超並列コンピュータ SR2201 を用いた基本アルゴリズムの性能評価

山本富士男[1]・川上芳尚[2]・荒木智行[1]

[1] 情報工学科
[2] 日本電信電話株式会社（平成９年３月情報工学科卒業）

Performance Evaluation of Basic Parallel Algorithms on the Hitachi SR2201

Fujio YAMAMOTO[1)], Yoshitaka KAWAKAMI[2)], Tomoyuki ARAKI[1)]

Abstract

Basic parallel algorithms including quick sort and LU decomposition were investigated on the Hitachi SR2201 distributed memory parallel computer. We composed several parallel quick sort algorithms with different data distribution. But we didn't get high performance because of heavy inter-processor communication. On the other hand, we obtained very high performance in parallel LU decomposition with partial pivoting. In this case, column-wise cyclic data distribution and partial send/receive without broadcasting were adopted. Additionally, a large-scale LU decomposition known as an out-of-core problem was solved. As foreseeing, for such problems, we actually recognized that several distributed disks or high-speed secondary memory must be supplied in the parallel computer system.

Key Words: Parallel Computing, Distributed memory, Quick sort, LU decomposition, Out-of-core

1. まえがき

本論文の目的は、文部省大型設備補助金を得て本学に導入された超並列コンピュータ SR2201 上での、基本並列アルゴリズムの性能特性を明らかにすることにある。

並列コンピュータのアーキテクチャは、共有記憶型と分散記憶型に大別される。共有記憶型は、並列コンピュータの構成要素である多数のプロセッサが一つのメモリ空間を共有する。その場合、メモリへの各プロセッサからのアクセスは、特定のパスを通ることになるので、多数のプロセッサを接続した場合は、そこがボトルネックとなってしまう。

このため、かなり多くのプロセッサを接続できる、いわゆる超並列コンピュータの構成は、分散記憶型が主流となっている。すなわち、各プロセッサは各々固有のメモリ空間を持ち、直接的にはそこへしかアクセスできない。他のプロセッサのメモリにある情報の取得は、プロセッサ間に設置されている相互通信網を使うデータ転送によって行うことになる。本学に導入された SR2201 も分散記憶型である。

分散記憶型超並列コンピュータに対する、アプリケーション側からの期待は非常に大きいものがある反面、そのためのプログラミングは一般に難しいと言われている。[1, 4] 並列度の高い処理方式を考え、計算とデータを適切にプロセッサに分配し、さらに必要なプロセッサ間の通信処理も規定しなければならないからである。高性能な自動超並列化コンパイラがあればそれらは大いに緩和されるが、現状ではそれに至っていない。すなわち、人手により、並列用プログラムをＣ言語などで書く必要がある。どのようなアルゴリズムを使って、どのようなプログラムを書けば高い性能を得ることができるかについての知見を貯える必要がある。

本論文では、その一環として、まず、SR2201 の基本ソフトウェアに含まれるプロセッサ間通信の性能を確認する。次に、数種類の基本的なアルゴリズム（整列、連立一次方程式解法関係）を超並列化して、Ｃ言語プログラムを作成し、その実行性能を分析する。

2. 超並列アルゴリズムの性能

2. 1 並列処理効率を下げる要因

本論文では、並列処理効率を下げる要因を、大きく二つに分類する。一つは、アイドル時間である。これは、プロセッサが、何も処理をせず空いている時間で、負荷分散が不均等でプロセッサ間の処理時間に差のあるとき、他のプロセッサからのデータを待つとき、競合すなわち、全プロセッサまたは、複数のプロセッサで、ディスクなど共有資源に同時にアクセスしたとき、プロセッサ間で

同期をとるときなどに発生する。

もう一つは、並列化によるオーバーヘッドである。これは、データの分割処理、プロセッサ間通信に要する処理など、逐次処理では必要ないが並列化するために必要な処理と、並列化できず、全プロセッサまたは、複数のプロッセッサでおこなわれる同一の処理である。

超並列アルゴリズムの性能は、アイドル時間とオーバーヘッドをどれだけ減らせるかによって決まる。すなわち、並列化する際に重要なのは、アイドル時間とオーバーヘッドの二つを、主処理に対して、どれだけ少ない割合にできるかである。

ただし、この二つの占める割合が少ないというだけで、良い性能が出ているとは言い切れない。なぜならば、並列に処理されている部分が、遅いアルゴリズムであるために、処理時間が増え、結果的にアイドル時間とオーバーヘッドの割合が少なくなっている場合もあるからである。並列アルゴリズムの性能は、プロセッサ数とデータ量に依存する。一般にプロセッサ数が少ないときや、データ量が多いときに、高い効率が出やすい。

