計算機アーキテクチャ　128－8
ハイパフォーマンス
コンピューティング　70－8
（1998．3．5）

クラスタ型並列計算機における Implicit Co-scheduling の性能評価

福地　健太郎[†]　松岡　聡[†]
堀　敦史[††]　石川　裕[††]

Implicit co-scheduling は Berkeley NOW プロジェクトで提案された、大域スケジューラーを持たず、オーバーヘッドが少い・実装が容易である等の利点を持つ並列ジョブスケジューリング技法である。これまでの性能評価では実行時間の対ギャングスケジューリング比にして 0.6 ～ 1.6 程度の性能が出るとされているが、実用的なアプリケーションでの性能評価はなされていない。本研究では、大規模高性能クラスター上で、NAS 並列ベンチマークを用いる事で、implicit co-scheduling の実践的な性能を測定した。その結果、FT,CG において実行時間の対ギャングスケジューリング比にして最大 2.3 倍という結果を得ており、Berkeley の評価が再現しなかった。これは、ネットワークの混雑等が原因と予測され、現在追試中である。

Evaluation of Implicit Co-scheduling on Clustered Parallel Computer

Kentarou Fukuchi,[†] Satoshi Matsuoka,[†] Atsushi Hori[††]
and Yutaka Ishikawa[††]

Implicit co-scheduling is a parallel job scheduling methodology proposed by the UC Berkeley NOW project, and embodies favorable characteristics such as lack of global schedulers, low overhead, and easy implementation. Previous literatures have claimed that overhead versus traditional gang schedulers was about a factor or 0.6 to 1.6; however, evaluations were not performed using real-life workloads. We have implemented an implicit co-scheduler on a large-scale, high-performance cluster, and used NAS parallel benchmarks to measure effective performance. There, we found that for FT and CG, the overhead versus gang scheduling can be as high as factor of 2.3, negating the Berkeley results. We conjecture that this is due to excessive network traffic, but are still in the process of performing additional experiments.

1. はじめに

並列計算機上で時分割スケジューリングしている環境では、プロセス間での通信において、目的プロセスがスケジュールされていない場合に無駄な待ちが生じる。時分割スケジューリングでは、このプロセス同期に伴うオーバーヘッドの削減が課題となる。

ギャングスケジューリング[1] は、並列計算機上の全プロセッサで、互いに通信しあうプロセス群を一斉に切り替えるスケジューリング手法で、前述したオーバーヘッドはほとんどない。しかし、プロセスの切り替えをネットワーク経由で指示するため、プロセス切り替えのオーバーヘッドが大きい。

Implicit co-scheduling[2)~4)] はギャングスケジューリングのような、大域的なプロセスの同期スケジューリングを行わず、通信部分に two-phase spin-block を導入する事で、通信部分の同期を狙うスケジューリング手法である。大域スケジューラーを持たない事から、プロセス同期の為のオーバーヘッドが少い、実装が容易である等の利点が挙げられているが、確実な co-scheduling が保証されておらず、実用的な面での性能が実証されていない。

これまでの性能評価の報告では、シミュレーションや簡単なベンチマークによるものが報告されている[2),4)]。これによると、実行時間の比較では、ギャングスケジューリングに対して、0.6 ～ 1.6 倍程度の性能を示している。しかし、評価環境が SUN Ultra1 16 台構成のワークステーションクラスタという、比較的小規模な構成だったこともあり、実践的な評価とは言い難い。

そこで、本研究では、新情報処理開発機構 (RWCP) の PC クラスター II という 64 台構成の高性能 PC クラスタ上で（詳しい仕様は表 1 を参照）、NAS 並列ベンチマークにより性能を測定した。NAS 並列ベンチマークは科学技術計算に多く使われるコンポーネント

† 東京工業大学
Tokyo Institute of Technology
{fukuchi,matsu}@is.titech.ac.jp
†† 新情報処理開発機構
Real World Computing Partnership
{hori,ishikawa}@trc.rwcp.or.jp

