ハイパフォーマンス
コンピューティング 67－9
（1997. 8. 19）

Knapsack問題における共有メモリ型/分散メモリ型並列計算機の性能比較

安藤 誠[†] 田中良夫[†] 久保田和人[†]
松田元彦[†] 秋山泰[†] 佐藤三久[†]

本稿では，knapsack 問題を分枝限定法で並列に解くプログラムをベンチマークとして用いて，共有メモリ型と分散メモリ型の並列計算機の性能特性を解析・比較した．多くの場合，共有メモリ型並列計算機のほうが良い性能を示したが，探索空間内において，貪欲解から離れた幅広い範囲を走査しなければ最適解を得られないような難しい例題においては，分散メモリ型並列計算機が良い性能を示す場合もみられた．

Performance Comparison of Shared- and Distributed-Memory Parallel Computers Using Knapsack Problem

MAKOTO ANDO ,[†] YOSHIO TANAKA ,[†] KAZUTO KUBOTA ,[†]
MOTOHIKO MATSUDA ,[†] YUTAKA AKIYAMA [†] and MITSUHISA SATO[†]

In this paper, we report the performance comparison between shared- and distributed-memory parallel computers using knapsack problem as benchmarks. In some cases such that the search tree is broad, distributed-memory parallel computer shows better performance, though shared-memory parallel computer shows good performance in many cases.

1. はじめに

Knapsack 問題[1]は，選択する荷物の総重量を制約条件として荷物の価値の合計を最大化する，形式的に最も単純な構造をもつ整数計画問題である．しかし，N 個の荷物が与えられた場合，それぞれを入れるか入れないかの 2 つの状態があるので，全解探索により最適解を求める場合の探索空間は 2^N と膨大な大きさになる．実際には，分枝限定法[2]などの手法を用いて走査する空間を狭めることにより，N が数千から数万におよぶ大規模な問題を実用的に解くことができる．ただし，この分枝限定法はすべての入力値（荷物の重さ，価値，およびナップサックの収容力の組合せ）に対して万能なわけではなく，場合によってはかなり大きな空間を探索しなければならないこともある．この場合には，探索を並列計算機上で並列に実行して，全体の処理時間を短縮することが有効である．

Knapsack 問題は，木探索を行う応用問題の典型例であり，knapsack 問題を解くプログラムを用いて並列計算機の性能特性を解析しプログラムの高速化を図ることは，他の応用問題の高速化にもつながると考えられる．本稿では，knapsack 問題を分枝限定法で並列に解くプログラムをベンチマークとして用いて，共有メモリ型と分散メモリ型の並列計算機の性能の比較を試みる．そのために，並列に分枝限定法を実行するために必要な機能が何であるかを明らかにし，その機能を使用したプログラムで性能を測定・比較する．

本稿では第 2 節で，分枝限定法を用いた knapsack 問題の解法について解説する．第 3 節で，分枝限定法の並列化の方法について解説し，問題を並列に解くために必要な機能を明らかにする．第 4 節では，並列プログラムを共有メモリ型並列計算機と分散メモリ型並列計算機で実行し，それぞれの計算機の性能を実測する．第 5 節では，実測結果の解析・比較を行い，最後に関連研究，および結論と課題について述べる．

2. 分枝限定法による knapsack 問題の解法

2.1 Knapsack 問題

Knapsack 問題 (0-1 knapsack 問題[1]) は，それぞれに重さ (w_i) と価値 (p_i) をもつ N 個の荷物が与えられた時，収容力 C のナップサックに対して，w_iの合計が C を超えない範囲で，p_iの合計が最大となるような荷物の組合せを選び出す最適化問題である．こ

