社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2003－HPC－96　(4)
2003／10／16

ERXPP — 数値ライブラリにより並列計算性能を簡易かつ適応的に引き出す方式の提案

須　田　礼　仁†

高速ネットワークの低価格化と爆発的な普及に伴い並列計算環境の構成は自由度を増しており、そのような計算資源を効率的に使うことは古典的な並列化手法では難しくなっている。本稿では ERXPP (Easy and Robust Extension of Parallel Performance) すなわち様々な並列環境に適応できるよう数値ライブラリを構築することで、アプリケーションプログラマへの負担を最小限に抑えつつネットワーク計算環境の計算性能を効率的に活かすことを提案する。また、ERXPP の機能の一部を実装して行った予備評価について報告する。高速ネットワークのおかげで ERXPP が有効に働くことが実証された一方で、多くの課題が山積していることも明らかとなった。

ERXPP: Easy and Robust Extension of Parallel Performance through Use of Numerical Library Routines

Reiji SUDA†

The architectural complexity of the parallel computing environments increases rapidly. The conventional strategies of parallel programming fail to provide enough efficiency on contemporary heterogeneous, dynamic and unreliable parallel systems. This paper proposes Easy and Robust Extension of Parallel Performance (ERXPP) through numerical library routines to exploit the performance of networked computing environments without much burdening the application programmers.

Experimental results of a preliminary implementation of a few ERXPP features on a fast spherical harmonic transform library are also reported. The results show that high-bandwidth networks are essential for the effectiveness of ERXPP, and thus it is the right time to develop routines with ERXPP features. The experiments also reveal several problems to be solved for fuller implementation of ERXPP features.

1. はじめに

計算機ネットワークの急速な進歩と低価格化は並列計算の世界に大きな変化を与えている。クラスタが普及してその利用が進む一方で、クラスタとして計画・構築されていない高速ネットワーク計算環境はもっと急速に広まっている。しかしそのような並列計算環境はヘテロで動的で信頼性が低く、その潜在的な計算能力を最大限に引き出すことはきわめて困難である。例えばヘテロというのは単に仕事量を CPU 性能に比例して割り当てれば解決するものではなく、通信量・通信性能・ネットワークトポロジーについても考慮しなければならない。他の負荷の影響により計算時間や通信時間がばらつく時にどう対処すればよいかという問題についても、我々の手元には十分な知見があるとは言えない。さらには耐故障性をアプリケーションレベルで実現するためにはそれなりの道具と工夫が必要であり、一般的に容易に実現できるとは言いがたい。

それでは、均一な構成のクラスタが安定動作している場合だけしか並列計算は現実的ではないとしてそれ以外の場合を見捨てるべきであろうか。私はそうではないと願いたい。しかしそのような難しい環境下において現実的な労力で計算性能を引き出すためには、これまでの並列化手法の延長線上ではない手法が必要である。可能性のある方法はいろいろ考えられるが、本稿では各種の並列環境に高度に適応した数値ライブラリ（または類するコアルーチン）を使用することにより、アプリケーションが容易に並列計算性能を活用すること — ERXPP (Easy and Robust Extension of Parallel Performance) — を提案する。呼び出し側（アプリケーション）が比較的簡単な実装であっても、数値ルーチンがアプリケーションの主要な計算を効率よく並列処理することにより、ネットワーク計算環境の計算性能を（最適にではなくとも）それなりに引き出すことができると期待するのである。

ERXPP は次のような状況で有効性を発揮できることを期待している。

- 呼び出し側は逐次のままか、SMP 上で並列化されているだけであるが、ライブラリルーチンはネットワーク上の計算資源も利用する
- 呼び出し側はホモなクラスタ上で並列化されている

† 東京大学 情報理工学系研究科/科学技術振興機構 CREST
G.S. of Information Science and Technology, the University of Tokyo/CREST, Japan Science and Technology Agency