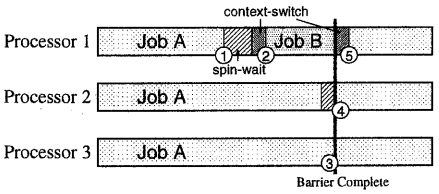

図1 Implicit co-scheduling の模式図

からなるベンチマークプログラム集であり、実践的なアプリケーションに近い挙動を示すものと期待される。Implicit co-scheduling の実装は、MPI 通信ライブラリに単純な two-phase *fixed* spin-block を導入する事で行った。

測定の結果、FT,CG ベンチマークにおいては、ギャングスケジューリングに対して最大 2.3 倍に速度が低下する事が分かった。今回のような、通信の多いアプリケーションにおいて、受信側で two-phase spin-block するような実装では Berkeley の評価は成立しない事が判明した。

2. Implicit co-scheduling

Implicit co-scheduling[2)~4)] は、カリフォルニア大バークレイ分校の NOW プロジェクトにより提案された並列ジョブスケジューリング手法で、明示的な同期スケジューリングをせずにプロセスを同期させる手法である。並列環境下での時分割スケジューリングの問題点は、プロセッサ間での通信時に目的プロセスがスケジュールされていない時の無駄な待ち時間にあった。そこで、implicit co-scheduling では通信部分での待ちに **two-phase spin-block** を用いる。この手法では、プロセスは受信待ち状態に入るとまず一定時間 spin-wait する。spin-wait 中に受信待ちが解消せずに一定時間経過するとブロックし、自主的に他のプロセスにプロセッサを空け渡す。spin-wait 中に受信待ちが解消した場合は、通信処理を開始する。spin-wait 時間を固定したものを特に two-phase *fixed* spin-block と呼び、実行時に変化するものを *adaptive* two-phase spin-block と呼ぶ[2)]。

図1は、two-phase spin-block のバリア同期時の挙動の模式図である。プロセッサ1上のジョブAが最初にバリアに到達し、まず spin-wait を開始する (1)。spin-wait 期間中にバリア同期が成立しなかったためブロックし、ローカルのスケジューラーによりプロセスが切り替えられて、ジョブB が実行を再開する (2)。プロセッサ3上のジョブA が最後にバリアに到達し、バリア同期が成立すると (3)、spin-wait 中のプロセッサ2上のジョブA は spin-wait を抜け、バリア成立後の命令を実行する (4)。プロセッサ1にバリア成立のメッセージが到達すると、ジョブA が実行を再開する (5)。

Implicit co-scheduling は、通信部分を変更するだけで実現でき、特別な大域スケジューラーを必要としないという利点がある。一般に通信部分はライブラリ化されているため、アプリケーションの開発者が手を加える必要はない。また、ローカルのスケジューラーは研究も多くなされており、優先順位付けや飢餓状態の回避など、様々な機能が取り入れられている。Implicit co-scheduling はこうしたローカルスケジューラーの性能を損わずに活用できる。また、投入されたジョブにおいて、プロセス同士の負荷のばらつき (load-imbalance) が大きい場合、多くのプロセスが受信待ち状態になる。ギャングスケジューリングにおいては、そのようなプロセスも強制的にクオンタムが割り当てられるが、受信待ち状態が解除されない限り、このクオンタムを浪費する事になる。この時間を他のプロセスに空け渡す事で全体の計算性能が向上する。

Two-phase *fixed* spin-block では、spin-wait 時間の調整が性能を左右する。これまでの研究では、コンテキストスイッチ時間の2倍の時間にしたもの[2)]と、ネットワークレイテンシの測定値および Microbenchmark の測定値を元に *LogP* モデル[5)]より導出したもの[4)]がそれぞれ発表されている。しかし並列アプリケーションの性能は様々な要因が関係するため、最適な spin-wait 時間を定める有効な方法を提示するのは難しい。