† 新情報処理開発機構
Real World Computing Partnership

れを数式で表現すると，以下に示す通りになる☆．

$$\text{目的関数}\quad z=\sum_{i=0}^{N-1}p_ix_i\ \rightarrow \text{最大} \tag{1}$$

$$\text{制約条件}\quad \sum_{i=0}^{N-1}w_ix_i\le C \tag{2}$$

$$x_i\in\{0,1\},\quad i=0,\cdots,N-1 \tag{3}$$

2.2 分枝操作

分枝操作とは，直接に解くことが困難な原問題 P を，より小さな部分問題 P_k に分割する操作である．具体的には，変数 x_i の値を $i=0$ から順に 0 か 1 に固定する操作にあたり，以下の式で表現される．

$$\text{目的関数}\quad z=\sum_{i\in F}p_ix_i+\sum_{i\in S^1}p_i\ \rightarrow \text{最大} \tag{4}$$

$$\text{制約条件}\quad \sum_{i\in F}w_ix_i\le C-\sum_{i\in S^1}w_i \tag{5}$$

$$S^1=\{i|P_k\text{で}\ x_i\text{の値が}\ 1\text{に固定}\} \tag{6}$$

$$F=\{i|P_k\text{で}\ x_i\text{は固定されていない}\} \tag{7}$$

この過程は図 1 の列挙木で表現できる．深さ方向(左端の数字)は着目している荷物の ID を示しており，木はそれぞれの荷物を「入れる」(左下向きの矢印)か「入れない」(右下向き)かで分岐する 2 分木となる．

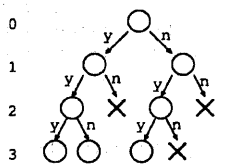

図 1 Knapsack 問題における分枝・限定操作

なお，図 1 中の「×」で示された部分は，後述する限定操作によって枝刈りされた部分で，この部分についてはこれ以上分枝操作は行われない．

2.3 限定操作(枝刈り)

限定操作では，まず分枝操作で生成された各部分問題について目的関数の値 z の上界値と下界値を算出する．それらの値から，その部分問題が最適解を生成する見込みがなくなったと判断できる時には，その部分問題に対してこれ以上分枝操作を行わないようにする．以下では，上界値と下界値の算出方法，および分枝操作停止の判断方法について説明する．

2.3.1 貪欲法による下界値の算出

ここで，効率 γ_i を p_i/w_i で定義し，さらに，与えられたすべての荷物が $\gamma_0\ge\gamma_1\ge\cdots\ge\gamma_{N-1}$ でソートされていると仮定する．この時，z の値を大きくするためには，単位重量当たりの価値を示す γ_i の大きい荷物を優先して入れるほうが良い．

貪欲法 (greedy algorithm) は，現在着目している部分問題で固定されていない x_i について，γ_i の大きな順に (つまり添字 i の小さい順に) w_i の合計が C を超えない範囲で 1 に固定する手法で，これによりきわめて良い目的関数の下界値を簡単に生成できる．

2.3.2 連続緩和法による上界値の算出

ここでも，与えられた荷物が $\gamma_0\ge\gamma_1\ge\cdots\ge\gamma_{N-1}$ でソートされているものと仮定する．現在着目している部分問題において，前述の貪欲法で 1 に固定された x_i の最大添字より 1 大きい添字を q とする．x_q を，$x_q\in\{0,1\}$(0 か 1 の離散値) から $0\le x_q\le 1$(0 から 1 までの連続値) に緩和することによって得られる z の値を考える (図 2)．この時，この部分問題の最適値は，x_q を連続緩和 (linear programming relaxation) して得られる値を超えることはなく，連続緩和によって得られる目的関数の値を，その部分問題の目的関数の上界値として使うことができる．

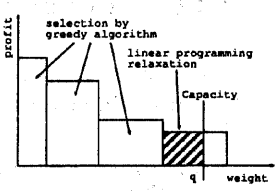

図 2 連続緩和法

2.3.3 上界値，下界値による枝刈り

探索を行うプログラムの中に Global Low (これまでに算出された下界値の最大のもの) と称する値を格納するための領域を確保する．分枝操作を進めるに当たり，それぞれの部分問題で目的関数の値 z の上限値と下界値を前述の方法で算出する．

まず，現在着目している部分問題の z の下界値と Global Low を比較し，もし Global Low より下界値が大きければ，Global Low を下界値で更新する．

次に，現在着目している部分問題の上界値と Global Low を比較し，もし Global Low よりその部分問題の上界値が小さければ，その部分問題は最適解を得る見込みがないので，分枝操作を停止できる．このようにして，走査する空間を縮小することができる．

