特別研究報告書

SPLASH ベンチマークによる超並列計算機 JUMP-1 の性能評価

指導教官　　富田 眞治 教授

京都大学工学部情報学科

齋藤 康二

平成 12 年 2 月 15 日

SPLASHベンチマークによる超並列計算機JUMP-1の性能評価

齋藤 康二

内容梗概

超並列計算機JUMP-1は、最大1024プロセッサからなる分散共有メモリ型並列計算機である。JUMP-1はクラスタ構成をとっており、その構成からクラスタ内とクラスタ間にわかれたメモリの管理を行っている。クラスタ内のメモリ管理はその速度が問われるため、完全にハードウェアで構成されたキャッシュコントローラにより実行される。一方、クラスタ間のコヒーレンス制御は柔軟に複雑な処理を行う必要があるため専用プロセッサMBP-lightによって行われる。これらで構成される高機能メモリシステムが、細粒度並列計算時における通信レイテンシの隠蔽、削減を可能としている。

1クラスタは4つのプロセッサカードと1つのクラスタメモリ、その管理を行うMBP-lightにより構成されており、各クラスタ間はRDT(Recursive Diagonal Torus)ネットワークで接続されている。また、プロセッサカードは1次キャッシュ内蔵の要素プロセッサ、外部2次キャッシュ、キャッシュコントローラからなっている。

現在、JUMP-1では、マルチクラスタでのアプリケーションの実行を試みている。本研究では、1クラスタでのアプリケーションの実行を行い、その性能評価を行った。

評価には、共有メモリ型並列システムのベンチマークであるSPLASH(Stanford Parallel Applications for Shared-Memory)-2のアプリケーションを用いた。SPLASH-2のアプリケーションはC言語で記述されており、PARMACSマクロを使って様々な並列システムへの拡張を可能としている。PARMACSマクロはArgonne National Laboratoryで開発され、並列プロセスの生成、同期や共有メモリの割り当てなどに関する基本的な処理を行うマクロである。実際の、SPLASH-2の実装は、PARMACSマクロを対象となるシステムに合わせて書き換えることによってなされる。アプリケーション中のPARMACSマクロはm4 preprocessorによって変換され、実行可能な状態になる。

SPLASH-2は、4つのcomputational kernelと8つのcomplete applicationsか

らなっている。本研究で評価に用いたプログラムは、RADIX kernel、LU kernel、OCEAN application である。RADIX kernel は整数 radix ソート、LU kernel は LU 分解を行う計算 kernel で、OCEAN application は対象となる海域の渦や海流による動きをシュミレーションするアプリケーションである。

評価の結果、SPARCserver1000 と比較して、JUMP-1 では 10 ~ 20 パーセント程度少ないサイクル数でアプリケーションが実行されることを確認した。また、プロセッサ台数による性能向上率は同等か、あるいは問題数に比べて通信量が多いようなアプリケーションにおいては SPARCserver1000 より良い結果が得られた。

また、本研究ではタイリングによるループの最適化技法の JUMP-1 における有効性についても評価した。タイリングは、各記憶階層において多重に適用可能であることが知られている。1 次キャッシュだけではなく、各メモリ階層に合わせて多重にタイリングを施すことによって、各階層でのヒット率を向上させることができ、アプリケーションのより高速な実行が期待できる。

JUMP-1 では各プロセッサ毎に 1 次、2 次キャッシュを持ち、クラスタ内においても階層的なメモリ構造が見られる。そこで、LU kernel に多重にタイリングを施し JUMP-1 で実行し、その有効性に関しての評価を行った。

LU kernel は外積形式ガウス法に 1 回タイリングを施したようなアルゴリズムで、並列実行時には各プロセスは自分の担当するタイルの処理を行っている。本研究では、タイル内の行列についてもう一回のタイリングを施し、各メモリ階層に合わせたマルチレベルタイリングを実現した。

その結果、2 次キャッシュまでタイリングを施すことで、10 ~ 30 パーセント程度の高速化が行われ、JUMP-1 におけるタイリングによる最適化の有効性を示すことができた。

Evaluation of a Massively Parallel Processors JUMP-1 using SPLASH benchmark

Yasuji Saito

Abstract

Massively Parallel Processor JUMP-1 is composed of maximum of 1024 processors with distributed shared Memory. JUMP-1 adopts a cluster architecture. Therefore JUMP-1 provides two level memory management, within cluster and inter cluster. Within cluster, Cache Controller maintains the coherence of primary cache and external secondary cache. Cache Controller is completely hardwired because its speed for usual secondary cache operations is critical to keep the total system performance high. On the other hand, inter-cluster, a memory controller co-processor called MBP-light manages global transaction flexibly. The intelligent memory system of JUMP-1 can hide or reduce fine-grained inter-processor communication latency.

A cluster of JUMP-1 consists mainly of four processor units, a cluster memory and a memory control co-processor MBP-light. Inter cluster Network is RDT(Recursive Diagonal Torus)network. A processor unit is composed of a element processor with primary cache, external secondary cache and Cache Controller.

Now, we try to implement applications on multi-cluster. In this paper, we evaluated applications on one-cluster.

We use SPLASH(The Stanford Parallel Applications for Shared_Memory)-2 benchmark in this evaluation. The SPLASH applications are written in C, extended with PARMACS macros. These macros were developed at the Argonne National Laboratory and offer basic primitives for synchronization, creation of parallel processes and shared memory allocation. For implementation, we adapt these macros to our system . These macros in applications can be used with the m4 macro-preprocessor.

SPLASH-2 is composed of a set of 4 computational kernels and 8 complete applications. We evaluated JUMP-1 with RADIX kernel, LU kernel, OCEAN application. The RADIX kernel implements the integer radix sort method. The LU kernel, it factors a dense matrix into the product of a lower triangular and

upper triangular matrix. The OCEAN application studies large scale ocean movements based on eddy and boundary currents.

The result on JUMP-1 is about 20 percent better than one on SPARC-server1000 that is SMP with same processors. Parallel processes on JUMP-1 execute better the OCEAN application with fine-grained message and small problem size.

We also evaluated the effectiveness of a loop optimize technique, Tiling on JUMP-1. It is known that Tiling can be applied to each class of memory hierarchy. Tiling for the size of each class of memory hierarchy improves the hit ratio of each memory hierarchy.

A processor of JUMP-1 has two private data caches, primary cache and external secondary cache and the memory system is hierarchical. In this paper, the LU kernel was applied multi-level Tiling to, and it evaluated on JUMP-1.

The argorithm of the LU kernel is optimization of outer product, it was already applied Tiling to once. We applied Tiling to their submatrix(block) once more for the size of each class of memory hierarchy of JUMP-1.

We get the 10 ~ 30 percent better result than non-tiling LU kernel. It indicates that tiling technique is effective on JUMP-1.