2.1 Berkeley による評価

今回の実験では、4) の報告を検証する。4) の性能評価では、16 台の Sun Ultra1 を 8 ポートの Myrinet スイッチで接続したクラスタを使用している。OS は Solaris2.5 で、Myrinet の通信ライブラリに AM-II、開発言語に Split-C を用いている。Implicit co-scheduling の実装は、Split-C の受信待ち部分に two-phase *fixed* spin-block を導入する事で行っている。計測には行列積やラディックスソート、フーリエ変換等のアプリケーションを用いており、通信量は一回の転送量の多いものと少いものの両方で測定したとしている。通信密度等の詳細については述べられていない。このような環境での implicit co-scheduling の性能は、実行速度をギャングスケジューリングと比べた結果、0.6 ~ 1.6 倍程度だとしている。性能にばらつきがあるのは、主に処理の粒度と負荷のばらつき (load-imbalance) に依るとしている。

3. SCore システムの紹介

Implicit co-scheduling の性能を評価するためのシステムとして、RWCP 並列分散システムソフトウェアつくば研究室 (pdslab) で開発している、SCore システム及び SCore 上に移植された MPI ライブラリ (MPIXX[6)]) を対象とした。

3.1 SCore システム

SCore は、RWCP で開発されている、クラスタ型並列計算機向けの並列実行環境である。SCore は同一アーキテクチャ同一性能の計算機を用いた (homogeneous) クラスターを対象としている。また、複数ユーザーによ

る共有利用を目的としており、ユーザーの対話性も視野に入れた設計が成されている[7]。

SCore システムは、以下に挙げるシステムによって構成されている。

PM Myrinet 用通信ライブラリ

SCore-S シングルユーザーによる使用を前提とした、並列実行環境

SCore-D マルチユーザーによる使用を前提とした並列ジョブスケジューラー

MPC++ 並列開発言語

また、今回測定に使った PC クラスタは、OS に NetBSD 1.2.1 を使用している。

3.1.1 PM

PM[8),9)] は、RWCP で開発された、Myrinet を対象とした通信ライブラリ / ドライバで、Myrinet に搭載されている RISC プロセッサ用のプログラムと、ホストコンピューター用のドライバ、低レベル通信ライブラリからなる。

PM はマルチユーザープロセスからの使用を前提として設計されている。マルチユーザープロセス環境では、物理的に一つしかないネットワークを複数のプロセスが使用できるように、ネットワークをソフトウェア的に多重化する必要がある。PM ではネットワークを多重化するために、チャネルという概念が導入されている。チャネルは複数個提供されており、プロセスはチャネルを介して通信する。ある送信ノードのチャネルから発信されたメッセージは、受信ノードの同じチャネルに配送される。

なお、チャネルの数には制限があるため、SCore-D ではチャネルをさらに多重化している。すなわち、一つのチャネルを複数のプロセスで共有する。SCore-D では、プロセス同士がチャネルを競合して使用する場合、チャネルを使用するプロセスを時分割で切り替える。このとき、現在のチャネルの状態を退避し、すでに退避されていた、次のプロセス用のチャネルの状態を復帰するという作業を行う。PM はこうしたチャネル切り替えのための機能として、チャネルの状態の退避 / 復帰機能を提供している。チャネルの状態はコンテキストと呼ばれ、その中身は主に送受信バッファからなる。

PM では、**Ack/Nack** 方式に一部変更を加えた方式によるフロー制御機能を持つ。この方式では、受信側で受信バッファの不足によりメッセージが受信できない場合は、**Nack** を送信ノードに返し、以降のメッセージを捨てる。送信側は **Nack** を受信したら、一度全ての送信を停止し、送信に失敗したメッセージ以降に送信したメッセージのみを全て再送する。これを **Ack** を受信するまで繰り返す。

この方式により、メッセージの受信状態が送信ノードから確認でき、安全にコンテキストを切り替えることができる。ただし、一度受信バッファのあふれによるメッセージの再送が起きると、ネットワークの負荷が高くなる可能性が高い。