また、アイドル時間とオーバーヘッドは、データ分割に大きな影響を受けるので、分散型アーキテクチャにおけるアルゴリズムを検討する上では、データ分割も、同時に考慮に入れる必要がある。

2．2 並列処理におけるデータアクセスの分類

アルゴリズムとデータ分割を検討する上で必要な、並列処理におけるデータアクセスについて、本論文では、独立型、共有型、排他型の３つに分類する。

独立型のリソースは、分散メモリ型の主記憶メモリなどで、プロセッサが常に独立にアクセスでき、アクセスが競合しない。並列度を高くするため、できるだけ、この形の処理が多くなるようにデータを分割する。

共有型のリソースは、共有メモリ型の主記憶メモリなどで、アクセスしているプロセッサがあるときは、他プロセッサはアクセスできない。アクセスが重なると競合し、処理の待ちが生じる。この待ちは、並列処理ではアイドル時間となるので、並列度を高めるためには、プロセッサのアクセスをづらす必要がある。

排他型のリソースは、分散メモリ型の他プロセッサのメモリなどで、データに直接アクセスできないため、プロセッサ間通信によって、送受信する必要がある。このとき、送信プロセッサ側の送信処理と受信プロセッサ側の受信処理は、オーバーヘッドとなり、また、受信側が受信処理を開始したときに、送信側がまだ送信処理を開始していない場合は、受信側はデータを待ち、アイドル時間となる。

3． 超並列コンピュータ SR2201 の概要

3．1 ハードウエア仕様

本学に設置されている SR2201 のハードウェア仕様は表１のとおりである。個々のプロセッサのことをノードとも呼ぶ。また、プロセッサ間通信のことをノード間通信とも呼ぶ。

表１ SR2201 のハードウェア仕様

構成	分散メモリ型 MIMD
ＣＰＵ	日立製 RISC プロセッサ
ノード数	16 台（2 台は、I/O 等専用）
ピーク性能	4.8GFlops（300MFlops*16）
主記憶容量	2.5Gbyte（64Mbyte*8 + 256Mbyte*8）
ノード間結合網	二次元クロスバ（通信速度 300Mbyte/sec）
通信方式	メッセージパッシング

本機には、超並列オペレーティングシステムの下で動く並列化支援システムとして ParallelWare [2, 3]が提供されている。これによって提供される関数を用いて、並列処理プログラムを作成する。

作成することのできるプログラミングモデルとしては、HOST-NODE モデルと CUBIX モデルの２つがある。HOST-NODE モデルは、一つのノードが HOST ノードとなり他のノードの実行を管理するモデルである。

CUBIX モデルでは、ユーザはホストプログラムを書く必要はなく、全ノードで同一プログラムが実行される。（ただし、各ノードが同一の計算を実行するとは限らない。）CUBIX モデルによって、広範な応用問題を十分に記述できるので、以下では、これを用い、Ｃ言語によるプログラムを作成することにした。

3．2 通信処理の基本性能

PARALELL WARE のライブラリに提供される関数のうち、今回使用する主要な通信処理用関数について、その性能を測定しておく。なお、以下の実測は、すべて平成９年２月に実施したものである。（その後、オペレーティングシステムと言語コンパイラのバージョンアップがあり、性能はこれより向上している場合があると考えられる。）

3．2．1 exwrite/exread 関数

exwrite 関数は、データを任意のノードに送信する関数で、任意のバイト数を送信することができる。exread 関数は、 exwrite 関数によって送信されたデータをブロッキング受信する関数である。ブロッキング受信とは、データが送信されていなければ送信されるまで待ち、送信されてから受信する関数である。送信処理（write1）と受信処理（read1）を同時に行った場合の測定結果と、送信処理（write2）が完全に終わってから受信処理（read2）をした場合の測定結果を図１に示す。

ここでの測定時間は、通信の立ち上がりなどのオーバーヘッドを含めた、プログラム上で実際の通信にかかるトータルの時間である。この時間をもとに転送速度を計算しているので、純粋な転送速度とは厳密には異なる。しかし、4byte のときにかかる時間がすべてオーバーヘッドであるとして計算しても 8Mbyte などの転送速度には、ほとんど影響しない。