SPLASH ベンチマークによる超並列計算機 **JUMP-1** の性能評価

目次

第 1 章　はじめに

我々は、現在、CC-NUMA 型並列計算機 JUMP-1 を開発している。JUMP-1 は、最大 1024 の要素プロセッサより成り、分散共有メモリ (DSM:Distributed Shared Memory) をによって、通信がなされる。JUMP-1 はクラスタ構造をとっている。1 つのクラスタは 4 つの要素プロセッサと 1 つのメモリユニットで形成されており、各クラスタ同士は大規模な静的網で接続されて全体のシステムを構成している。

このような構成を反映して、JUMP-1 では、DSM 管理はクラスタ間とクラスタ内の 2 段階に階層化されている。JUMP-1 における高機能メモリシステムは、共有メモリを用いたプロセッサ間の通信におけるレイテンシを隠蔽し、削減している。クラスタ内のコヒーレンス制御はその速度を問われるためハードウェアによって高速に行われ、クラスタ間では、処理の高速性とともに柔軟性や処理の複雑さに対処しなければならないためメモリ管理の専用プロセッサでソフトウェアによって制御される。

現在、JUMP-1 の状況は、1 クラスタでの数値計算等のアプリケーションの実行は確認され、マルチクラスタでの動作の確認を行っている。

そこで、本研究では、共有メモリ型並列システムのベンチマークプログラムである SPLASH-2 のプログラムを JUMP-1 上で実行し、JUMP-1 の 1 クラスタでの性能評価を行う。

SPLASH-2 では並列プロセスや共有メモリ管理を行う命令が PARMACS マクロと呼ばれるマクロを用いて記述されており、それらを各システムに合わせて変換することで、そのシステム上での実行が可能となる。本研究では、その JUMP-1 での実装方法を示す。

また、近年、プロセッサとメモリの速度差が大きくなっており、そのためプロセッサの性能を十分に発揮することが難しくなっている。しかし、キャッシュを効果的に利用し、ヒット率を向上させることで、プロセッサの性能をより引き出すことが出来る。その手法を 1 次キャッシュだけでなく、各メモリ階層に用いることで、さらに、プロセッサとメモリの速度差を埋めることが可能である。

JUMP-1 では、各プロセッサ毎に 1 次、2 次キャッシュを、各クラスタ毎にクラスタメモリを持ち、それらは階層的な構成をとっている。そこで、本研究では、SPLASH-2 のプログラムに、多重にタイリングを施し、JUMP-1 のメモリ

階層の評価も行う。

本稿では、第 2 章で対象となる JUMP-1 の概要として、そのクラスタ構成やメモリシステムについて述べる。第 3 章で、SPLASH-2 とその JUMP-1 上への実装について述べ、第 4 章でマルチレベルタイリングについて述べたあと、第 5 章で本研究で得られた結果とその考察を行う。

第 2 章　JUMP-1 の概要

この章では、超並列計算機 JUMP-1 の設計方針と物理的構成、メモリシステムについて述べる。

2.1　JUMP-1 の設計方針

並列計算機の性能は、局所的な演算とプロセッサ間の通信の性能による。要素プロセッサとして、市販の RISC プロセッサを用いると、局所的な演算性能は高くなるが、長時間の通信レイテンシを考慮して設計されているということは少なく、通信完了を待たなければならないような状況になった場合、システムの性能の低下の要因となる。要素プロセッサとして、並列計算機専用のプロセッサを開発した場合、RISC プロセッサでの問題点は解決されるが、局所的な演算性能が RISC プロセッサと同等の並列計算機専用プロセッサの開発には非常に大きなコストがかかる。JUMP-1 では局所的な演算の性能を確保するために、市販の RISC プロセッサを用いている。

JUMP-1 では要素プロセッサ間の通信は共有メモリによってなされる。通常の共有メモリ型並列計算機では、通信が頻繁に行われ、さらにその通信パターンが非定型である場合、通信レイテンシによって性能が低下してしまう。そこで、JUMP-1 では共有メモリ管理専用の高機能メモリシステムを構築することによって、細粒度の通信を行った場合にも通信レイテンシを削減、隠蔽し、通信のオーバーヘッドを下げようとしている。

また、JUMP-1 はクラスタ構成を採用している。これは、

1. クラスタ化により処理に内在する局所性を利用し、高速化を可能とする。
2. クラスタ化により通常の数値計算に必要なネットワークのコストを低下できる。
3. クラスタ化によりネットワークの使用効率を上昇させることが可能である。

4. 階層化された同期を含む共有記憶プロトコルを採用することにより、ネットワーク負荷を低下させることが可能である。
5. 現在の基板実装技術ではクラスタを 1 つの基板上に実装が可能である。
6. プロセッサから直接参照できる主記憶の量が増大する。
7. クラスタ内部の結合密度が高くなると、自動並列化コンパイラによる局所並列性の抽出が容易になる。

などの理由による。

よって、JUMP-1 において高速実行可能な並列プログラムを開発する場合、クラスタ内とクラスタ間を意識する必要があるが、大規模問題の並列化においては、アルゴリズムを工夫することで階層的な並列性を実現できることが多いと考えられる。

2.2 JUMP-1 の構成

JUMP-1 はクラスタを 1 つの単位として、各クラスタ間を RDT(Recursive Diagonal Torus) ネットワークで相互に結合するという構成をとっている。その結合網である RDT は 2 次元トーラスが Fat-tree 状に階層化された構造を持つ。クラスタ間の通信は、RDT-Router が他のクラスタの RDT-Router と RDT-Network を通じてデータ転送することで実現される。

各クラスタ内では、4 つのプロセッサカードと 1 つのメモリユニットがクラスタバスによって結合されている。クラスタバスは 4 つの要素プロセッサが共有しているメモリバスである。

図 1 に JUMP-1 のクラスタ内の構成を示す。各クラスタは 1 枚のプリント基板上にプリントされている。

- プロセッサカード
  クラスタ内の 4 つの要素プロセッサは、それぞれ 1 枚のプリント基盤上に実装されており、これをプロセッサカードと呼ぶ。各プロセッサカードは、20KByte の命令キャッシュと 16KByte のデータキャッシュを内蔵する要素プロセッサ SuperSPARC+、1MByte の外部 2 次キャッシュとその制御を行う Cashe Controller(以下、CC) から成っている。図 2 にプロセッサカード内の構成を示す。
- メモリユニット
  16MByte のクラスタメモリとその制御を行うプロセッサ MBP-light から構