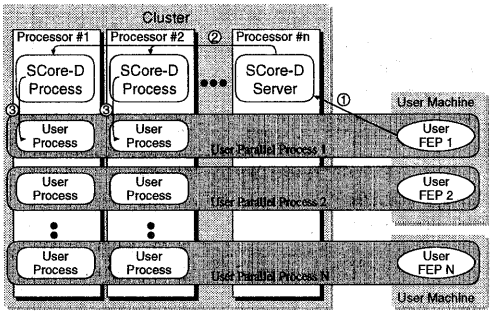

図 2 SCore-D 概念図

3.1.2 SCore-S

SCore-S はシングルユーザーを対象とした並列実行環境で、実体はランタイムライブラリとして提供されている。各ノードにおけるプロセスの起動・停止、PM を介した通信、エラーハンドリング等の機能を持つ。MPC++ で作成された実行プログラムは SCore-S のランタイムライブラリを含み、SCore-S の提供する機能を利用できるようになっている。

3.1.3 SCore-D

SCore-D[10)] はギャングスケジューリングの機能を提供するシステムで、MPC++ によって記述されており、SCore-S 上で動作する。

SCore-D システムでは、ユーザーは、クラスターと TCP/IP 接続された外の計算機からジョブを投入する。図 2 はその動作の流れを示している。まずユーザーの計算機上で実行された Front End Process(FEP) は、SCore-D サーバーと通信して、実行するアプリケーションを指示する (1)。SCore-D サーバーはクラスターの各ノードで動作している SCore-D デーモンに対し、各ノードで指定されたアプリケーションを起動するよう指示する (2)。各ノードにおいて、SCore-D デーモンは指示されたアプリケーションを子プロセスとして起動する (3)。以上の段階を経て並列アプリケーションの実行が開始する。

3.1.4 MPC++

MPC++ は、ローカル / リモートでのスレッド生成、ポインタのグローバル拡張、リダクション演算等の機能を提供する、並列アプリケーション開発言語である。対象とする並列プログラミングモデルは、SPMD である。MPC++ Version 2.0 には、C++ のテンプレート機能を使った高レベルな通信モデルのプログラミングをサポートする Level 0 と、上記に加えて自己反映計算機能をサポートした Level 1 がある。

MPC++ 自体はスケジューリング機能を持たず、受信待ち部分の実装は spin-wait になっており、並列ジョブスケジューリング機能は外部のスケジューラー (SCore-D) が受け持つ。

Spin wait

```
len = poll();
while (len == 0) {
    len = poll();
}
```

Two-Phase Spin-Block

```
len = poll();
if (len == 0) {
    for (i = spin_count; i > 0; i--){
        if ((len = poll()) != 0) break;
    }
    if (len == 0) {
        len = sleep();
    }
}
```

図3 spin-wait から two-phase spin-block への変更

3.2 Gang scheduling の実装

SCore-D のギャングスケジューリング機能は、シグナルを用いて実装されている。ジョブの切り替え時は、SCore-D プロセスから停止させるジョブに **SIGSTOP** を送り、再開させるプロセスに **SIGCONT** を送る。

ジョブの切り替え時には、ネットワーク内にまだ転送されていないメッセージが残っていない事を確認する必要があるが、PM は、3.1.1節で説明したような **Ack/Nack** によるフロー制御機能を持つので、ネットワーク内にメッセージが残っているかを確認できる。SCore-D では、送信した全てのメッセージが、正しく受信された (**Ack**)、あるいは再送が決定している (**Nack**) 事が判明した場合は、ネットワーク内が安定状態 (stable) であり、ジョブを切り替えても安全であると判断する。

4. two-phase spin-block の実装

今回の実装では、SCore 上で動作する MPI 通信ライブラリの通信部分に、two-phase *fixed* spin-block を導入した。SCore 上で動作する MPI 通信ライブラリは MPIXX[6] と呼ばれ、MPICH を SCore 上に移植したものである。MPICH では移植性を確保するために、Abstract Device Interface(ADI) と呼ばれる低位の機種依存部分と、実際の MPI の API を提供する機種非依存な部分に分離した構成になっている。MPIXX の受信部分の実装では、ADI の中で到着メッセージをポーリングして調べるようになっている。これらの箇所を、図3 に示すような方法で変更した。変数 **spin_count** によって、spin-wait 時間を調節する事ができる。ループにかかる時間を実時間に換算すると、今回使用したシステムでは、1000 回のループで約 650μsec となる。