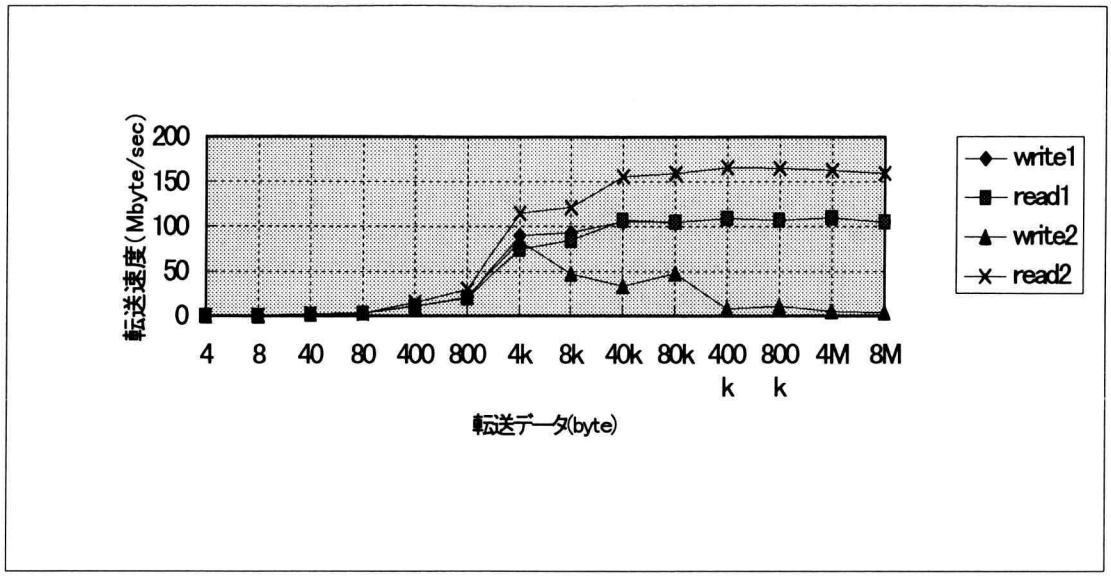

図 1 exwrite/exread 関数の転送速度

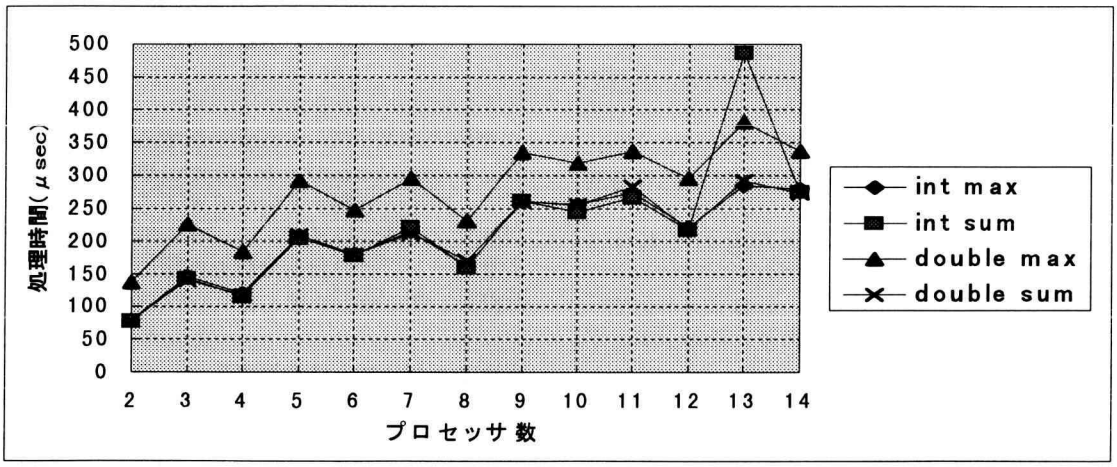

図 2 excombop 関数の処理時間

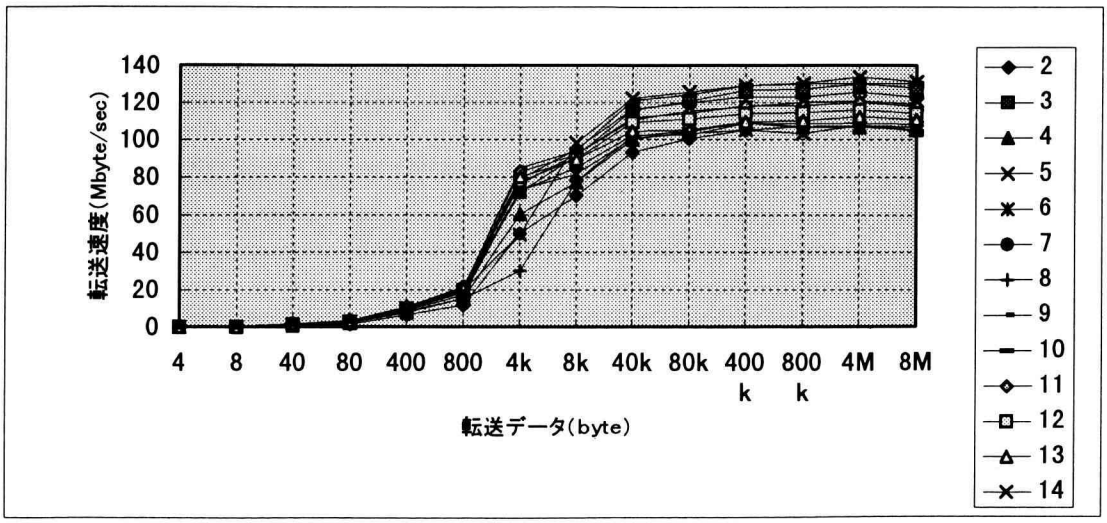

図 3 exbroadcast 関数の転送速度

結果を見ると、送受信を同時に行った場合は、一括して送るデータが 4kbyte 付近を境に約 100Mbyte/sec で、ハードウェア仕様の 1/3 強の性能を出している。（ただし、表1の通信速度 300Mbytes/sec は、双方向の通信が同時に起こった場合のピーク性能であると考えられる。）

一方、送信と受信を別々に行なった場合、受信速度は、最大で約 160Mbyte/sec ハードウェア仕様の 1/2 強の性能を出しているが、送信速度は、4kbyte 付近を境を境に極端に遅くなっている。通信バッファの容量の影響だと思われる。この結果より、ノード間通信は、なるべくタイミングを合わせて処理させたほうが、効率が良いことが分かる。

3．2．2 excombop 関数

excombop は、ノードデータの収集および結合演算の関数で、ユーザ提供関数ポインタを必要としない、簡素なインターフェースを提供する。全プロセサ上で実行され、分散された値の組のグローバル処理による和、平均、積、最小値、最大値の計算などの結合演算ができる。このうち、本論文中で必要となる、和と最大値の処理について、int 型と double 型それぞれについての測定結果を図2に示す。

測定の結果、int 型の和、最大値の計算と double 型の和の計算の処理時間は、ほぼ一致している。double 型の最大値の計算は、若干、処理時間が多くかかっているが、折れ線の形は、ほぼ同形である。したがって、処理時間に影響を及ぼしているのは、通信コストのみであることが分かる。時間自体も非常に小さなものであり、excombop 関数自体のプログラム上でのオーバーヘッドの影響は無視できる。しかし、当然、計算コストに比べて通信コストは非常に大きなものであり、その上アイドル時間を作る原因にもなり得る。

3．2．3 exbroadcast 関数

exbroadcast 関数は、ブロードキャスト通信をする関数で、 PE 数をnとしたときに$O(\log_2 n)$で、送受信することができるので、すべて、あるいは、複数のノードにデータを送信するときには、データ転送が高速にできる。転送するデータ量は、write/read 関数と同様である。