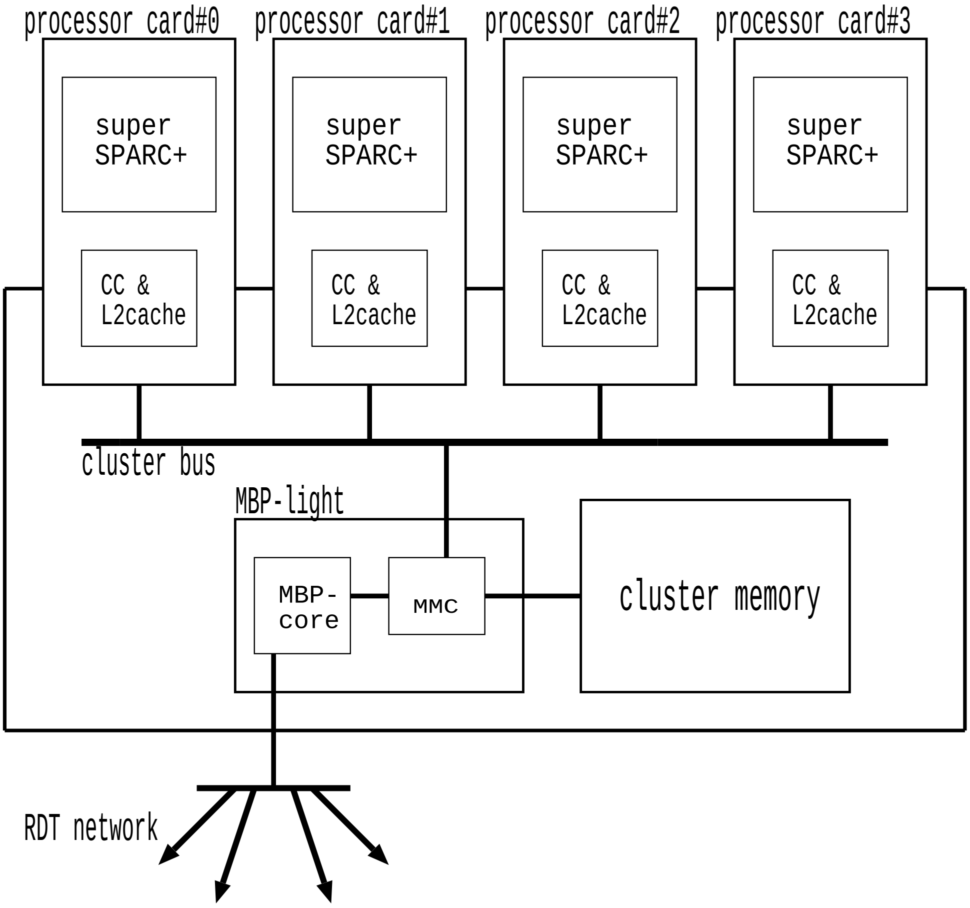

図 1: JUMP-1 のクラスタ構成

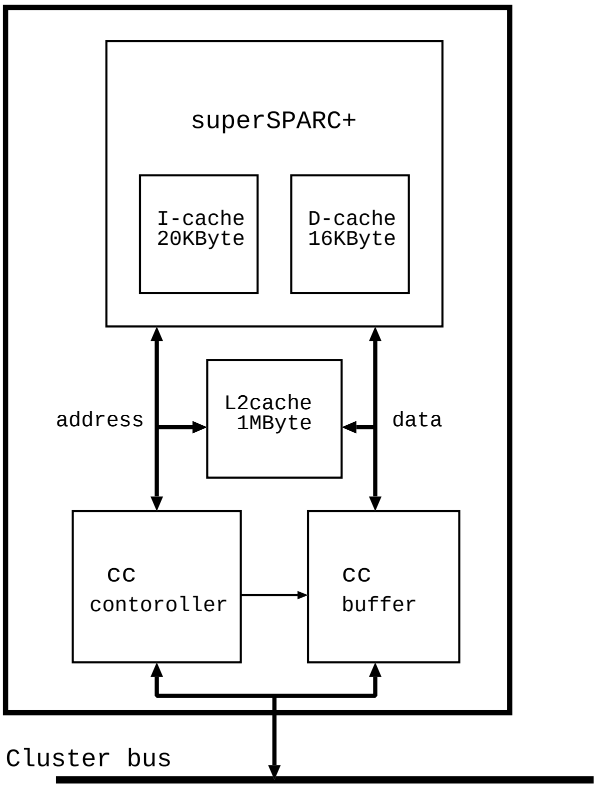

図 2: プロセッサカードの構成

成されている。クラスタメモリは主記憶として利用される他、クラスタ間でのメモリアクセスに対して、他のクラスタメモリのコピーとしても利用される。つまり、プロセッサから見ると 3 次キャッシュとしても利用できるということである。

JUMP-1 はクラスタ構成をとっているため、クラスタ間とクラスタ内という形でメモリアクセスに関しても 2 階層に分かれており、以下に示す MBP-light、CC によって管理されている。

- MBP-light

  MBP-light は、MMC と MBP-Core から成り、クラスタメモリの管理を担当している。

  MBP-Core は、主にクラスタ間にまたがるデータの管理を行うプロセッサで、MMC 及び RDT-Router からの処理要求パケットをうけとり、対応するメモリ処理をソフトウェアによって柔軟に処理している。RDT 側からの到着パケットの処理でも、それが共有状態を変更するようなものである場合、その間にメモリアクセスがないよう、クラスタバスをロックしなけれ

ばならない。しかし、共有状態の変更を要するパケットであるかは、実際にそのパケットの処理を開始しなければ判断できない。そこで、RDT 側からのパケットの到着があると、その性質を判断するまでコヒーレンス制御のために他のパケットの処理を行うことができなくなっている。それゆえ、クラスタ間にまたがるデータアクセスはクラスタ内に閉じたデータアクセスよりも大幅に時間がかかる。

MMC はクラスタメモリのコントローラである。クラスタ内でのメモリアクセスや MBP-Core からのメモリアクセスはパケットという形で与えられ、主記憶に物理的に近い位置に MMC を設置して、パケットの内容に対応したメモリアクセスをハードウェアによって高速に処理する。クラスタメモリはデータの共有状態を示すタグを所持しており、メモリアクセスが行われると、MMC はメモリオブジェクトのタグの値を見て、データがクラスタ内に閉じていればクラスタバスにパケットを流して終了するが、データがクラスタ内で閉じていなければ、MBP-Core に処理を依頼する。

- CC

  クラスタ内の共有メモリ、主に 1MByte の外部 2 次キャッシュの管理を行っており、CC は共有バスベースのスヌープ方式によって 2 次キャッシュ間のコヒーレンスを制御する。2 次キャッシュの状態は CC によって管理されており、MOESI のプロトコルに類似しているが、CC はクラスタ間の共有状態も管理している。MBP-light がソフトウェアで処理を行うのに対し、クラスタ内での共有メモリ管理にかかる速度がシステム全体の性能を大きく左右すると考えられることから、完全にハードウェアによって構成されている。

  また、JUMP-1 のキャッシュシステムは、各メモリオブジェクトに対して適切なコヒーレンスプロトコルを指定することで性能の向上を図っており、CC は主に 2 種類のコヒーレンスプロトコルを提供している。write-invalidate プロトコルと write-update プロトコルであり、ユーザはこれを指定することが出来る。超並列計算機において洗練されたプログラムを実行する場合、write-update プロトコルは write-invalidate プロトコルよりも優れていると言われているが、問題の性質や計算機資源によっては write-invalidate の方が優れている場合があるため、各データに対しコヒーレンスプロトコルを切り替えることは有効である。

また、クラスタ内だけのメモリに注目すると、各要素プロセッサに内蔵されている1次キャッシュ(データキャッシュ)、各プロセッサカードの外部2次キャッシュ、クラスタメモリといった階層的な構成になっている。さらに、上記のようにクラスタメモリがクラスタ間でのメモリアクセスにおいて3次キャッシュとして利用されることを考えると、各階層に合わせてタイリングを行い、キャッシュのヒット率を向上させることの有効性が期待される。

第3章　SPLASH-2の実装

SPLASH-2[?]は、共有メモリ型並列システムのベンチマークプログラムであり、これは従来のStanford Parallel Apricatiоns of SHared-memory(SPLASH)の拡張及び変更を加えたもので、結果、