が、ライブラリルーチンはヘテロなプロセッサを含むより大きな環境を利用する
- 呼び出し側は静的なデータ分散を用いているが、ライブラリルーチンは動的負荷分散を行い性能を高める
- 呼び出し側は安定したシステム上で動作しているが、ライブラリルーチンは信頼性の低い計算資源をも利用して性能の向上を図る

これまでライブラリルーチンは高い性能を提供すべく開発されてきたが、本稿ではこれまでよりも高い適応性と耐故障性とをライブラリルーチンに付加することを提案するものだと言うことができる。

2. ERXPPの概念定義

2.1 ERXPPの目標

ERXPP の概念を明確に定義するには、その目標を示すのがよいように思われる。それを一文にまとめると「アプリケーションプログラムは相対的に単純な実装であっても、ヘテロで動的で信頼性の低い並列計算環境に適応して高い性能を実現するライブラリを構築すること」である。

ここで重要なのは次の5点である。

単体での高性能: それぞれのプロセッサ上で高い性能を実現することは基本である。ERXPPのメインテーマではなくとも、これが無視されることはありえない。

ヘテロへの適応性: プロセッサやネットワークがヘテロ構成であっても、できるだけ高い効率でこれらを利用することが望まれる。ハードウェアとソフトウェアの両方に関する多様な構成に適応しなければならない。

ダイナミクスへの適応性: 並列計算環境は様々な意味で動的である。プロセッサが追加されたり削除されたりすることもあるし、他のユーザーがリソースを使用したために実効性能が変化する場合もある。また、アプリケーションのデータが動的な場合もある。いずれの場合に対してもライブラリは適応して、データ分散やスケジューリングを最適化しなければならない。

故障への適応性: プロセッサ数が増え、ネットワークが広範囲に及ぶにつれて、故障の可能性も高くなる。完全に故障しているのではなくとも、一時的にストールすることはよくある。このような場合にアプリケーション全体を耐故障的に構築するのは最も望ましいかもしれないが、ERXPP の基本スタンスからは異なる手法が導き出される。すなわち、アプリケーション側は信頼性の高いシステムで実行される（従って簡易な実装が可能である）が、ライブラリルーチンは信頼性の低いリソースを追加的に活用できるよう耐故障性を実装するというものである。これは信頼性のヘテロ性と考えることができる。但し、CPU やメモリ、通信チャネルのエラー、セキュリティはERXPPの範囲では考慮せず、別に対処してあるものとする。

利用の簡便性: 上記のような効果をアプリケーションプログラマへの負担を最小限に抑えて実現することが求められる。呼び出し側のプログラムが並列環境に最適に適応していない状況でも使えなければならない。呼び出し時に仮定されるデータ分散は柔軟でなければならない。また、システムのパラメタや状態に関する情報を（ライブラリルーチン自体が、あるいは補助的なソフトウェアが）自動的に収集し、呼び出し側のプログラムが様々なパラメタ設定をしなくてもよいように努力しなければならない。

2.2 関 連 研 究

この節では、上記の ERXPP の各種の機能に関係する先行研究について考察する。

ヘテロ性への対応: ヘテロ性対応に関する研究は意外と少なく[4)~7)]、既存の結果は極めて不十分である。比較的容易なプロセッサのヘテロ性でも、プロセッサ性能に比例するデータを割り当てればよいというものではなく、それに伴って生じる通信の非一様性とのトレードオフが必要である。ネットワークがヘテロな場合にはトポロジーも考慮する必要がある。いずれにしても、古典的な「並列アルゴリズム」から「スケジューリング」への転換が必要となる。

動的負荷分散と耐故障性: これらは長い歴史があるので、参考になる先行研究もかなりある[8)~10)]。しかし、信頼性のヘテロ性や、動的システム構成におけるリカバリなど、課題も残っている。一方で、実行時間のぶれ（擾乱）に対する適切な対処手法は十分に開発されていないようである。

自動適応やグリッド対応の数値ライブラリ: 計算グリッドは ERXPP と極めて類似した仮定を持っている。また、オートチューニングあるいはセルフアダプティブと呼ばれるライブラリなどの開発も活発である。ERXPPが目標としている機能すべてを含むライブラリはまだないように思うが、極めて近接したテーマを持っている研究がいくつかある[11)~13)]。