図3に示した測定結果より、転送時間は、$O(\log_2 n)$の期待どおりの性能を発揮していることが分かる。転送速度は、exwrite/exread 関数の結果とほぼ同じであるが、exbroadcast 関数のほうが若干速くなっている。ノード数による転送速度のばらつきも少ない。しかし、 exbroadcast 関数は、ブロードキャストするノード全体で同期を取るので、実際にプログラム中で使用するときには、アイドル時間を少なくなるように考慮にいれる必要がある。

以上を念頭におき、次章において、数種類の基本アルゴリズム[5-8]の超並列化とその性能特性について検討する。（さらに詳しい議論については、文献[10]にある。）

4．整列（クイックソート）

整列アルゴリズムを超並列化する上での特徴は、入力データがn個すなわち、一次元配列であるという点と、すべてのデータは、少なくとも1回は、比較処理を必要とするため、1度は同一ノードにすべてのデータが集まっている必要がある点である。整列すべきデータが、すべて主記憶上にあるとき、内部整列という。

この内部整列で、最高速の整列アルゴリズムがクイックソートである。クイックソートのアルゴリズムは、データ数をnとしたときの平均の計算量が$O(n\log_2 n)$であるが、最悪の場合の計算量は、$O(n^2)$となってしまう。

この差は、部分配列に分けるときの枢軸にどの要素を持ってくるかが、要因となって現れる。最悪の場合を避けるには、枢軸を基準に分けられる2つの配列の大きさがなるべく同じになるように分ける必要がある。具体的には、複数の要素をサンプリングし、それらの代表値を枢軸にするという方式である。実際には、3 つのデータの中央値をとるという方法が良く利用される。本論文においてもこの方法を採用する。

4．1 アルゴリズムの超並列化

（1）単純データ分割

整列アルゴリズムを超並列化するにあたってのデータ分割の方法であるが、ここでは、単純に入力時に各ノードで、均等に分割する場合を考える。この特徴は、アルゴリズム自体が単純で、データを均等に分割することによりノードごとの負荷分散も均等になるところにある。ただし、プロセッサ数をn_p、データ数をn_dとしたとき、逐次アルゴリズムに換算すると、$O(\frac{n_d}{n_p}\log_2 \frac{n_d}{n_p})$の計算を$n_p$回繰り返すアルゴリズムとなり、プロセッサ数によってアルゴリズムは異なる。