1. 科学技術計算やグラフィックス分野の広範囲な計算を代表する。
2. よりよいアルゴリズムを実装する。
3. より正確に計算機アーキテクチャを想定する。

といった利点を持っている。

SPLASH-2には4つの数値計算kernelと8つの完全なappricationが用意されている。

3.1　PARMACSマクロ

SPLASH-2のプログラムはC言語で記述されており、PARMACSマクロによって並列システムで動作するように拡張される。このマクロはthe Argone National Laboratoryで開発され、m4 preprocessorを使ってマクロの変換を行うことで、アプリケーションを実行可能な状態にできる。

PARMACSマクロは並列プロセスの同期やプロセスの生成、共有メモリの割り当てなどにおける基本的な処理を行う。これらのマクロは異なるアーキテクチャにおいて、SPLASH-2を可能とするために用いられている。実装を行う場合、単純にマクロの内容をシステムに合わせて変更するだけで他のシステムでの実行が可能となる。つまり、このようなマクロを用意することで、実行する対象のアーキテクチャと独立にアプリケーションをプログラミングすることができている。Encore Multimax、SGI、Alliantなどいくつかのシステムへの実装は公開されており、それを用いて実装することができるが、そういったシス

テム以外へのマクロの実装や特殊な処理を行う場合にはユーザが手を加える必要がある。

SPLASH-2ではnull PARMACS macrosが提供されており、これを用いてアプリケーションのマクロを変換することで、並列ではなく、1プロセスでの実行が可能となる。このnull macrosに適切なコードを挿入することで、アプリケーションを様々な異なる並列モデルへと拡張することができる。また、この特徴を利用して、アプリケーションが並列実行時にどのように振舞うかを調べることもできる。

以下にSPLASH-2で提示されているPARMACSマクロにうちいくつかを挙げ、その役割を述べる。

- MAIN_INITENV
  アプリーションを実行する上で必要となる種々の変数等に関して、初期化を行う。
- MAIN_ENV
  このマクロはSPLASH-2を実行するときに必要な変数等の定義を行う。実際には、lock変数やbarrier変数などの定義と、システムで必要なライブラリの宣言などを行っている。アプリケーション中では最初に一度だけ呼ばれる。
- CREATE
  CREATEは指定された数の並列プロセスを生成するためのマクロである。
- WAIT_FOR_END
  WAIT_FOR_ENDはCREATEによって生成されたプロセスが全て到達するまで、そこでプロセスをblockしておく。
- G_MALLOC
  G_MALLOCは共有メモリ上に、C言語のmallocなどと同様にメモリ領域を確保するためのマクロである。
- G_FREE
  G_MALLOCで確保した領域を解放するためのマクロである。G_MALLOCと同様にC言語におけるfreeと同じ役割を果たす。
- LOCK
  相互排除領域をlock変数を用いてprotectするマクロである。相互排除領域には、同時には1つのプロセスのみがアクセスすることが許されており、

他のプロセスが既にアクセスしている最中ならば、誰もアクセスしていない状態になるまで、その場で待ち続ける。プロセスが相互排除領域での処理を終了すると、UNLOCK マクロで lock を解除して、他のプロセッサからのアクセスを許可する。

- UNLOCK

  LOCK を解除して、その相互排除領域への他プロセッサのアクセスを可能にする。

- BARRIER

  指定した数のプロセスが同じ BARRIER 命令に到達するまで、実行をブロックするためのマクロである。全てのプロセスが到達すると、ブロックされていた全プロセスは解放され、後続の処理を行う。

- CLOCK

  呼び出された時点での時間を返す。

また、SPLASH-2 では上であげたマクロの他に以下のようなマクロが提供されているが、これらのマクロはアプリケーションの実行においてはまったく必要がなく、よって JUMP-1 への実装の必要もない。

- GETSUB
- EVENT
- PAUSE

3.2 SPLASH-2 の JUMP-1 への実装

以下に上記の PARMACS マクロのうちの幾つかにおける、JUMP-1 への実装方法を示す。

- CREATE

  JUMP-1 では、このマクロが実行される段階でプロセスを生成して処理を行うのではなく、各プロセッサにはあらかじめ master のプロセッサと同じプログラムが送られており、BARRIER のような形でそれらの実行は止められている。

  master プロセッサのみ SPLASH-2 のアプリケーションをはじめから実行し、CREATE マクロの命令が実行されると、BARRIER が解除されて、master 以外のプロセッサがアプリケーションの並列プロセスを開始する。

- WAIT_FOR_END

後述のBARRIERと同様な方法で実現している。JUMP-1の場合は、master以外のプロセッサは各々のプロセスを終了した時点で、終了したプロセッサ数を数える変数の値を1インクリメントして終了する。masterプロセッサは全プロセッサの並列プロセスの終了を待って、その後、残された処理を実行する。

- G_MALLOC

  今回のJUMP-1への実装では、SPLASH-2では扱うアプリケーションの問題サイズによって、G_MALLOCで要求されるメモリサイズが一定なので、SPLASH-2のアプリーションを実行する前に静的にその領域を確保しておき、それを実際に要求された段階で渡している。静的にメモリ領域を確保するmallo_initという関数をMAIN_INITENVで呼びだすことで、アプリケーションの実行前に準備を行っている。また、並列実行時に同時に要求が起こった場合に備えて、LOCKをかけて順次処理を行うようにしている。

- G_FREE

  実際にSPLASH-2のアプリケーション中で使われることがほとんどなく、本研究で用いたアプリケーションでも使われていないため、現在未実装のままである。

- LOCK

  JUMP-1への実装においては、superSPARC+のSWAP命令を用いて、lock変数の制御を行っている。SWAP命令はSPARCアーキテクチャのバージョン7で追加された命令で、現在使用されている一部のシステムでは、この命令をハードウェアで実行しない。

  SWAP命令が実行されると、対象とするメモリの値をロードし、そこに格納したい値をストアする。つまり、メモリの内容とストアバッファの内容を交換するような形で動作する。通常のロードの際プロセッサはブロックされ、ロードの完了とともにブロックは解除されるが、SWAP命令ではロードをした後、すぐにプロセッサのブロックを解除せず、ストア・バッファが空になり、メモリ操作が始まるまでブロックしつづける。この操作は不可分に行われるので、ほかの操作がロードとストアの間に割り込むことはできない。

  JUMP-1でのLOCKの実現方法は以下の通りである。

lock 変数を 0 に初期化しておく。lock 変数が 0 のとき、その領域へアクセスしているプロセスは存在しないことを、-1 のときは、その領域へアクセスしているプロセスが存在することを示す。
LOCK 命令が出されると、上記の SWAP 命令を用いて、lock 変数を-1 と交換する。この際、上記のとおり、他の操作による割り込みは起こらない。交換して得られた値が-1 の場合、他のプロセスがアクセスしているので、その場で 0 が得られるまで繰り返し同じ操作を行う。得られた値が 0 の場合、相互排除領域へのアクセス権を得る。その後、相互排除領域での処理が終了すると、UNLOCK 命令によって、先程とは逆に 0 と lock 変数を交換する。これによって、その領域にアクセスしているプロセッサがいなくなったことになり、他のプロセッサがアクセスできるようになる。

