プログラミング-言語・基礎・実践- 21－3
(1995. 3. 9)

スケーラブルプログラミングシステム：CC++

藤田昭平
東京工大 大学院 情報理工学研究科
fujita@cs.titech.ac.jp

あらまし 並列分散コンピューティングシステムを広く普及させるためには、スケーラブルでポータブルな並列分散プログラムの開発環境を完備しなければならない。

このための1つの方法は、高水準言語を用いて問題の仕様記述に近いレベルで並列分散プログラムを記述し、ネットワーク化された(マルチプロセッサ)ワークステーション、マルチコンピュータ等種々のシステムで効率よく実行できるようにすることである。

本報告では、Caltech および ANL で開発中のスケーラブルプログラミングシステム：CC++ の特徴、処理系について述べ、"Collaboratory" 等の適用分野を論じる。

キーワード 並列分散プログラミング、スケーラビリティ、CC++、Collaboratory

Scalable Programming System : CC++

Shohei FUJITA
TITech, Graduate School of Computer Science
Meguro-ku, Tokyo 152, Japan
fujita@cs.titech.ac.jp

Abstract *The difficulty of developing scalable and portable parallel/distributed programs is the major obstacle to the more widespread use of parallel/distributed computing systems.*

One approach to overcome this obstacle is to devise high level programming notations that allows parallel/distributed programs to be written close to the level of problem specification, yet still be compiled to execute efficiently across a variety of networks of (multiprocessor) workstations and multicomputer systems.

This report describes a scalable parallel/distributed programming system : CC++ being developed at Caltech and ANL ,and evaluates its applicability to "Collaboratory"etc.

Key Words Parallel/distributed programming, Scalability, CC++, Collaboratory

1 はじめに

1980 年代に “研究室” レベルで試験的に使用されてきた並列マシンは、1990 年代に入って商用化され、“実社会” での問題に適用されつつある。しかし、“雄大な挑戦 (GC)” 問題の解決に際しては、

- **大規模科学技術計算、シミュレーション**
- **ネットワーキング**
- **大規模 (分散) データベース**

を統合しなければならない。このための “コンピューティング & コミュニケーション” 能力は充分であるとはいえない。さらに “高性能” なコンピューティングシステム、すなわち、**“3T” コンピューティングシステムを目指して、**ハードウェア、ソフトウェアおよび応用分野の研究を三位一体に推進すべきである [1] [2] [3] [4]。

並列分散システムの重要な機能の 1 つは

- **スケーラビリティ(Scalability)**

である。現在、並列分散処理の分野で最も必要なことは**スケーラブルな**並列分散プログラミング環境を完備し、従来の単一プロセッサシステムと同様なソフトウェア開発環境を整えることである。

本報告では、オブジェクト指向言語に関数型言語の機能を融合したスケーラブルな**構成型並列分散プログラミングシステム：CC++**の特徴、処理系および適用分野について述べる。

2 MPP ソフトウェア技術の現状と研究課題

2.1 応用分野とアルゴリズム

エンドユーザは高度な計算アルゴリズムには詳しいが、(一般に)MPP システムそれ自体にはあまり慣れていない。MPP システムの厄介なプログラミングを容易にするソフトウェア開発ツールが、当面差し迫った問題である。MPP の市場および GC 応用分野は、現在それほど大きくない。今後の発展の誘因となるような役立つ事例を追求すべきである。

“スケーラブル” な応用分野が充分に解明されていない。並列分散プログラミングのパラダイムに関する理解が不充分である。実行性能の最適化とポータビリティに関するトレードオフをどうするか？

アルゴリズムとコードのポーティングを強力に支援すべきである。“缶詰” パッケージでは実行性能の点で一般的過ぎるし、マシンアーキテクチャの観点からはあまりにも特殊化されている。“型枠” 的なアプローチを推進すべきである。この型枠は汎用的なコードとし、特別な要求あるいは特殊なシステムのために、エンドユーザがこれを自由に変更し最適化できるようにすべきである。