3. 予備的な実験と評価

本節では、地球上の流れ場のシミュレータにおける球面調和関数変換を例として、ERXPPの機能を一部実装して行った実験の結果を報告する。実装内容は上述のERXPPの機能からするとごく限られたものであるが、得られた結果は示唆的である。

3.1 用いたソフトウェア

数値ライブラリとしては著者ら[1)]が開発してきている高速球面調和関数変換 FLTSS[2)] を用いた。球面調和関数変換はフーリエ変換とルジャンドル変換とからなり、並列処理に際しては、図 1 に示したように、両変換の間に全体全通信が入る（2次元フーリエ変換に似ている）。なお本稿は FLTSS の並列処理の最初の報告となる。

アプリケーションは京都大学の余田成男教授らによる地球大気の乱流シミュレーションプログラム[3)]である。これは2次元球面上の非発散渦度方程式を解くものであるが、モデルがシンプルなので球面調和関数変換以外には通信の必要がない。特に図 1 のように物理空間を行（東西）方向に、スペクトル空間を列（南北）方向に、それぞれ分散

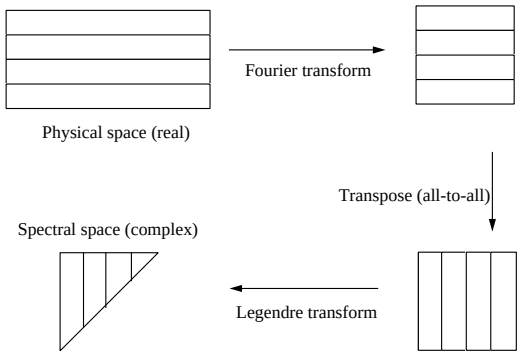

図1 並列球面調和関数変換におけるデータ分散と通信
Fig. 1 Data distribution and communication in parallel spherical harmonic transform

させる場合が最も簡単で、球面調和関数変換にもともと含まれる全体全通信以外の通信は必要なくなるので、今回の実装はいずれもこのような分割を用いている。この状況はブロック分割でもサイクリック分割でも同様であるが、ブロック分割を用いるとスペクトル空間でデータ分散に不均衡が生じる。

3.2 実装したERXPP機能

今回 ERXPP の機能として実装したのは変換計算の再分散の機構である。呼び出し側（アプリケーション）がすべてのプロセッサを使い切っていない場合や、データ分散に不均衡がある場合、さらにはプロセッサがヘテロな場合など、アプリケーションが用いているデータ分散が最適でないような状況において、データを転送して計算負荷を移動して負荷の均衡化を行うことにより所要時間の短縮を図るものである。また、スケジューリングを実行時に行うことにより、プロセッサ構成の動的変動に対応することが可能となる。

処理の再分散においては、それにかかる通信コストもあわせて考慮して最適化する必要がある。図 1 に示すように、球面調和関数変換はフーリエ変換、全体全通信、ルジャンドル変換の3つの部分からなっているが、今回の実装ではフーリエ変換とルジャンドル変換のそれぞれについて、データ移送のための通信時間と計算時間をあわせた所要時間の最適化を行うスケジューラを構築した。一方で全体全通信は古典的なサイクリックアルゴリズムを用いて単純に実装し、その所要時間はスケジューラでは一切考慮しないこととした。

スケジューラのアルゴリズムの詳細は紙面の都合で省略するが、divisible load のアルゴリズム[6),7)]を拡張したもので、シングルラウンドである。さらに、変換の実行中に計算所要時間を計測して実効性能を測定し、その結果を次の呼び出しに利用することにより、プロセッサ性能の動的な変化に対応することができるようにした。そのためスケジューラは手続きの呼出しごとに実行することにした。しかし通信の実効性能は実行中に測定することが難しいので、事前に測定しておいた値で固定することにした。