- BARRIER

  BARRIER では次の 3 つの変数を用いる。プロセス数を数えるための変数 barrier.count[t]、LOCK を用いて実装したため lock 変数 barrier.lock、そして複数回の同じ BARRIER の呼びだしに備えるための処理に使う変数 barrier.turn である。複数回の同じ BARRIER の呼び出しに備えるというのは、非常に短い間隔で同じ名前の BARRIER が使われるような場合、間違って意図していない BARRIER を抜けてしまう、などということが起こらないようにするためである。実際には、同じ名前の BARRIER に対して、barrier.turn が 0 と 1 の値を交互にとることで、同じ名前でも違う BARRIER でブロックされるような形になっている。
  以下に、本研究において実装した BARRIER の実行時の流れを示す。ただし、N 個のプロセスの到達を待つ BARRIER とする。
  BARRIER 命令が実行されると、まず LOCK をかけて、barrier.turn を読み、以後、その値で示される側の BARRIER であることを確認する。その後、barrier.count[t](t=barrier.turn) を確認して、自分がその BARRIER にとって、最後のプロセスかどうかを確かめる。
  最後のプロセスではない場合、barrier.count[t](t=barrier.turn) の値を 1 インクリメントした後、LOCK を解除し、barrier.count[t] が N になるまで待つ。
  最後のプロセス (barrier.count[t]=N-1) の場合、barrier.turn を変更し (0→1 or 1→0)、barrier.count[barrier.turn] を 0 にする。これで、次回に、この回

で使わなかった方のBARRIERを使う準備が出来たということになる。その後、barrier.count[t]の値をNにして、待っていた他のプロセスを解放し、自分もLOCKを解除してBARRIERを抜けて、次の処理へ向かう。

- CLOCK
  アプリケーション中では、時間測定のスタート地点とストップ地点でCLOCKマクロを実行し、返した値の差を取ることで実行時間を求めている。一方、JUMP-1では、MBPに命令を送りcounterを動かして実行時間ではなくクロック数を測っている。つまりJUMP-1では、測定したい部分のをサイクル数を直接測定しているということになる。
  よって、JUMP-1では実装時に他のマクロと同様にm4 preprocessorによって変換を行っても、正しく測定することはできない。実際に実装する際には、まず、MBPにcounterをリセットさせる命令を出し、測定したい個所の前後でcounterのスタート命令とストップ命令を実行させるようにプログラムを書き加えることで必要なデータを得ることができる。

3.3 評価アプリケーションについて

本研究において、JUMP-1に実装し、その評価に用いたのは、SPLASH-2のアプリケーションのうち、以下の3つである。

- LU
- RADIX
- OCEAN

それらの説明を簡単にしておく。

LU LUは密行列を上三角行列と下三角行列に分解する。n×nの密行列Aは部分行列の要素上の時間的局所性を利用するために、B×BのブロックのN×N(n=NB)の配列に分割される。ブロックは所有権を持つプロセッサによって更新されつつ、2次元のscatter行列をもちいてブロック所有権を割り当てることで、通信を減らそうとしている。
ブロックの大きさBはキャッシュミス率を低く抑えるためには十分に大きく、負荷バランスを良く保つためには十分に小さくするべきであるが、実際には、かなり小さいブロックサイズ(B=8またはB=16)が良い負荷バランスを示す。ブロック内の要素は空間的局所性を利用するために隣接して配置され、ブロックは所有しているプロセッサに対してlocalに配置される。

分割されたブロックを A[i][j] という形で表すとすると、まず、ブロック A[k][k] を所有するプロセッサがそのブロックに関して local に計算を行う。この処理が終了するのを待って、ブロック A[i][k](i≥k)、ブロック A[k][j](k≤j) に関しての処理をそれらを所有するプロセッサが行う。この段階で先に計算されたブロック A[k][k] を参照する形で通信が行われる。それらの処理終了後、次にブロック A[i][j] をブロック A[k][j]、ブロック A[i][k] を参照して書きかえる。ここで、処理を行うのはブロック A[i][j] を所有するプロセッサであるため、ブロック A[k][j]、ブロック A[i][k] の参照において通信が行われる。

RADIX　文献 [?] に記述された方法を基にしている。アルゴリズムは反復的で、key の数字の基数 r 毎に一回の反復を行い、各反復毎に、プロセッサは割当てられた key を用いて local な histogram を生成する。local な histogram は global な histogram に集計されて、最終的には、各プロセッサはこの global は histogram を用いて、次回の反復のための新しい配列へと key を置換する。置換の段階で all-to-all な通信を要求し、基本的に送信時に置換が決定されるので、key は読み込みよりも書き込みを通じて通信が行われる。

詳しくは、文献 [?][?] を参照のこと。

OCEAN　OCEAN は渦や海流の境界を基本とした大規模な海の動きを調べるアプリケーションである。対象とする領域 (grid) を碁盤の目のような部分 grid に分割して各プロセッサに処理させている。grid は概念的に 4 次元の配列とし表現されている。

OCEAN は SPLASH-2 の application の中でも、決まった領域に対して処理を行うようなアプリケーションであり、アプリケーション中、どの時点で通信が行われれば効率がよいかが、ほぼ決定されている。それゆえロードバランスやデータの局所性を意識した処理の分割が容易に行われ、静的な分割によって良い並列実行の性能が得られると考えられる。

詳細は文献 [?] を参照のこと。

第 4 章　マルチレベルタイリング

近年、プロセッサとメモリの速度差が大きくなり、プロセッサの示す演算性能はキャッシュに依存している。そのため、プロセッサの性能を引き出すため

には、キャッシュを効果的に利用する最適化技術が不可欠である。特に、ますます大きくなるプロセッサとメモリの速度差を埋めるためには、1 次キャッシュだけではなく、メモリの各階層において、ヒット率を向上させることが重要となる。

JUMP-1 においても階層的なメモリ構成が見られることから、JUMP-1 でのマルチレベルタイリングによるキャッシュヒット率の向上の有効性の評価を行う。JUMP-1 では、クラスタメモリのアクセスで MBP-light に処理の依頼が起こると、時間がかかってしまう。よって、2 次キャッシュまでのヒット率を向上させて、クラスタメモリへのアクセスを減少させることによって、性能の向上が期待される。

評価の対象とするプログラムは SPLASH-2 のアプリケーションの LU を用いる。また、レジスタレベルまで考慮に入れると、より大きな階層構造が見られるが、本研究においてはレジスタレベルでのタイリングは行っていない。

4.1 タイリング

まず、マルチレベルタイリングにおいて基本的な処理であるタイリングについて簡単に説明する。

1 重のループを 2 重のループに分解する処理をストリップマイニングと呼び、この操作によって得られる外側のループをストリップループ、内側のループを要素ループという。タイリングとは、多重ループの各ループにストリップマイニングを施し、生じるストリップループを外側へ、要素ループを最内側へ交換する処理である。

以下に、行列積におけるタイリング例を示す。

タイリング処理後には、要素ループを互いに交換することにより、タイル内の処理順序が制御され、また、ストリップループを互いに交換することにより、タイル間の処理順序が制御される。また、これらの制御は独立である。

```
for(i=0 ; i<N ; i++)
    for(j=0 ; j<N ; j++)
        for(k=0 ; k<N ; k++)
            c[i][j] += a[i][k]*b[k][j];
```