並列分散計算用の**非同期アルゴリズム**は、高性能コンピューティングのために極めて重要である [1]。高水準言語でアルゴリズムを記述することにより、ソースコードへの変換は容易になるが、マシンに依存した最適化の問題は残る。このような問題は、分散メモリアーキテクチャのマシンでは特に顕著である。共有アドレス空間を有している並列マシンでは、それほど問題にならない。このようにマシンアーキテクチャが種々あるために、それぞれに適したアルゴリズムを開発しなければならない。

2.2 可視化技術とデータベース

リアルタイムシミュレーションから得られる多量のデータの迅速で高度な解析手段である科学的可視化は、MPP システムを導入する誘因の 1 つである。また、応用ソフトウェアの開発に際しても、非常に有効な手段である。

しかし、ディスク、I/O の帯域制限のため、現在使用されている可視化システムの性能は満足できるものではない。これを解決する方法は、**分散可視化環境**を構築することである。すなわち、MPP、グラフィックワークステーション、データベースサーバを “超” 高速通信網でネットワーク化し**分散メタコンピュータ**を構築すべきである。

2.3 ソフトウェア開発ツール

MPP 応用ソフトウェアの開発に際しては、並列分散プログラムの**デバッグ**のためのツールが必要である。並列分散プログラムでは、誤りを含んでいる動作を解析する仕事は複雑になる。並列分散プログラムでは、容易に再現できないような**過渡的なエラー**が潜んでいることがある。共有メモリシステムでは競合問題が、メッセージパッシングシステムではタイミングが原因となる。

応用コードが正しく実行されていても、その実行性能は利用可能なプロセッサの数に大きく依存する。

性能評価ツールは、次のような問題に答えねばならない。

- プログラムの修正により、性能がどれくらい改善されるか？
- 適用問題およびコンピュータシステムのスケールを変えることにより、性能はどのように変化するか？

以上の問題は、応用ソフトウェア開発者のみならず、システムソフトウェアおよびハードウェアの設計者にとっても、次世代の高性能コンピューティングシステムを開発するために必要不可欠である。

2.4 プログラミング言語とコンパイラ技術

言語/コンパイラは、エンドユーザとマシンの実行環境とのインタフェースとなるものである。これらは、コンピューティングシステム全体の性能、使いやすさ、応用コードのポータビリティに大きな影響を及ぼす。唯一のプログラミングパラダイムだけでは、エンドユーザの種々の問題に対処できない。データ並列、タスク並列、オブジェクト指向、関数型等幾つかのプログラミングパラダイムを融合する必要がある。

言語/コンパイラは、それぞれ別々に開発されたプログラムモジュールの“再利用”を促進する必要がある。それぞれ別々に開発されたプログラムモジュールから1つのプログラムを構成する方法は、MPPシステムのソフトウェア開発に大きく貢献する。

言語/コンパイラの開発は、次の三段階に渡って行われるべきである。

- 研究用のプロトタイプ
- さらに進んだ開発用のプロトタイプ
- 商品化された製品

この際、エンドユーザと言語/コンパイラ開発者の協調作業が大切である。

2.5 オペレーティングシステムと実行環境

ハードウェアおよびファームウェア化された計算資源との抽象的なユーザインタフェースをもたらすのが、オペレーティングシステムの役目である。MPPシステムに代表されるような新しいコンピュータシステムの出現により、オペレーティングシステムに対する要求も変化している。スケーラブルなシステムソフトウェアの開発は重要な研究課題である。

3 構成型並列分散プログラミング

大規模システムの解析および設計では、これを要素に分割し、その後個々の要素を結合する手段がしばしば用いられる。この際、個々の要素が有している性質が、各要素の結合により構成されたシステム (Composite Systems) で継承されるかが問題になる [5]。