5. 評　　価

5.1 実験内容

性能評価には RWCP の PC クラスター II を用いる。PC クラスター II の主な仕様を表1 に示す。

表1 PC クラスター II の仕様

Processor	PentiumPro
Clock	200MHz
Cache	512KB
Memory	256MB
Number of Processors	64
Myrinet Link Speed	160MB/sec
Myrinet Memory	1024KB

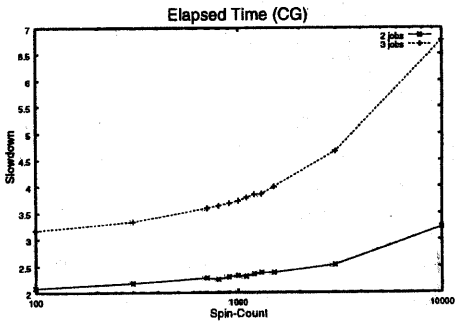

図4 Spin-Time に対する実行時間の変化 (CG)

ベンチマークプログラムは NAS 並列ベンチマーク (NPB) を用いた。使用した NPB のバージョンは 2.3 である。NPB ではバージョン 2.0 から、通信部分は MPI により記述されている。

この NAS 並列ベンチマークを、4節で示した変更を施した MPIXX を用いてコンパイルし、ギャングスケジュール機能を取り除いた SCore-D が作動している PC クラスター II 上で実行して測定した。使用プロセッサは全て 64 プロセッサとした。

今回の測定では、FT、CG の二つのベンチマークプログラムを用いて、実行時間を測定した。問題サイズはすべてクラス A を用いた。FT は三次元 FFT プログラムで、要素数は $256 \times 256 \times 128$ である。CG は共役勾配法を用いて連立一次方程式を解くプログラムであり、行列の大きさは 14000×14000 である。

第一の実験では、FT と CG それぞれにおいて、複数のジョブを同時に投入し、各ジョブのそれぞれの実行時間を測定した。第二の実験では、ギャングスケジューリングとの性能を比較した。比較に用いるギャングスケジューラは、SCore-D を使用した。実験の手法は第一実験と同様である。

5.2 実験結果

図4 と図5 は、各ベンチマークプログラムについてそれぞれジョブの投入数を 1,2,3 として計測した結果をグラフに示したものである。ジョブの投入数を 1 とした時の実行時間を 1 とし、投入数が 2,3 の時の実行時間の比をグラフの縦軸に示した。グラフの横軸は、spin-wait 期間のループ回数を示す。

CG において、implicit co-scheduling の効果は全

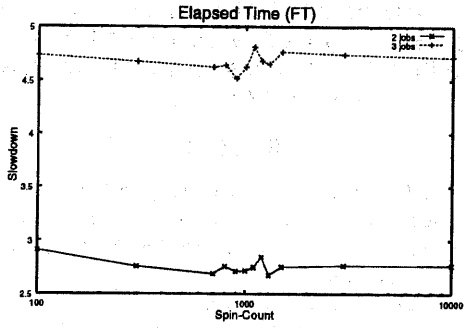

図5 Spin-Time に対する実行時間の変化 (FT)

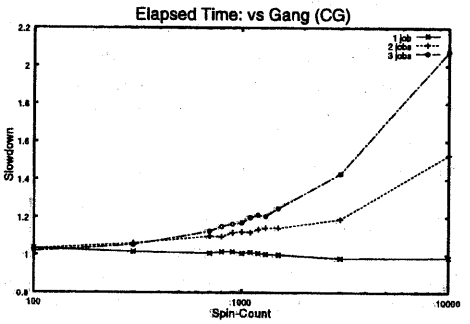

図6 ギャングスケジューラーとの実行時間の比較 (CG)

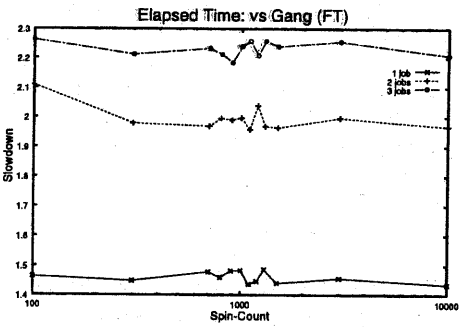

図7 ギャングスケジューラーとの実行時間の比較 (FT)