例 1 . タイリング前

```
for(k1=0 ; k1<N ; k1+=Sk)
  for(i1=0 ; i1<N ; i1+=Si)
    for(j1=0 ; j1<N ; j1+=Sj)

      for(k=k1 ; k<Sk ; k++)
        for(i=i1 ; i<Si ; i++)
          for(j=j1 ; j<Sj ; j++)
            c[i][j] += a[i][k]*b[k][j];
```

例２．タイリング後

多重ループは、各ループに対応する次元を持つ直交空間を定義する。この空間をそのループの反復空間 (iteration space) という。タイリングは、反復空間を複数のタイルに分割する。

a[i][j] のような配列参照によって参照される配列の部分を、その配列参照のフットプリントという。タイリング後は、タイルのフットプリントの面積を縮小することによって、参照の局所性の向上を図ることができる。

4.2 マルチレベル・タイリング

タイリングは多重に行うことが可能で、タイルを階層的にサブタイルに分割することで、多重タイリングがなされる。

原理的には、各メモリ階層に対応するタイルのサイズとタイル内サブタイルの処理順序を階層の容量に合わせて決めることによって、各階層に対するヒット率を向上させることができる。

しかし、最適なタイルサイズとサブタイルの処理順序は、各階層間で互いに独立ではなく、全体として最適な解を求めることは難しい。

実際にマルチレベルタイリングを施す手法だが、図5に2次元反復空間でのマルチレベルタイリングの模式図を示す。図中、黒丸がループ本体の1回の実行を表す。

以下、タイル内の最外側のループの制御を行う変数をタイルの自由軸、内側の変数を束縛軸と呼ぶことにする。例2でいうと、k が自由軸、i、j が束縛軸ということになる。まず、内側のループから細長いタイルを作る (図3.first step)。

束縛軸の長さはこの時決定し、自由軸の長さは決めずに次のステップへ進む。この時点では自由軸の長さは該当するメモリ階層に収まる大きさであれば、いくらでも大きくとってもよく、大きくすることでヒット率は向上する。次のステップ(図3.second step)では、このサブタイルを前ステップでの束縛軸方向にまとめ、その方向に細長いタイルを作る。ここで、決めずにおいておいた自由軸の長さを決める。自由軸の長さは前段階ではいくらでも大きくとるころができたが、ここでは、その長さによってはこの階層でのミス率を悪化させてしまう。つまり、前階層とこの階層の間でミス率をうまくバランスさせる必要がある。これを階層的に繰り返すことで、2次元反復空間のマルチレベルタイリングがなされる。

3次元反復空間についても、同様な処理を行うことでマルチレベルタイリングを施すことができる。ただし、2次元反復空間においては、束縛軸は常にどちらか一方であるために次のレベルのタイリングにおいて、自由軸の選択をする必要が無く、一意に決まった。しかし、3次元になると、自由軸は2本のうちどちらか一方を選択する必要が生じるので、これについては注意が必要である。

例えば、レジスタ・レベルで考えれば、軸のとりかたによってはストアの回数を減少させることが可能である。また、この効果は、1階層おきに適応することができる。つまり、レジスタ・レベルにおいてストア回数を減少させる方向で軸を決定すると、次には2次キャッシュから主記憶へのwritebackを減らすことができる。つまり、自由軸の選択においては、書き込みの回数によってそれを選択すればよいと言える。

4.3 アプリケーションのタイリング

SPLASH-2のLUは外積形式ガウス法に1回タイリングを施したような形をしており、第3章で述べた通り、密行列をブロックに分割して、ブロックを各プロセッサに割当てて処理させることで並列実行を行っている。このブロックのサイズを適度に大きくすることでも、キャッシュヒット率を向上させることが可能である。

SPALSH-2のLUはDEFAULTの状態で16×16のブロックに密行列を分割しており、1次キャッシュへのタイリングがすでに施されているような状態になっている。また、2次キャッシュへのタイリングを考えた場合、DEFAULTの問題サイズでは2次キャッシュに完全に収まってしまう。本研究では問題サイズと

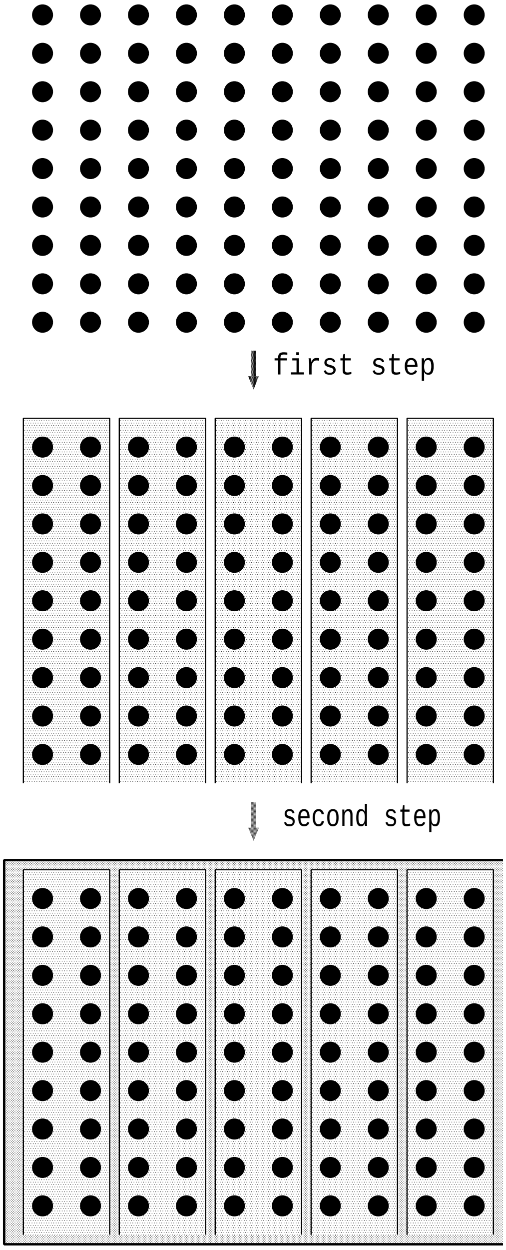

図 3: マルチレベルタイリングの模式図

ブロックサイズを変更し、2次キャッシュへのタイリングの評価が行えるように調整をした。

1次キャッシュレベルでのタイリングはブロックのサイズを1次キャッシュのサイズに合わせることでキャッシュのヒット率の向上を図った。

マルチレベル・タイリングでは、ブロック内での計算に関して、さらにタイリングを施す。第3章で述べたブロックA[k][k]内における計算など、一部のブロックでは計算順序が固定されている部分があり、前節で述べた方法ではこれらのブロックのタイリングは不可能であるが、それら以外の多数のブロックでの処理は行列積に近いような処理である。実際には、2つのブロックの一方を参照し、そのブロックの値と自分自身のブロックの値から自身のブロックの要素の値を計算するものと、3つのブロックのうち2つのブロックを参照し残りの1つのブロックの要素を決定する処理がある。どちらの処理も、共に二重のループによって制御されているので、各プロセッサに割り当てられるブロックのサイズが十分に大きい場合には、前節の方法を適用して2次元反復空間としてマルチレベルタイリングを施すこともできる。