それぞれの要素プログラムを並列に構成することにより構築されるプログラムを、構成型並列プログラム (Compositional Parallel Program) という。ただし、この構成型並列プログラムは各要素プログラムの性質を継承している [6]。

本章では、オブジェクト指向言語に関数型言語の機能を融合したマルチパラダイムの構成型並列分散プログラミング言語 CC++ の特徴、処理系および適用分野について述べる。

3.1 並列分散プログラミングの問題点

並列分散プログラミングは、なぜ“困難”なのか？

- マシンアーキテクチャに大きく依存し、ポータビリティがない、
- プログラムの再利用が困難である、
- プログラムのテスト、デバッグ、さらに正当性の検証が困難である、
- 性能面でのオーバヘッドが大きい、

等々の理由により、並列分散プログラミングは“あきあきする厄介な”仕事であると言われている。並列分散システムの多くの“潜在的な”利点にもかかわらず、このような障壁が並列分散システムの“実用化”を阻害している。

3.2 構成型並列分散プログラミング言語：CC++

構成型並列分散プログラミング言語：CC++[1] は MIMD アーキテクチャの MPP および 分散メタ

[1] Caltech と ANL で開発中。

コンピュータシステムのためのプログラミング言語である。

CC++は逐次言語 C++ に幾つかの簡単な拡張を加えたものである。CC++ は C, C++ のスーパセットであり、正しい C あるいは C++ プログラムは、また正しい CC++ プログラムである。C および C++ との整合性により、C, C++ のユーザは CC++ へ容易に移行できる。

CC++ は、以下に示す8個の機能を追加することにより、C++ を拡張する。

- **par**

  並列に実行される "文" を囲むブロック。

- **parfor**

  並列に実行される "繰り返しループ"。

- **spawn**

  並列に実行する新しい制御の "スレッド" を生成する。

- **atomic**

  並列に構成されたオブジェクトの "実行順序" を制御する。C++オブジェクトへのアクセスの排他制御を行う。

- **sync**

  単一代入変数で、"同期" のために使用される。

- **processor object**

  計算の分散 ("分散オブジェクト") を記述する。

- **global**

  プロセッサオブジェクトをリンクする "ポインタ" であり、プロセッサオブジェクト間の "通信" を記述する。

- **data transfer**

  プロセッサオブジェクト間で、どのようなデータを転送するかを記述する。

このような僅かな拡張と単純さにもかかわらず、C++ の記述能力と併せて、CC++ は**並列分散プログラミング**のための豊富で強力な記述能力を有している。

CC++ の基本的な特徴を要約すると、

- CC++ は、オブジェクト指向言語 C++ を拡張し、関数型言語の機能を融合した**マルチパラダイムプログラミング言語**である。
- CC++ は、一般的な共有メモリ型プログラミングモデルを提供している。**分散オブジェクト**へのアクセスは、グローバルポインタによる。
- CC++ では、**決定的プログラム**および**非決定的プログラム**の両方を記述できる。
- CC++ では、全ての代入操作を値の**単一代入**として扱う。
- CC++ は**形式的手法**を支援することを目的としている。これにより、正しいプログラムの開発が促進され、テスト、デバックの "手間" が減少する。

3.2.1 分散コンピューティング

複数の分離アドレス空間に跨る計算を分散コンピューティングという。分離アドレス空間のスレッドは同一のメモリにアクセスできない。あるアドレス空間から他のアドレス空間へデータを渡すには、これら2つのスレッドが通信をする必要がある。このような通信は、しばしば時間がかかる。

通信に時間がかかるので、データのどの部分をどのアドレス空間に割り当てるかが重要になる。それぞれのスレッドは、頻繁に使用するデータへのアクセスを速くしたい。すなわち、同一アドレス空間でアクセスしたい。各スレッドはデータの大部分に速くアクセスできるように、利用できるアドレス空間に計算を分散させる。