く見られず、即時ブロッキングの方が効率良くなっている事が分かる。FT では非常に効率が悪くなっているが、これは受信バッファあふれによる再送が多くなっているためだと推測しており、現在追試中である。(再送回数は多い時で 1 ジョブにつき 8 万回程度観測した)

図6と図7は、ギャングスケジューラーと性能を比較したものである。implicit co-scheduling とギャングスケジューリングにおいて、同条件でベンチマークプログラムを実行し、ジョブの平均実行時間を比較した。グラフの縦軸は、ギャングスケジューリング下でのジョブの実行時間に対する、implicit co-scheduling 下でのジョブの実行時間の比を示している。

CG において、ジョブが一つのときに、spin-wait 時間が増えるにつれて効率が良くなっているが、これはコンテキストスイッチの回数が減少したためである。ジョブが増えた場合には implicit co-scheduling による性能の改善は見られない。

5.3 考　　察

全体に、文献 4) で報告されているような性能向上は見られなかった。以下、それぞれのケースについて考察する。

CG ベンチマークにおいて、spin-wait 時間を長くすると性能が低下する傾向が見られる。FT ベンチマークでは逆に、spin-wait 時間を長くするとわずかではあるが性能が向上する傾向が見られる。

受信待ち部分において、即時ブロックする場合と spin-wait する場合の性能を測定した予備実験の結果から、今回計測に使用したシステムでは CG ベンチマークの計測において、spin-wait した場合に極端な速度低下を引き起こし、即時ブロックをした場合は比較的良好な性能を示す事が分かっている。逆に、FT ベンチマークでは spin-wait をした方が、即時ブロックをした場合よりも性能が低下しにくいという事が分っている。この原因はまだ判明していないが、今回の実験結果にも予備実験の結果と同様の現象が起きている。

FT ベンチマークにおいて、受信バッファのあふれによるメッセージの再送が性能を低下させている。今回の計測では、同種のジョブをほぼ同時に複数投入しているため、全てのジョブがほぼ同時に通信部分の処理を開始しており、ネットワークの混雑の原因となっている。参考までに、CG ベンチマークにおいて、反復処理の中で、通信処理を開始する部分の時刻を記録したものを図8に示す。グラフには、反復の 1 回目から 10 回目までを記録している。全てのジョブにおいてほぼ同時に通信処理を開始する様子が分かる。

Two-phase spin-block において受信バッファが常に詰っている状況では、spin-wait 期間にメッセージ受信が成立しても、メッセージはバッファリングされていたものなので、すでに送信元のプロセスは切り替えられている可能性がある。そのため、通信処理の同期という本来の目的が達成されていないものと推測する。

FT では、ギャングスケジューリングに比べて著しい性能低下が見られる。ギャングスケジューリングでは、ジョブを切り替える時に 3.1.1節で述べたようなフロー制御を行う。この時、ネットワークが安定状態になるのを待っている間に受信バッファに残っていたメッセージが処理され、受信バッファ不足が解消されてメッセージ再送を少くする効果があるものと推測する。一方、implicit co-scheduling では前述のようなフロー制御が働かないため、受信バッファ不足の解消が遅れる傾向があるものと推測するが、今回の実験で得たデータでは現象の詳細は判断がつかない。受信バッファの状態・通信量の対時間密度等の情報を測定した上で検証する必要がある。

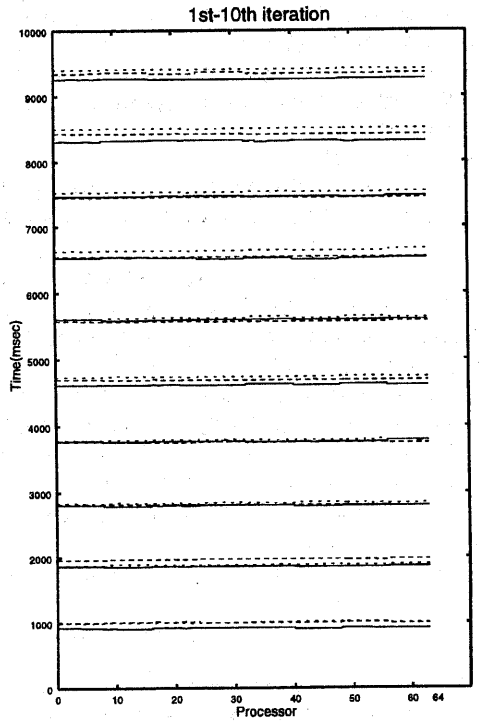

図8 CG ベンチマークにおける各プロセスのスケジューリング状況 (1–10 回目)