3.3 実験結果

以下の実験で用いた計算機は、CPU が Xeon 2.4 GHz、主記憶 512 MB、OS が RedHat Linux 7.3 のノードを G-bit イーサで接続したもので、コンパイラは Intel fortran compiler を用いた LAM MPI によっている。

ここであらかじめ表（表1から表7）の記述について説明をしておく。ERXPP の欄の off は処理の再分散を含まない単純な並列化の結果、on は処理の再分散を行った場合の結果であることを示す。processors の欄は、app の列が呼び出し側、lib の列がライブラリ側の使用ノード数である。forward または inverse transform のそれぞれについて total, FT, LT の欄があるが、これらは球面調和関数の正変換および逆変換について、変換全体、フーリエ変換、ルジャンドル変換の所要時間（1回あたりの平均）を示す。本実験では所要時間の内訳を正確に評価するために、（本来は不必要な）同期をフーリエ/ルジャンドル変換の前後に挿入している。これらの部分について個別に最適化してスケジューリングしているため、このような評価でより適切に評価できるものと思う。フーリエ/ルジャンドル変換の所要時間には処理の再分散のための通信時間が含まれているので、全体全通信の所要時間は変換全体の所要時間から両変換の所要時間を差し引くことによりかなり正確に見積もることができる。simulation の欄はシミュレーション全体の所要時間を示す。シミュレーションは実験時間の節約のために3回しか反復しておらず、その中で逆変換が 24 回、正変換が 12 回呼び出されている。逆変換は正変換のおよそ倍の時間がかかっているが、これは逆変換が正変換の2倍のデータ量を変換するためである。呼び出し回数も含めると、逆変換には正変換の4倍の比重があることになる。

実験1: 表1は呼び出し側が1ノードのみで実行されている場合（マスター・スレーブ構成）の性能である。シミュレーション時間に着目すると、並列化効率という指標では理想からは遠く離れているが、プロセッサ数を増すにつれて着実に短くはなっている。ルジャンドル変換はそれなりに高速化されているが、フーリエ変換ではほとんど速くならない。この違いは両変換の処理の再分散に要する通信量が同程度なのに対して、計算量はフーリエ変換のほうがずっと少ないことによっている。全体全通信は球面調和関数変換の所要時間のかなりの部分を占めていることもわかる。

実験2: 表2は 100 Mbps のネットワーク、クロック 2 GHz のマシンで表1と同じ実験を行った結果である。ここではシミュレーション時間はかえって余計にかかるようになっている。変換時間の内訳を比べると、性能低下の主要な原因は全体全通信の所要時間にあり、スケジューラが全体全通信を考慮していないことが問題であることがわかる。ルジャンドル変換は多少高速化されているが、フーリエ変換はかなりの場合スケジューラにより処理を再分散すべきでないという判断を下されている。全体全通信も含め

表1 マスタースレーブ構成での性能
Table 1 Performance results of master-slave configuration

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	1	1	0.160	0.060	0.100	0.320	0.118	0.202	10.457
on	1	2	0.140	0.054	0.055	0.225	0.094	0.107	8.449
on	1	3	0.128	0.054	0.039	0.188	0.086	0.075	7.251
on	1	4	0.113	0.052	0.031	0.171	0.082	0.058	6.600

表2 100 Mbps ネットワークでの性能
Table 2 Performance results with 100 Mbps network

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	1	1	0.182	0.061	0.120	0.384	0.137	0.247	12.306
on	1	2	0.196	0.062	0.079	0.434	0.136	0.160	14.710
on	1	3	0.204	0.062	0.067	0.456	0.131	0.128	14.944
on	1	4	0.206	0.062	0.059	0.480	0.127	0.111	15.583

表3 派遣社員構成での性能
Table 3 Performance results for temp-staff configurations

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	2	2	0.101	0.032	0.050	0.186	0.061	0.102	6.134
on	2	2	0.105	0.032	0.051	0.188	0.061	0.103	6.255
on	2	3	0.092	0.035	0.036	0.159	0.055	0.073	5.703
on	2	4	0.081	0.029	0.028	0.134	0.048	0.054	4.869
off	3	3	0.072	0.021	0.034	0.136	0.041	0.068	4.461
on	3	3	0.076	0.022	0.034	0.141	0.040	0.069	4.635
on	3	4	0.070	0.025	0.027	0.126	0.042	0.056	4.370