（2）非ブロードキャスト型

次に、データ分割をクイックソートの分割方法にそって、データを分割していく場合を考える。このアルゴリズムの特徴は、オーバーヘッドが、データの送信処理だけであり、前項（1）のアルゴリズムと違い、最後にマージをする必要がない。アイドル時間は、データを持っていないノードが、データを受信するまでの時間である。

欠点としては、クイックソートの性質によりデータの並びによって、ノードごとの負荷分散にばらつきが生じる可能性が高い。ただし、データ分割の初期段階においては、分けられた2つの部分配列の大きさの比によって、ノード数を割り当てることができる。

（3）ブロードキャスト型

ここでは、分割方法は前項（2）と同じで、データの通信方法として、入力時に全データを全ノードにブロードキャストする場合を考える。すなわち、この場合、各ノードが逐次アルゴリズムと同じように処理をしていき、特定の部分配列だけを整列する。

このアルゴリズムの特徴は、最初に$O(\log_2 n_p)$で

高速にブロードキャストし、それ以後ノード間通信を必要とせず、完全に並列に処理できる。したがって、アイドル時間は、極めて少なくなる。しかし、通信するデータ量は、前の２つのアルゴリズムより、多くなる。

４．２　実行結果と検討

整列するデータ数を 100、3,000、10,000、50,000、100,000 で実行で実行した。ここでは、上記３つのデータ分割のうち、もっとも性能のよかった単純分割型の結果を図４に示す。

3 つの方法とも、プロセッサ数を増やすごとに、処理時間は短くなってゆくが、並列処理効率は、だんだん落ちる。2 台の時点ですでに 50%という低い結果である。これは、通信によるアイドル時間も含めた通信コストが、計算処理に比べ非常に大きいことが分かる。そのなかで、（１）単純分割が一番良い結果を示した。他の 2 つのアルゴリズムにはない、最後に各ノードのデータを集めマージする処理があるにもかかわらず、このような結果になったことは、他の 2 つのアルゴリズムでは、各ノードへの負荷分散に、大きな差が出てしまうことを意味している。

（２）と（３）の比較では、前者のアルゴリズムのほうが、速い結果となっているので、通信するデータは少ないほうが良いことが分かる。

整列アルゴリズムを並列化する方法は、今回の３つアルゴリズムがすべてというわけではない。しかし、すでに逐次アルゴリズムでは、クイックソートなどの計算量が $O(n\log_2 n)$ のアルゴリズムが最速であることが証明されているので、整列アルゴリズム関しては、分散メモリ型並列コンピュータ向かないと思われる。工夫により、（２）非ブロードキャスト型の並列処理効率は、上げることは可能であるが、70%、80%といった高い並列処理効率を出すアルゴリズムになる可能性は低いように思われる。

５．連立一次方程式

５．１　連立一次方程式の解法について

連立一次方程式の解法を超並列化する上での特徴は、入力データが行列すなわち、二次元配列であり、そのデータを入力時に分割することができることである。また、ノード間で、通信を繰り返しながら、処理を進めていく点にある。分散メモリ型コンピュータが、威力を発揮する分野の問題である。並列処理時にノード間通信を繰り返し行う問題として、取り上げる。

ここでは、部分ピボッティング付き LU 分解を使った方法を使う。ＬＵ分解処理をするためのデータ分割の方法が性能上重要であるが、ここでは文献[9]等の結果から、一列づつ各プロセッサへ順番に割り付ける列方向巡回分割（column cyclic 分割）を採用した。

ピボッティング処理時に、他のプロセッサへデータを送る方法として、broadcast 型と write/read 型の２つを試した。broadcast 型では、k 列目の $k+1$ 行目以降のデータは、全ノードで必要となるので、この処理に入る前にブロードキャストしておく。同時に、ピボッティングのデータもブロードキャストする。

write/read 型では、broadcast 型でブロードキャストしていたデータを、それぞれ、隣のプロセッサ（ノード０ならノード１、ノード１ならノード２、…、ノードｎならノード０）だけに転送することを考える。この場合は、全プロセッサに行き渡るまでには、ブロードキャストするより時間がかかるが、各ノードにおいて、同期を取る必要がなくアイドル時間を最小に抑えることができると考えられる。

また、一回の通信回数も $\log_2 n$ から１へと減り、さらに、必要なデータを持っているノードとデータを最後に受信したノードは、それぞれ、受信、送信をする必要がなくトータルの通信回数も減らすことができる。