今回の計測では、各プロセス毎の負荷のばらつき (load-imbalance) を考慮した計測をしておらず、implicit co-scheduling に有利な場合での性能は評価していない。処理の粒度や負荷のばらつき、プロセッサ数を変えた場合の性能測定を今後の課題とする。

6. ま と め

Berkeley NOW プロジェクトの提案する Implicit co-scheduling を、RWCP の PC クラスタ上で動作する MPI ライブラリに実装し、NAS 並列ベンチマークによる性能評価を行った。今回の実験では NAS 並列ベンチマークのうち、FT と CG を用いて測定したが、NOW プロジェクトの評価結果とは異なり、ギャングスケジューリングに対して実行時間にして最大 2.3 倍程度という大幅な性能低下が見られた。また、最適な spin-wait 時間が存在すると言われているが、今回の実験では spin-wait を長くするに応じて性能が低下する現象が見られた。

原因の一つとして、受信バッファのあふれから来るメッセージの再送による、ネットワーク負荷の上昇があるが、今回の実験データでは、受信バッファの状態・通信量等のデータが不足しており、詳細はまだ分っていない。これらのデータの計測とともに、アプリケーションの種類・処理の粒度・負荷のばらつき・プロセッサ数等を変えた場合の性能測定を今後の課題とする。また、今回は受信側で two-phase *fixed* spin-block を行うという単純な実装を用いたので、今後は送信側でも同様の処理をするなどして計測する必要がある。

参 考 文 献

1) Ousterhout, J. K.: Scheduling Techniques for Concurrent Systems, *Third International Conference on Distributed Computing Systems*, pp. 22–30 (1982).
2) Dusseau, A.C., Arpaci, R.H. and Culler, D.E.: Effective Distributed Scheduling of Parallel Workloads, *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems* (1996).
3) Arpaci-Dusseau, A. C. and Culler, D. E.: Extending Proportional-Share Scheduling to a Network of Workstations, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)* (1997).
4) Arpaci-Dusseau, A.C., Culler, D.E. and Mainwaring, A. M.: Scheduling with Implicit Information in Distributed Systems. To appear in *Sigmetrics'98 Conference on the Measurement and Modeling of Computer Systems.*
5) Culler, D., Liu, L. T., Martin, R. and Yoshikawa, C.: LogP Performance Assessment of Fast Network Interfaces, *IEEE Micro* (1996).
6) O'Carroll, F. B., Hori, A., Tezuka, H., Ishikawa, Y. and Matsuoka, S.: Implementing MPI in a High-Performance, Multithreaded Language MPC++, *SWoPP'96*, pp. 141–146 (1996).
7) 堀敦史, 手塚宏史, 石川裕, 曽田哲之, 原田浩, 古田敦, 山田務, 岡靖裕: ワークステーションクラスタにおける並列プログラミング環境の実現, *IPSJ SIG Notes 96-OS-73*, Information Processing Society of Japan, pp. 121–126 (1996).
8) 手塚宏史, 堀敦史, 石川裕: ワークステーションクラスタ用通信ライブラリ PM の設計と実装, *JSPP '96*, pp. 41–48 (1996).
9) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating Coordinated High Performance Communication Library, *High-Performance Computing and Networking '97*, Lecture Notes in Computer Science, Vol. 1225, Springer-Verlag, pp. 708–717 (1997).
10) Hori, A., Tezuka, H. and Ishikawa, Y.: User-level Parallel Operating System for Clustered Commodity Computers, *Proceedings of Cluster Computing Conference '97* (1997).