表4 ルジャンドル変換で負荷不均衡がある場合の性能
Table 4 Performance results with load imbalance at Legendre transforms

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	2	2	0.130	0.033	0.080	0.258	0.061	0.160	8.225
on	2	2	0.115	0.033	0.053	0.196	0.061	0.105	6.816
on	2	3	0.098	0.034	0.038	0.160	0.055	0.072	5.855
on	2	4	0.086	0.030	0.029	0.138	0.048	0.055	5.028
off	3	3	0.100	0.022	0.062	0.191	0.040	0.125	6.160
on	3	3	0.080	0.022	0.036	0.144	0.040	0.070	4.956
on	3	4	0.075	0.025	0.029	0.132	0.042	0.054	4.688
off	4	4	0.080	0.016	0.050	0.159	0.031	0.101	5.064
on	4	4	0.065	0.016	0.027	0.121	0.030	0.053	4.089

て最適化されれば、ルジャンドル変換部分の効果でわずかに高速化が達成される可能性はある。

この実験からわかる重要な知見は、集団通信も含めてスケジューリングを行わないと性能を悪化させてしまう（すなわち ERXPP の理念に反する）ということである。逆に先の実験はネットワークが十分高性能であれば多少いい加減なことをしても性能が向上できるということも意味している。このことは、今回実装した処理の再分散のような機能は高速ネットワークがあって初めて意味があるということをも意味している。

実験3: 表3は呼び出し側が複数プロセッサにサイクリック分散されていて、ライブラリルーチンはさらにプロセッサ数を増やして実行する場合（派遣社員構成）の結果である。追加のプロセッサがない場合には、ERXPP が on になると数ミリ秒長くなっているが、これがスケジューリングのオーバーヘッドである。それ以外の場合にはマスタースレーブ構成とほぼ同様の傾向となった。

実験4: 表4は、表3と同様の構成であるが、呼び出し側の分散をブロック分散にし、ルジャンドル変換では負荷の不均衡が生じるようにした場合の結果である。処理の再分散により負荷の不均衡は改善され、追加のプロセッサもそれなりに有効利用されている。

実験5: 以下の実験ではノードのうち 1 台に負荷をかけている。負荷としたプロセスは 1024×1024 の行列の積を繰り返し計算するもので、ノードが 2 CPU あるため 3 プロセス立ち上げることにより、実効ノード性能が無負荷

表5 マスターが一台で負荷がかかっている場合の性能
Table 5 Performance results with a single loaded master

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	1	1	0.374	0.139	0.235	0.809	0.285	0.500	25.348
on	1	2	0.377	0.102	0.103	0.542	0.229	0.197	20.103
on	1	3	0.295	0.083	0.046	0.634	0.197	0.104	21.523
on	1	4	0.260	0.085	0.046	0.452	0.105	0.084	17.045

の場合の半分になることを期待している。

表5はマスタースレーブ構成でマスターに負荷がかかっている場合の結果である。処理の分散の効果は思ったほど高くない。しかしフーリエ/ルジャンドル変換の性能は向上しているところから、効果の低さの原因は全体全通信における性能低下であることがわかる。この状況を改善するためには、全体全通信をスケジューリングに追加するのみならず、通信性能のヘテロ性を考慮して全体全通信のアルゴリズムを最適化する必要がある。

実験6: 表6はマスタースレーブ構成でスレーブの一台に負荷がかかっている場合の結果である。結果は悲惨であるが、実は表5の場合よりも所要時間は短縮している。1台の場合よりも性能が低下している原因はやはり全体全通信にある。集団通信のような細粒度の処理は他のプロセスの存在によるスケジューリングの擾乱の影響を激しく受けることによるものと思われる。従って、細粒度の並列プログラムをマルチタスク環境で実行する場合には、所要時間のばらつきを吸収できるような工夫が必要であるということが知見として得られる。一方で逆ルジャンドル変換はそこそこの性能向上を達成しており、粒度が粗ければ他のプロセスから受ける擾乱の影響が相対的に小さくなることを示している。