C++ オブジェクトでは、ある計算に関係しているデータはまとめてグループ分けされる (member function)。ただし、C++ は同一アドレス空間を前提にしている。CC++ は、この考えをプロセッサオブジェクトへと拡張する。一連のデータおよびこれに関係する計算を、1つの**プロセッサオブジェクト**にグループ分けする。それぞれのプロセッサオブジェクトは分離アドレス空間にある。

もちろん、各計算が関係する全てのデータに速くアクセスできるように、分割できるとは限らない。CC++ では、**アクセスに時間を要するデータと、アクセスにあまり時間がかからないデータを区別する。**

アクセスに時間を要する(他のアドレス空間またはプロセッサオブジェクトの)データを参照するポインタを**グローバルポインタ**という。短時間にアクセスできる(同じプロセッサオブジェクトの)データを参照するポインタはローカルポインタと言われる。

グローバルポインタを逆参照することにより、参照された値をフェッチし、他のプロセッサオブジェクトへの通信が生成される。この通信の内容は、**データ転送関数**という CC++ の仕組みを用いて記述される。

CC++ での計算過程は、プロセッサオブジェクトの集合から成る。それぞれのプロセッサオブジェクトは、従来の C++ オブジェクトから構成されている。以下では、

- プロセッサオブジェクト (*processor object*)
- グローバルポインタ (*global pointer*)
- データ転送関数 (*data transfer function*)

をさらに詳しく説明する。

3.2.2 プロセッサオブジェクト

プロセッサオブジェクトはデータと計算の集合であり、1つのアドレス空間を定義する。CC++ では、それぞれのプロセッサオブジェクトは分離アドレス空間にある。ただし、各プロセッサオブジェクトは、必ずしも物理的に相異なるアドレス空間に割り当てられなくてもよい。

アプリケーションの記述に使用される仮想的なアドレス空間(プロセッサオブジェクト)と、これをインプリメントするために使用される物理的なアドレス空間を区別することが大切である。これにより、アプリケーションを記述する問題と使用できる物理的な処理資源へこの(アプリケーション)プログラムを割り当てる問題を別々に扱える。すなわち、抽象化されたオブジェクトを用いてアプリケーションを記述する。その後で、このオブジェクトから使用できる物理的な処理資源へのマッピングを定義する。使用できる物理的な処理資源が変わっても、(アプリケーション)プログラムは影響されない。物理的な処理資源の変化に応じて、このマッピングだけを変えればよい。これにより、プログラムのポータビリティが高められる。

CC++ を用いてコンパイルされた実行可能コードに型を割り当てることにより、プロセッサオブジェクト型が定義される。実行可能コードを割り当てるためのプロセッサオブジェクト型がコンパイラオプション **-ptype=**を用いて定められる。実行可能コードでは、型を宣言しなければならない。こうしないと、リンク時のエラーになる。

プロセッサオブジェクトを実行可能コードとして定義することは、プロセッサオブジェクトに対して2つの型のメンバ:非明示的と明示的なものがあることを意味する。非明示的なメンバは、実行可能コード内のスコープでの関数とオブジェクトである。一方、明示的なメンバ関数は、プロセッサオブジェクト型で明示的に宣言されたものである。非明示的なメンバはプロセッサオブジェクト型の保護されたメンバであり、プロセッサオブジェクトへのグローバルポインタを用いてアクセスできない。

プロセッサオブジェクトは C++ オブジェクトと同様な動作をする。データメンバを貯蔵し、そのデータでのメンバ関数を遂行するようにリクエストされる。プロセッサオブジェクトに対するグローバルポインタを介してプロセッサオブジェクトのメンバ関数を呼び出すと、そのメンバ関数を遂行するために制御のスレッドが生成される。このメンバ関数が終了するとスレッドも終了する。複数のメンバ関数がプロセッサオブジェクトで同時に実行される。