５．２　実行結果と検討

連立一次方程式の次元を２０、100、500、1,000、2,500 と変えた場合を検討した。

実行結果を図５、図６に示す。この結果より、ＬＵ分解は broadcast 型、write/read 型とも 500 元以上では、同じように 100%に近い並列処理効率を得いることができた。スーパーリニアとなっている箇所もいくつかある。データサイズが十分に大きければ、さらにプロセッサ数を増やしても同様の並列処理効率が得られると推測できる。

だだし、一概に、これら 2 種類のデータ分割を含めたアルゴリズムが、最も良いアルゴリズムであると結論づけることは、もちろん、できない。しかし、この結果は理想的な結果であり、もしこれ以上の並列処理効率を望むとすると、如何にしてもっとキャッシュメモリを有効に使うかという次元でアルゴリズムを考えることになる。

broadcast 型と write/read 型を比較した場合、write/read 型のほうが、高い並列処理効率が得られた。要因は、2 つ考えられる。1つ目は、exbroadcast 関数は全ノードで同期を取るので、このときに発生するアイドル時間である。処理中に、この関数がループによって繰り返し呼び出されることにより、アイドル時間が積み重なっていくからである。write/read 型では、送信側と受信側でさえ同期を取る必要がなくアイドル時間が発生するのは、受信側でデータ待ちが生じたときのみである。

２つ目は、プロセッサ数を n としたとき、broadcast 型では１回の通信で $O(\log_2 n)$ であるのに対し、write/read 型では、プロセッサ数に関係なく $O(1)$ で済む点である。write/read 型では、broadcast 型と比べて、すべてのノードにデータ行き渡るまでには時間がかかるが、処理中、ループによって繰り返し呼び出されることで、実行の最初に生じた遅延がそのまま実行の最後の遅延となるだけである。