実験7: 表7は、呼び出し側がサイクリック分割で並列化されているが、そのうち一つに負荷がかかっている場合の結果である。フーリエ変換が何もしない場合よりも遅くなっていることが多いが、これも粒度が十分粗くないために他のプロセスによるスケジューリングの影響を受けたものと思われる。やはりルジャンドル変換は若干性能が改善しており、全体としては処理の再分散によりわずかに性能が改善する。

今回は負荷が時間的に変化するような実験は行わなかった。しかし、スケジューラはルーチンの呼び出しごとに起動されており、参照する性能は何の平均化もなく直前の実行の値を用いるため、数回の呼び出しの後に新しい状況に対応したスケジューリングになることが予想される。

4. まとめと考察

本稿では、高度に適応的な並列化を実現した数値ライブラリを使用することにより、アプリケーションプログラマへの負担を最小限に抑えつつ、複雑化する並列計算環境の計算性能を引き出す ERXPP の概念を提案した。また、球面調和関数変換を用いた流体シミュレーションを例題として、ERXPP の機能の一部を実装し予備的な評価をおこなった。

今回の実験は ERXPP で必要と考えている機能のごくわずかしか取り上げなかったが、それでもさまざまな知見が得られた。通信量の多い「処理の再分散」という方式でも高速化を達成することができることが明らかとなったが、これは Gbit ネットワークの性能のおかげであり、このような並列化手法はようやく現実的になりつつあるところだといえよう。さらに 10Gbit のネットワークが普及すればもっと多様な可能性が広がるものと期待される。しかし、改善すべき点や残された課題も多い。

まずはヘテロ性に対応した集団通信の最適化である。集団通信の最適化は均一な環境ですら自明でない問題[14]であるので、SMP やヘテロなネットワークトポロジーの場合にはなおさら難しい課題である。データ量がヘテロな場合には、計算の所要時間もばらばらなのが当然であるから、ばらばらな時刻にスタートしてばらばらな時刻に終了する集団通信を考えなければならないが、そのような集団通信を最適化するという研究は見当たらないようである。

スケジューリングについては今回はシングルラウンドとしたが、LAN であれば通信遅延の影響は比較的小さくマルチラウンドが最適になる可能性もある。しかし最も簡単な問題設定である divisible load でも、今回の設定のようにマスターが複数となった場合の研究はほとんどなく、研究開発が必要である。

また、通信性能の実行時モニタリングも実効性能の動的変化に対応するために必要である。通信遅延の隠蔽などのためユーザーレベルプログラムからは実際の通信所要時間はほとんど見えないとはいえ、ユーザーに見える情報だけからでも通信性能の低下・向上を検出することは不可能ではないはずである。しかし既存の研究は見当たらない。さらに、所要時間のモデルは最適化の成否を決する重要事項であるが、これの実行時のチューニングも重要な検討課題である。

さらに、各種の擾乱によるスケジューリングへの影響のコントロールも課題である。擾乱に対するスケジューリングの工夫についての研究も若干あるものの、極めて不十分である。OS のスケジューリングに関して各種の研究があるが、ERXPP の主旨からするとスケジューラに依存せずにある程度の性能を出したい。ここでも集団通信が重要課題であることはすでに指摘したとおりである。

今回は耐故障性については考えなかった。これは使用

表6 スレーブに負荷がかかっている場合の性能
Table 6 Performance results with a loaded slave

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	1	1	0.160	0.060	0.100	0.320	0.118	0.202	10.457
on	1	2	0.294	0.061	0.119	0.429	0.166	0.181	16.039
on	1	3	0.224	0.052	0.079	0.353	0.101	0.102	12.882
on	1	4	0.147	0.051	0.037	0.280	0.090	0.061	10.323