また、**プロセッサオブジェクト内に並行性 (intra-object concurrency)** が存在する。

3.2.3 グローバルポインタ

複数の分離アドレス空間に跨る計算として分散コンピューティングを定義し、1つのアドレス空間に対して1つのプロセッサオブジェクトを定義した。これらのプロセッサオブジェクトがどのように定義され、生成されるかを 3.2.2 節で述べた。本節では、生成されたプロセッサオブジェクトが、データを交換するために、グローバルポインタをどのように使用するかを論じる。

CC++ には2つのポインタがある。すなわち、グローバルポインタとローカルポインタである。グローバルポインタは、その計算に関係する任意のプロセッサオブジェクトのアドレスを参照できる。ローカルポインタは、生成された場所でのプロセッサオブジェクトのアドレスだけを参照できる。グローバルポイ

ンタはアクセスに時間を要するデータを指す。一方、ローカルポインタはアクセスに時間がかからないデータを指す。CC++では、プロセッサオブジェクト間の参照とプロセッサオブジェクト内での参照がある。

グローバルポインタは、ローカルポインタと同様に使用できる。グローバルポインタが逆参照されると、それが参照する値が返される。グローバルポインタがオブジェクトを参照すると、このオブジェクトを介してメンバ関数が呼び出される。どちらの場合も、オブジェクトは別のプロセッサオブジェクトにあるので、値をフェッチする、あるいは関数を呼び出すために非明示的な通信が行なわれる。キーワード **global** を用いてグローバルポインタが宣言される。

グローバルポインタにより参照された値をフェッチするのに必要な通信は透過的で、あたかもローカルポインタであるかのように、グローバルポインタを含む式を書ける。

オブジェクトが存在しているプロセッサオブジェクトへの引数の転送と同様に、グローバルポインタにより参照されたオブジェクトのメンバ関数を呼び出すのに必要な通信は透過的である。

このグローバルポインタを通じての関数呼び出しメカニズムは、**遠隔手続き呼出し (RPC)** による。それぞれの RPC は遠隔プロセッサオブジェクトで別々の制御のスレッドを生成する。したがって、幾つかの RPC を同時に実行できる。不定変数の危険な共有を避けるために、**atomic** と **sync** を使用する。関数呼び出しのセマンティックスは、RPC により保存されている。すなわち、この関数が遠隔プロセッサオブジェクトで終了するまで、関数呼び出し文は終了しない。

CC++ には、遠隔呼出しされた関数の引数と戻し値が、プロセッサオブジェクト間でどのように転送されるかを制御するメカニズムがある。3.2.4 節で、このメカニズムを説明する。

グローバルポインタは、必ずしも他のプロセッサオブジェクトのアドレスを参照しなくてもよい。参照されたアドレスがこの関数が呼び出されたプロセッサオブジェクトにある場合には、ローカルポインタによる関数呼び出しと同じになる。

遠隔呼出しされた関数は、ローカルポインタ、参照、あるいはアレイ引数を持っていない場合がある。1 つのプロセッサオブジェクトから他のプロセッサオブジェクトへとこれらの型がコピーされないために、このような事態が生じる。同様な理由で、遠隔呼出しされた関数はローカルポインタ、参照、あるいは アレイを返さない場合がある。いずれかの制約を侵すと、コンパイル時エラーとなる。

3.2.4 データ転送関数

グローバルポインタを介して引数を持っている関数が呼び出されると、これらの引数は遠隔プロセッサオブジェクトおよびこれらのコピーを呼び出した関数へとコピーされる。同様に、関数の戻し値もこの関数を遠隔呼出ししたプロセッサオブジェクトへと転送される。

これらが基本型の場合には、この引数を転送することは簡単である。しかし、これらが**ユーザ定義型**の場合、特にローカルポインタを含んでいる場合は複雑になる。ローカルポインタは、これが生成されたプロセッサオブジェクトでのみ有効であることに注意。

任意の型がどのようにして転送されるかを記述するために、CC++ では全ての型に 1 対の関数があてがわれる。この関数は、その型を他のプロセッサオブジェクトへどのようにして転送するかを定義する。これらの関数は、その型のための**データ転送関数**と言われる。