グラフを見ると、プロセッサ数の増加に伴い差がひらいているので、プロセッサ数が 100、1,000 となったとき、大きな差となると、推測できる。

なお、LU 分解後の解の計算に関しては、文献[10]に詳しく述べているので、ここでは省略する。

６．Out-of-core 問題

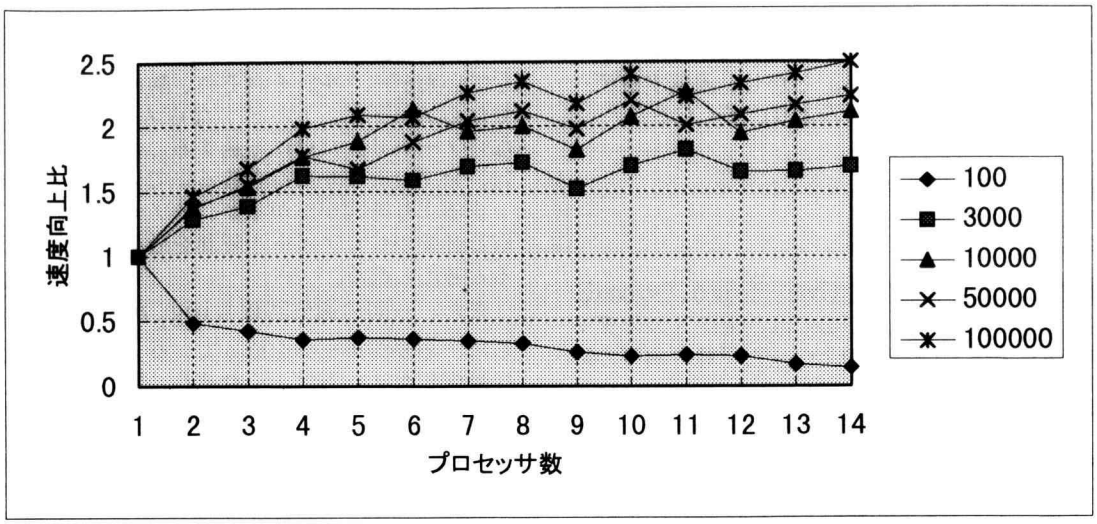

図4 単純分割クイックソートの速度向上比

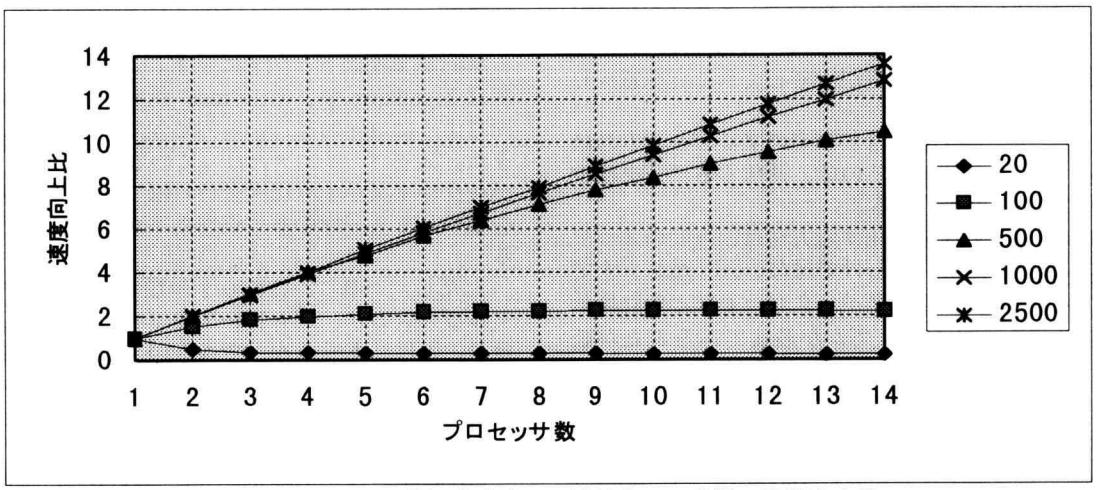

図5 broadcast 型部分ﾋﾟﾎﾞｯﾃｨﾝｸﾞ付き LU 分解の速度向上比

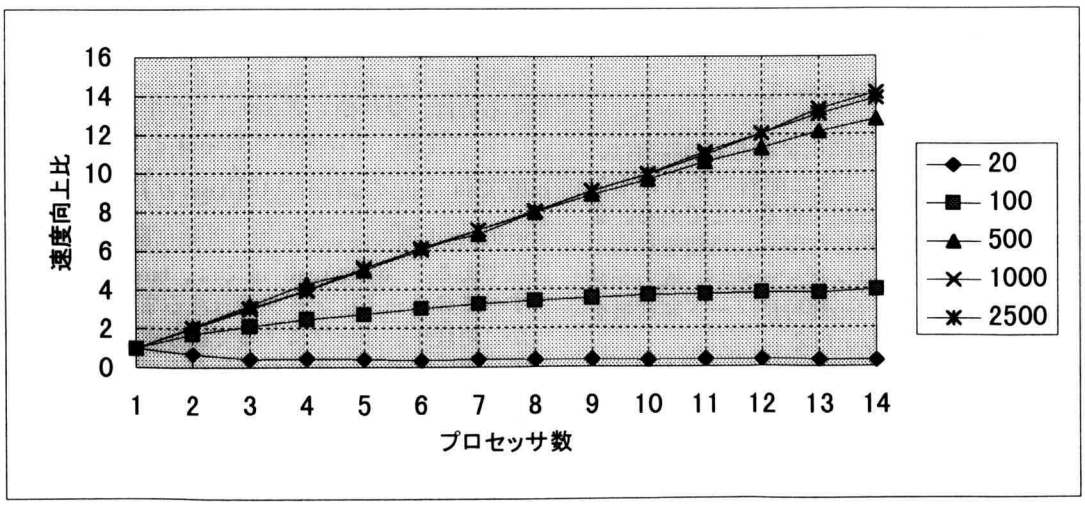

図6 write/read 型部分ﾋﾟﾎﾞｯﾃｨﾝｸﾞ付き LU 分解の速度向上比

6．1　大規模ＬＵ分解

Out-of-core 問題とは、問題の大きさが、主記憶メモリに入りきらず、外部記憶であるディスクなどのＩ／Ｏとやり取りをしながら処理を進めいていく問題である。この問題を、本論文では外部記憶に共有ハードディスクを使用し、前章で理想的な超並列化ができたＬＵ分解を例にとって検討する。このような係数行列などの入力データが入りきらない場合のＬＵ分解を大規模ＬＵ分解と呼ぶ。

ここで、ＬＵ分解をするアルゴリズムは、前章の write/read 型のアルゴリズムを用いる。ただし、ここでは、共有ハードディスクの競合の具合を考察しやすくするため、ピボッティングは行わない。共有ハードディスクとのやり取りをする部分のアルゴリズムを、以下に示す。（図７参照）

① 計算に必要となる列を共有ハードディスクまたは、ノード間通信によりバッファにいれる。
② 更新する要素をメモリに入りきる限り読み込む。（だたし、列単位）
③ 読み込んだ要素を更新する。
④ 更新した要素をディスクに書き込む。
⑤ ①出読み込んだ列を用いて更新する要素が残っていれば②へ。
⑥ 更新する行列が小さくなるので、②で読み込むことのできる列の数を更新し、①へ。