表7 複数マスターのうち一つに負荷がかかっている場合の性能
Table 7 Performance results for multiple masters with loaded one

	processors		forward transform			inverse transform			simulation
ERXPP	app	lib	total	FT	LT	total	FT	LT	total
off	2	2	0.249	0.039	0.067	0.473	0.098	0.243	17.820
on	2	2	0.347	0.062	0.060	0.416	0.115	0.196	16.855
off	3	3	0.163	0.028	0.048	0.439	0.052	0.175	15.271
on	3	3	0.212	0.036	0.039	0.405	0.055	0.118	13.993
off	4	4	0.166	0.020	0.034	0.300	0.039	0.066	10.867
on	4	4	0.151	0.018	0.028	0.288	0.042	0.061	10.051

している MPI が故障に対応できないということのみならず、そもそもユーザーレベルで故障に対応するのはかなり大変だということによっている。耐故障性を考えるにはやはりミドルウェアレベルのサポートがほしい。このほかリソースの増減や負荷の変動に関する情報などもミドルウェアなどで提供されることは望ましい。

今回は数値ライブラリとして球面調和関数変換を取り上げたが、並列性の単純さや適当な計算量など、最初に取り上げる例としては比較的よかったようである。FFT だけであれば Gbit でもまだスループットが不足であったであろうし、線形ソルバーなどはもっとアルゴリズムが複雑なので十分な解析ができなかったであろう。いずれにせよ、今回のような簡単な問題であってもこれだけ様々な課題が残ったのであるから、今後の展開のためには非常な努力が必要であることは間違いがない。

謝　　辞

この研究は科学技術振興機構の CREST（大規模シミュレーション向け基盤ソフトウェアの開発）、文部科学省の 21 世紀 COE プログラム（情報科学技術戦略コア）および科学研究費（非構造多重格子を用いた離散化手法とその効率的な並列実装技術）の援助を受けています。

計算機を使用させてくださった東京大学の小柳義夫教授と小柳研究室の諸氏に感謝いたします。

参 考 文 献

1) R. Suda and M. Takami, *A Fast Spherical Harmonics Transform Algorithm*, Math.Comp., Vol.71, No. 238, Apr. 2002, pp. 703–715.
2) FLTSS ホームページ `http://www.na.cse.nagoya-u.ac.jp/~reiji/fltss/`.
3) 余田成男、山田道夫、石岡圭一、「スペクトル法による球面上の流体方程式の数値解法」、京都大学大型計算機センター広報、Vol. 23, No. 5, Oct. 1990, pp. 283–290.
4) O. Beaumont, A. Legrand, F. Rastello and Y. Robert, *Static LU decomposition on heterogeneous platforms*, Int. J. High Perf. Comp. Appl., Vol. 15, No. 3, 2001, pp. 310–323.
5) O.Beaumont et al., *A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)*, IEEE Trans. Comp., Vol. 50, No. 10, 2001, pp. 1052–1070.
6) C. Banino, O. Beaumont, A. Legrand and Y. Robert, *Scheduling strategies for master-slave tasking on heterogeneous processor grids*, LIP Tech. Rep. 2002-12.
7) Y. Yang and H. Casanova, *RUMR: Robust Scheduling for Divisible Workloads*, Proc. HPDC-12, Jun. 2003.
8) G. Bosilca et al., *MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes*, Proc. SC2002, Nov. 2002.
9) Y.Kim, J.S.Plank and J.Dongarra, *Fault Tolerant Matrix Operations for Networks of Workstations Using Multiple Checkpointing*, Proc.HPC Asia '97, Apr. 1997, pp. 460–465.
10) G. Fagg and J. Dongarra, *FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world*, Euro PVM/MPI User's Group Meeting 2000, pp. 345–353.
11) GrADS homepage: `http://hipersoft.cs.rice.edu/grads/`
12) GRAIL homepage: `http://grail.sdsc.edu`
13) SANS homepage: `http://icl.cs.utk.edu/iclprojects/pages/sans.html`
14) S. S. Vadhiyar, G. E. Fagg and J. Dongarra, *Automatically Tuned Collective Communications*, Proc. SC2000.