そういったブロックについては、データのアクセスは行列積のアクセスと同様なアクセス方法をしており、計算結果がループの実行順序に依存していないので、タイリングも行列積に対する方法と同様に施すことができる。よって、タイリング後のデータアクセスについても、タイリング後の行列積と同様である。

第5章　性能評価結果

この章では、本研究で行った測定の結果を述べる。

5.1　評価環境

性能評価の対象としては、同一のCPU(SuperSPARC+)を用いたSMP、SPARCserver1000を用いている。2次キャッシュの容量はJUMP-1と同じく1MByteであるが、両者はメインメモリと2次キャッシュコントローラの実装において異なる。また、SPARCserver1000はOS稼働状態で計測しているが、性能評価において及ぼされる影響は考慮に入れる必要はほとんど無いと思われる。

SPLASH-2のPARMACSマクロについては、CREATE、WAIT_FOR_END、CLOCK以外のマクロは、JUMP-1とSPARCserver1000でまったく同じである

が、それらに関しては、SPARCserver1000ではPOSIXスレッドとgettimeofday関数を用いて実現している。よって、プロセスの生成においてはJUMP-1と異なり、SPARCserver1000ではCREATE命令が行なわれてからスレッドが生成され、それから並列プロセスが実行される。実際には、両者共に、並列プロセスが生成された段階で計測を始めているので、その結果にたいしてこの違いによる影響はほぼ無いと考えられる。

SPLASH-2のアプリケーションは上記の通り、C言語で記述されており、SPARC compiler ver3.0.1によってコンパイルしている。コンパイル時にRADIXとLUについては、SPARC V8対応の最適化オプションを選択している。OCEANには、最適化オプションを選択していない。

また、評価に用いたSPLASH-2の各アプリケーションの推奨問題サイズと、実際に評価に用いた問題サイズを下表に示す。OCEANでは、推奨サイズで実行するとクラスタメモリの容量を超えてしまうため、問題サイズを小さくした。

	推奨サイズ	問題サイズ
LU	512×512行列、16×16block	512×512行列、16×16block
RADIX	256K個の整数、基数1024	256K個の整数、基数1024
OCEAN	130×130ocean(grid)	18×18ocean(grid)

また、JUMP-1ではコヒーレンスプロトコルとして、write-invalidateとwrite-updateが提供されているが、本研究ではwrite-invalidateを選択して評価を行った。

5.2 SPLASH-2の実行結果

上記の3つのアプリケーションをJUMP-1及びSPARCserver1000で実行した時のサイクル数を図4、図5、図6のグラフに示す。グラフは縦軸にSPARCserver1000での1プロセッサの実行サイクル数を1としたときの速度比を、横軸にプロセッサ台数をとっている。

5.3 考察

これらの結果からは、JUMP-1の1クラスタではSPARCserver1000よりも性能が良いと考えられる。グラフより、全てのアプリケーションにおいてJUMP-1での実行が、SPARCserver1000での実行にかかるサイクル数よりも1割から2割程度少ないサイクル数で終了していることがわかる。

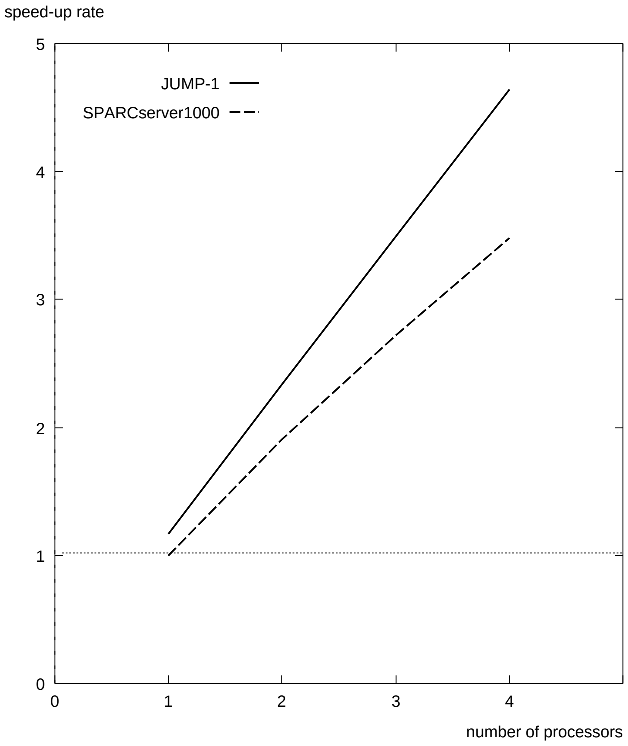

図 4: RADIX 実行時の速度比

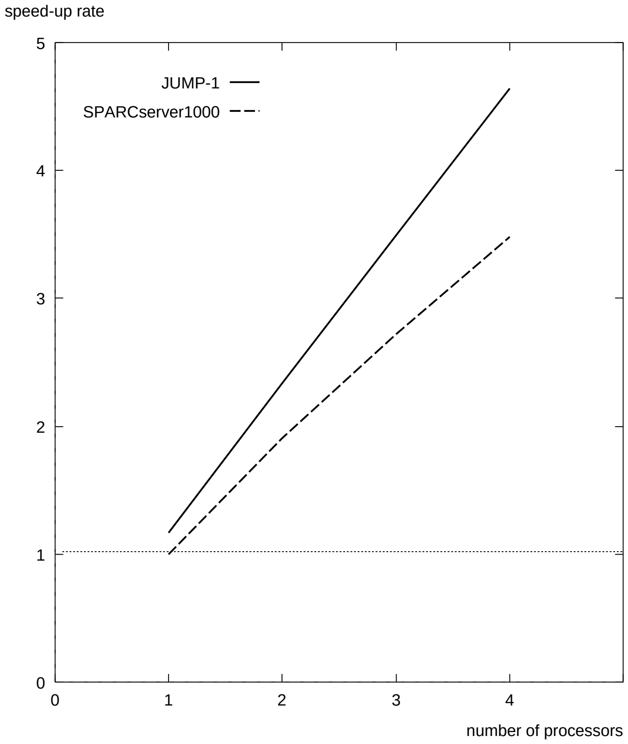

図 5: LU 実行時の速度比

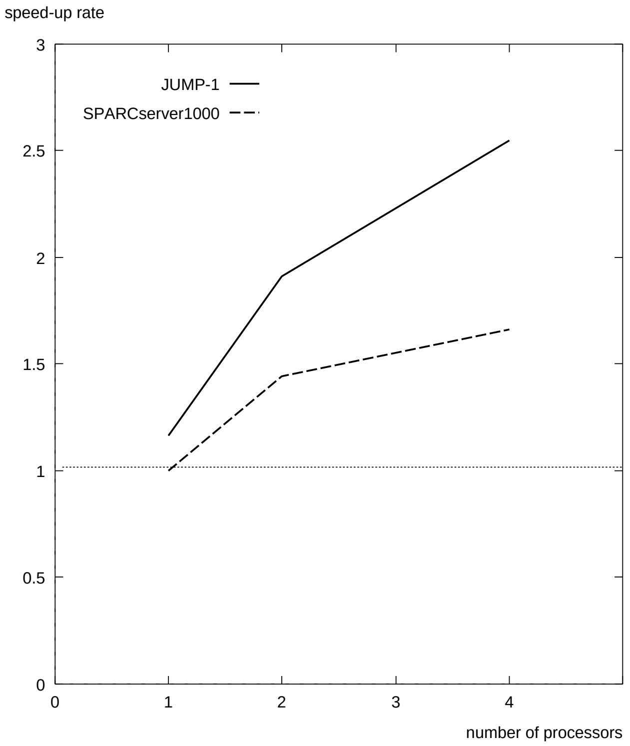

図 6: OCEAN 実行時の速度比

台数効果の点から見ると、RADIX と LU における JUMP-1 と SPARCserver1000 の測定結果はほぼ並行線を辿り、両者の台数効果は同等であると言える。OCEAN においては、JUMP-1 の方がより良い台数効果を得ている。OCEAN では、他のアプリケーションと比べて問題サイズが小さい。よって、演算量に比べて通信量が多くなり、SPARCserver1000 では、通信レイテンシを隠蔽できない為だと思われる。

