログラムだと仮定しているからである。あるユニットがミッションを生成した場合、そのユニットは生成したミッション (子ミッション) の終了を待たずに終了することを許す。その結果として、親ミッションは子ミッションの終了を待たずに終了することができる。

ミッションの親子関係はほとんど意味を持たないという立場に立てば、新しく生成されたミッションを、ログインミッション (ユーザがコンピュータ・コロニーを利用する際に、システムからまず最初に割り当てられるミッション) の下に登録すれば、ミッションの管理を容易にすることができる (図 13)。

6　おわりに

本論文では、並列アプリケーションを指向した共有メモリベースの分散システムとしてコンピュータ・コロニーを提案し、さらにコンピュータ・コロニーを実現するためのソフトウェア環境である分散 OS Colonia について述べた。

Colonia はコンピュータ・コロニーのハードウェアが提供する高速・低遅延な通信を積極的に利用し、並列アプリケーションの効率的な実行や動的負荷分散による計算機資源の有効利用、マルチユーザへの適切な資源の配分を目指す。これらの実現のため、Colonia はミッション-ユニットモデルと呼ぶプログラミングモデルを採用した。

現在、Colonia のプロトタイプとして、Solaris オペレーティングシステム上で動作する擬似マイクロカーネルの開発が進められている [5]。今後の課題としては、Colonia カーネルの詳細設計を進めると共に、各種システムサーバの機能についても考察を進めることが挙げられる。

謝辞

本研究の機会を与えてくださった富田眞治教授に深甚な謝意を表します。

また、本研究に関して適切な御指導を賜った中島浩助教授および森眞一郎助教授、数々の有用な御助言を頂いた五島正裕助手に心から感謝いたします。

最後に、本研究の共同研究者である山添博史氏をはじめ、日頃から御討論頂いた計算機工学講座計算機構成分野の諸氏に感謝いたします。

参考文献

[1] 福井俊之, 鈴木茂夫, 中村秀一, 下山朋彦, 数藤義明, 濱口一正, 柴山茂樹: 光バスクラスタシステムの仕様と基本性能の評価, 情処研報, Vol. 119, No. 25, pp. 143–148 (1996).

[2] 文部省重点領域研究「超並列原理に基づく情報処理基本体系」: 第 6 回シンポジウム予稿集 (1995).

[3] Douglis, F., Kaashoek, M. F., Tanenbaum, A. S. and Ousterhout, J. K.: A Comparison of Two Distributed Systems: Amoeba and Sprite, *Computing Systems*, Vol. 4, No. 3, pp. 353–384 (1991).

[4] 前川守, 所真理雄, 清水謙太朗: 分散オペレーティングシステム — UNIX の次にくるもの, 共立出版, pp. 94–97 (1991).

[5] 山添博史: 分散オペレーティングシステム Colonia の共有メモリ管理とそのエミュレータ, 特別研究報告書, 京都大学工学部情報工学科 (1997).

[6] 青木秀貴, 山添博史, 五島正裕, 森眞一郎, 中島浩, 富田眞治: 共有メモリベースのシームレスな並列計算機環境を実現するオペレーティングシステムの構想, 情処研報, 97-OS-34 (1997).

[7] A. S. タネンバウム (引地信行, 引地美恵子 訳): OS の基礎と応用 — 設計から実装, DOS から分散 OS Amoeba まで, トッパン (1995).

[8] J. ベーコン (藤田昭平, 篠田陽一, 今泉貴史 訳): 並行分散システム — オペレーティングシステム, データベース, 分散マルチメディアシステムへの統合的アプローチ, トッパン (1996).

[9] 萩原宏, 津田孝夫, 大久保英嗣: 現代オペレーティングシステムの基礎, オーム社 (1988).

[10] 前川守: オペレーティングシステム, 岩波講座ソフトウェア科学, No. 6, 岩波書店 (1988).

[11] Protić, J., Tomašević, M. and Milutinović, V.: Distributed Shared Memory: Concepts and Systems, *IEEE Parallel & Distributed Technology*, Summer, pp. 63–79 (1996).

[12] Blumrich, M. A., Li, K., Alpert, R., Dubnicki, C. and Felten, E. W.: Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer, *21st Annual International Symposium on Computer Architecture*, pp. 142–153 (1994).

[13] Gillett, R. B.: Memory Channel Network for PCI, *IEEE Miciro*, February, 12–18 (1996).

[14] SPARC International, INC. (相越克久, 田中長光 訳 / 多田好克監訳): SPARC アーキテクチャ・マニュアル バージョン 8, トッパン (1992).

[15] ロス P. ネルソン (富士ソフトウエア教育出版部 訳): 80386/80486 プログラミングガイド, 富士ソフトウエア株式会社 (1992).

[16] 日本サン・マイクロシステムズ株式会社: SPARCstation 20 システム・アーキテクチャ, Sun Expert, No. 16 (1994).