ある型に対して 1 度定義されると、その型を転送するために、これらの関数は**コンパイラ**により自動的に呼び出される。これらの関数を明示的に呼び出す必要はない。この型の引数をとる関数を、グローバルポインタを介して呼び出すことにより、これらは非明示的に呼び出される。また、遠隔呼出しされた関数がその型の値を戻す時に、これは自動的に呼び出される。

関数

CCVoid& operator<<(CCVoid&, const TYPE& obj-in);

は、**TYPE** がどのように**包まれる**かを定義する。オブジェクト TYPE が他のプロセッサオブジェクトに転送されねばならない時は、コンパイラにより呼び出される。

同様に、関数

CCVoid& operator >> (CCVoid&, TYPE& obj-out);

は、TYPE がどのように開かれるかを定義する。オブジェクト TYPE が他のプロセッサオブジェクトから受けとられると、コンパイラにより呼び出される。終了すると、obj-out は 最初のプロセッサオブジェクトでの operator<<に対する引数として使用された obj-in のコピーとなる。

型 CCVoid は、iostream ライブラリの class ios に類似したコンパイラ定義の型である。C++ の入出力ストリームに類似したデータ転送関数を使用する。

3.3 CC++ の処理系

本節では、CC++ の処理系の概要を述べる。この処理系 (プロトタイプ) の目標は、機能的に正しく動作することを実証することである。

ここでは、CC++ プログラムを正しい C++ プログラムへとプリプロセスする。この結果得られる C++ プログラムを、C++ コンパイラを用いて実行可能コードに変換する。 CC++ の処理系は、

- CC++ から C++ へのトランスレータ
- マルチスレッド実行時支援システム

から成る。

3.3.1 CC++ から C++ へのトランスレータ

CC++ を C++ へと変換するために、CC++ プログラムを構文解析し、構文ツリーを作る。この構文ツリーを、C++ の構文ツリーへと変換する。CC++ の構文解析は、GNU C++(G++) を改変したものである。

この変換過程での最大の難点は、セマンティクスのチェックである。すなわち、CC++ プログラムの意味と、これがトランスレータにより変換された C++ プログラムの意味は、“等価” でなければならない。

3.3.2 マルチスレッド実行時支援システム

実行時支援システムは、システムの抽象的なモデルをコンパイラに提供し、ポータビリティを高める。このモデルは 5 つの抽象化 : ノード、コンテキスト、スレッド、グローバルポインタ、リモートサービスリクエストより成る [7]。

- ノード

  ノードは物理的な処理資源を示す。ノードはマシンの名前、(マルチコンピュータでは) プロセッサの番号で識別される。プログラムで規定されたノードの数により処理能力が決められる。ノードの数は動的に変化する。

- コンテキスト

  実行可能なコードを含むアドレス空間をコンテキストという。ノードには、1 つ以上のコンテキストが配置される。コンテキストは動的に生成され消滅する。

- スレッド

  コンテキスト内にスレッドが生成され、1 つ以上のスレッドで計算が行われる。コンテキスト内のスレッドは全て同等である。したがって、計算は非同期に行われる。

- グローバルポインタ

  グローバルポインタはノードとコンテキストで指定される。スレッドは同一のコンテキスト内のデータを、ローカルデータとしてアクセスできる。しかし、(同一または異なるノードでの) 異なるコンテキスト内のデータにアクセスするには、グローバルポインタを介さねばならない。

- リモートサービスリクエスト

  スレッドは、リモートサービスリクエストを用いて、リモートノードで行われる動作をリクエストできる。グローバルポインタで指定されたコンテキストで、指定された関数の実行が行われる。リモートサービスリクエストは RPC とは異なる。

ポータビリティを高めるために、各種のスレッドおよび通信プロトコルはスレッドモジュール、プロトコルモジュールとしてカプセル化されている。現在支援されているのは、POSIX スレッド、DCE スレッド、C スレッド、Solaris スレッドである。