つまり、演算量に対して通信量が多くないような問題では、両者の台数効果は同等であり、逆に、通信量の比率が大きいような問題では、JUMP-1 のほうが同一プロセッサを用いている SPARCserver1000 よりも、台数効果が得られると予想される。

5.4 JUMP-1 におけるタイリングの効果

図 7 に SPLASH-2 の LU にタイリングを施さないもの、1 次キャッシュのみにタイリングを施したもの、2 次キャッシュ・レベルまでタイリングを施したものの性能比を示す。現在 JUMP-1 に装備されているクラスタメモリは 1 つだけ

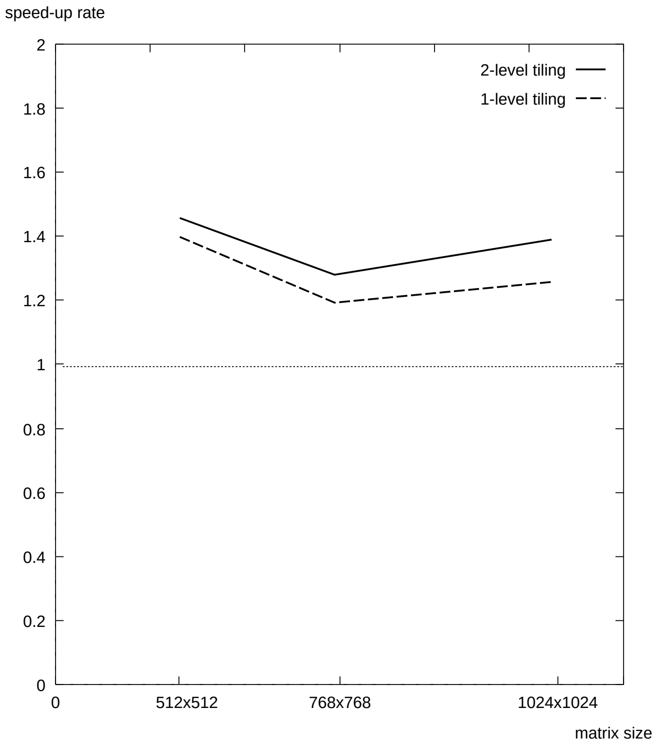

図 7: タイリングの効果

なので、問題のサイズが大きくなると実行できなくなる。2 次キャッシュへのタイリングを施そうとしているので、1 プロセッサで実行し、プロセッサに割り当てられる問題サイズを大きくして 2 次キャッシュの溢れが起こるようなサイズの問題を用いた。

グラフは縦軸にタイリングを施さない場合のサイクル数を 1 としたときの速度比、横軸に問題サイズをとった。グラフ上には、2 次キャッシュの溢れが起こる問題サイズ 512×512 から 1024×1024 のタイリングの結果のみを示した。

それぞれ 1 次キャッシュ、2 次キャッシュまでのメモリ階層にタイリングを施した場合の速度比より、タイリングによる性能向上が見られる。10 ~ 30 パーセント程度性能が向上しているのがわかる。また、両者において、512×512 から 768×768 にかけて速度比が悪化しているが、これは LU kernel のアルゴリズム上、キャッシュの汚染を回避できず、ミス・レイテンシが増加しているためだと思われる。

本研究ではメモリ容量の関係でここで用いた以上の大きなサイズの問題は扱うことができなかったが、各メモリ階層にたいしてのタイリングの効果が得ら

れており、また、より大きなサイズの問題にたいしても、同様な効果が得られるのであろう。

第6章　おわりに

本研究ではSPLASH-2により、JUMP-1の1クラスタでの性能評価を行った。SPARCserver1000と比べて、JUMP-1は1割から2割程度、速く実行することが確認された。また、演算量に比べて通信量が多くなるような問題においては、SPARCserver1000よりも台数効果を出せることがわかった。

今後は、本研究では実装できなかったSPLASH-2のアプリケーションについて、実装を行い、評価を行いたいと考えている。また、マルチクラスタでの性能評価が行えなかったが、マルチクラスタでの実装も行い、性能評価を行う予定である。

また、本研究では、タイリングの対象としたアルゴリズムの性質上、マルチレベル・タイリングの手法自体の評価としては良好な結果が得られなかったが、そういった最適化手法のJUMP-1における有効性は示すことができたと思われる。

マルチクラスタになることで、メモリの階層はより深くなる。大規模な問題に対して本研究で行った2次キャッシュまでのタイリングだけではなく、クラスタメモリのレベルまで考慮にいれたタイリングを行うことで、JUMP-1のシステム全体におけるメモリ階層の評価を行うことができるだろう。クラスタ間でのデータアクセスにかかる時間を考えると、そういった手法のJUMP-1での有効性が期待される。また、各レベルでのメモリの階層的構造を意識したアプリケーションの開発の有効性も期待できる。

謝辞

本研究の機会を与えて下さり、適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また、貴重なご助言をいただいた森眞一郎助教授、五島正裕助手に深く感謝致します。

さらに、共同研究者である秤谷雅史氏、小西将人氏をはじめとして、日頃から御鞭撻下さった京都大学工学研究科情報工学教室富田研究室の諸兄に感謝致します。

参考文献

[1] M.Goshima, et al.The Intelligent Cache Controller of a Massively Parallel Computer JUMP-1,*In Proc.IWIA '97,Intn'l Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*,pp.116-124.

[2] 井上浩明. 超並列計算機 JUMP-1 の分散共有記憶に関する研究. 修士論文. 慶應義塾大学大学院,1999.

[3] Steven Cameron Woo, et al.The SPLASH-2 Programs:Characterization and Methodological Considerations.In *Proceeding of the 22nd Annual International Symposium on Computer Architecture*,pp.24-36,June 1995.

[4] Guy E.Blelloch,et al.A Comparison of Sorting Algorithms for the Connection Machine CM-1.In *the Symposyum on Parallel Algorithms and Architectures*,pp.3-16,July 1991.

[5] Steven Cameron Woo,et al.The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors.Technical Report CSL-TR-93-593,Stanford University,December 1993.

[6] Chris Holt,et al.The Effects of Latency,Occrpancy and Bandwidth in Distributed Shared Memory Multiprocessors.Technical Report CSL-TR-95-660,Stanford University,January 1995.

[7] 津田健. メモリ ·アクセスの局所性を最適化するループ再構成法. 卒業論文. 京都大学,1999.