6．2　実行結果と検討

各ノードのメモリは、2000 要素入るものとして 500 次元の場合を検討した。

実行結果を図８に示す。メモリに全部入る場合において理想的な並列処理効率を得られた 500 次元という少ない場合においても、ハードディスクへのアクセスの負荷が計算処理と比べて大きく、並列化の効果が現れなかった。

図の結果は、ノードが増えるごとにシステム全体のメモリも増やして実行させた場合である。したがって、ノードの増加ともにメモリに入りきらない部分が少なくなっている。その上、他の章で示したグラフと違い、速度向上比や並列処理効率ではなく実行時間であるにもかかわらず、グラフ上の変化はほとんど見られない。

これはハードディスクというメインメモリへのアクセスと比べて桁違いに遅い外部記憶を共有して使用するため、競合により大きなアイドル時間が発生するのが主な原因と判断される。アルゴリズムの改良により外部記憶へのアクセスを大幅に減らすことも可能であるが、実用的な並列処理効率を得ることは、困難であると思われる。

実用的な並列処理効率を得るためには、半導体高速補助記憶などの高速な記憶装置を使用することが有効であると思われる。また、本論文では確めることのできなかった分散型の外部記憶を用いた場合の検討が残されている。

7．むすび

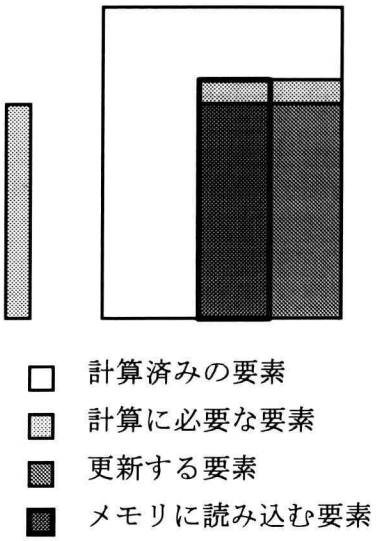

図７ Out-of-Core LU 分解でのノード毎の行列

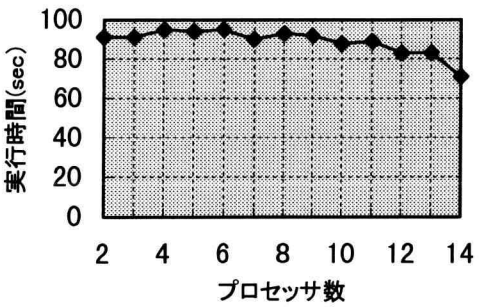

図８ Out-of-Core LU 分解の計算時間

基本的な問題について、超並列アルゴリズムを検討し、SR2201 に実装することにより、その性能を評価した。

クイックソートの並列化では、データ転送の負荷が大きく、また負荷分散も均等にするのが難しく、このような問題では分散メモリ型並列コンピュータに実装することが困難であることを示した。ＬＵ分解のアルゴリズムに関しては、データ分割を列方向巡回分割とし、ノード間通信を write/read 型とすることで、部分的にスーパーリニアとなる高い並列処理効率が得られた。最後に、メモリに比べて非常にアクセスの遅いハードディスクを共有して out-of-core 問題を解くことが、性能上困難であることを示した。

本論文での検討結果から、並列処理性能を左右する要因をよく把握すれば、高性能な超並列アルゴリズムを構築することは、色々な問題で十分可能であるという実感を得た。

今後の課題には、並列クイックソートの改善、分散ハードディスクを用いた大規模ＬＵ分解、および反復系解法などが含まれる。

参考文献

[1] 山本富士男：日立超並列コンピュータ利用の手引き，第0.1版～第2.7版，1996

[2] HITACHI Parallelware ユーザーズガイド－C－，1996

[3] HITACHI Parallelware リファレンス－C－，1996

[4] 江丸裕教、高井昌彰：超並列計算機 SR2001 による最短経路問題の解法について，ハイパフォーマンスコンピューティング 60-2(1996.3.4)

[5] 高田伸彦：コンピュータサイエンス，アイテック，pp.95-97, 1996

[6] 戸川隼人：数値計算法，コロナ社， pp.24-25，1992

[7] 近藤嘉雪：Cプログラマのためのアルゴリズムとデータ構造，ソフトバンク pp.177-237，1995

[8] 津田孝夫：岩波講座ソフトウェア科学 9 数値処理プログラミング，岩波書店

[9] 山本富士男：超並列機向けデータ分割の自動評価方式、並列処理シンポジューム JSPP'92 論文集、pp.407-414， 1992

[10] 川上芳尚：超並列コンピュータ SR2201 における基本アルゴリズムの検討、平成8年度神奈川工科大学情報工学科（山本富士男研究室）卒業論文、1997