3.4 CC++ の適用分野

CC++ は、以下に示すような幅広い分野に適している。

- **共有メモリモデルおよび分散メモリモデル**

  CC++ は、共有メモリマルチプロセッサ、分散メモリマルチコンピュータ、さらにヘテロジニアスなネットワークベースの (分散メタコンピュータ) システムで使用できる。

- **粒度**

  CC++ は細粒度 (fine-grain) から疎粒度 (coarse-grain) まで幅広い並列分散計算を記述できる。細粒度の並列性は、小さなスレッド間の "頻繁な通信" を必要とする。疎粒度の並列性は、大きな分散オブジェクト間の "疎な通信" に適している。

- **データ並列およびタスク並列プログラミング**

  データ並列およびタスク並列の両方 (およびその組合せ) を記述できる。

"構成型" 記述法 ("Compositional" Notation) を提供する CC++ は、

♠ **大規模科学技術計算、シミュレーション**

♡ **グラフィッカルユーザインタフェース (GUI)**

♣ **リアクティブ システム**

等幅広い分野での使用が期待される。

4 おわりに

並列分散コンピューティングは、"潜在力" のある技術 ("Precompetitive" Technology) である。この潜在力を発揮させるには、"スケーラブル" な並列分散プログラミング環境を完備しなければならない。

本報告では、オブジェクト指向言語に関数型言語の機能を融合した**構成型並列分散プログラミング言語：CC++** の特徴、処理系および適用分野について論じた。

多領域に跨る研究を "効果的に" 行なう

- "Collaboratory" = "Collaboration" + "Laboratory"

では、分散メタコンピュータシステムを構築しなければならない [13]。本報告で論じた構成型並列分散プログラミングシステム：CC++ は、このような **"Collaboratory"** での使用に適している。

参考文献

[1] S. Fujita, "Distributed MIMD Multiprocessor System with MicroAda/SuperMicro for Asynchronous Concurrent Newton Algorithms," **Proc. 5th ACM – SIGSMALL Symp.**, pp.49 - 59, (1982).

[2] Office of Technology Assessment, **"High Performance Computing & Networking for Science,"** (Sep. 1989).

[3] **Workshop on System Software and Tools for HPC Environments**, (Pasadena, Apr. 14 - 16, 1992).

[4] **Workshop and Conference on Grand Challenges Applications and Software Technology**, (Pittsburgh, May 4 - 7, 1993).

[5] S. Fujita et al.,"Controllability and Observability of Composite Time-Varying Linear Systems, " (in Japanese), **計測自動制御学会 論文集**、4 巻、3 号、234 - 241、(1968).

[6] K.M. Chandy et al.,"Compositional C++ : Compositional Parallel Programming," **Caltech CS-TR-92-13**, (1992).

[7] I. Foster et al., "Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems," Preprint,**ANL/MCS**, (1994).

[8] B. Randell et al., **Advanced Course on Functional Programming and its Applications**, (20 - 31 July 1981).

[9] 藤田、細谷 (監訳) **Ada：言語とプログラミング方法論**、丸善、(1990).

[10] 藤田、"スケーラブルプログラミングシステム：背景と目的"、**情報処理学会、第 49 回全国大会講演論文集** (5), pp.5-21&22, (1994).

[11] 藤田、"コンピュータサイエンス & エンジニアリングの "健全なる" 成長のために"、**情報処理学会、新しい時代の情報処理教育カリキュラムシンポジウム論文集**、pp.1 - 8, (1994).

[12] 藤田、"並列分散コンピューティングのためのスケーラブルプログラミングシステム"、**情報処理学会、コンピュータシステムシンポジウム論文集**、pp. 79 - 86, (1994).

[13] 藤田、"分散マルチメディアシステムによる Collaboratory の構築に向けて"、**情報処理学会、情報学シンポジウム論文集**, pp. 73 - 82, (1995).