2.4 釘付けテストによる問題規模の縮小

本稿のプログラムでは，分枝操作と限定操作に加え，以下の釘付けテスト[2]も採用した．

まず，2.3.2 で与えられる q を用いて，μ_i を定義する．

$$\mu_i=\begin{cases}p_i-\gamma_qw_i, & i=0,\cdots,q\\ -p_i+\gamma_qw_i, & i\ge q+1\end{cases} \tag{8}$$

ここで，部分問題の暫定解を $\bar{z}$ とし，連続緩和して得られる上界を z_c とすると，

(1) $\mu_i\ge z_c-\bar{z}$ かつ $i<q$ なら，暫定解よりも良い実行可能解は $x_i=1$ を満たす．

(2) $\mu_i\ge z_c-\bar{z}$ かつ $i>q$ なら，暫定解よりも良い

☆ ここに与えられる定数は，
(1) すべての i に対して $w_i\le C$ かつ $\sum_{i=0}^{N-1}w_i>C$
(2) p_i,w_i,C はすべて正の整数
をともに満たすものとする．

実行可能解は $x_i = 0$ を満たす.

上記の条件を満たす変数は 0 または 1 に固定され，問題の規模を縮小させることが可能となる.

3. 分枝限定法の並列化

3.1 グローバルリストによる部分問題の管理

Knapsack 問題の原問題の探索空間の広さは，前述した通り 2^N である．しかし，分枝限定法の利用で多くの場合，図 1 で示された列挙木の，比較的左側の(前半の荷物を優先的に入れる貪欲解に近い) 縦に細長い部分木だけを集中して探索することが予想される.

このことから，あらかじめ探索空間 (部分問題) を利用可能なスレッド/プロセスの数に分割しておいて，それらを分担して処理するというやり方では，よほど上手に部分問題の再分配を行わない限り，プロセス間の仕事量の不均衡が顕著になる.

そこで，本稿のプログラムでは，文献[3]でも紹介されているように，スレッド/プロセス間の可搬性を考慮して，部分問題を 1 つのセルで表現した (図 3). このセルには，その部分問題における固定された x_i の組や，現在までの p_i の合計，ナップサックの残り収容力などの情報が格納される．さらに，それらをつなげたグローバルなリスト (linked list) を 1 本用意した．問題を解くスレッド/プロセスは適宜グローバルリストにアクセスし，リストの先頭からセルを 1 つ受けとる．セルを受けとったプロセスは，適宜分枝操作と限定操作を行って部分問題を生成し (本稿の実装では，4096 回の分枝操作をまとめて行う仕様にした)，それらをグローバルリストに戻す．これらの操作は，リストが空になるまで続けられる.

この方法は，各スレッド/プロセスが分枝・限定操作をグローバルリストに対して直接行う方法と，スレッド/プロセスがそれぞれ独自のローカルキューだけをもつ方式の中間にあたり，共有空間の排他制御 (共有メモリ) やプロセス間の通信 (分散メモリ) にかかるオーバヘッドを低減しつつ，かなり均一な負荷分散が可能になると期待できる.

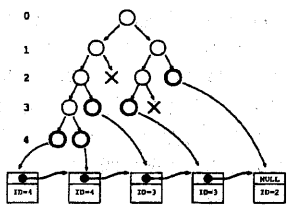

図 3 部分問題のセル

なお，部分問題をグローバルリストに戻す際には，列挙木での位置がより左にあるものが，常にリストのより先頭側になるように挿入する.

3.2 共有メモリ型計算機でのプログラミング

Solaris thread を使った．前述のグローバルリストと Global Low を共有メモリ空間に置き，各々のスレッドは，排他制御をしながらそれらにアクセスするようにした.

3.3 分散メモリ型計算機でのプログラミング

通信ライブラリとして MPI を用いた．文献[4]と同じく，マスター—スレーブ型の構成をとるようにした．マスタは常に 1 つ存在し，グローバルリストの管理に専念する．スレーブは，マスタに部分問題を要求することによって新しい部分問題を得る.

Global Low もマスタが管理する．スレーブは，マスタに部分問題要求の通信をする際に，自プロセス内のローカルな Global Low も同時に送信する．マスタは，スレーブからの Global Low と自分の Global Low を比較し，適宜更新する．またマスタは，スレーブへの部分問題配布の際に，新しい Global Low も送信する.

4. 実測結果

4.1 使用した計算機

表 1 に実測に使用した計算機を示す．以下では，SPARCcenter 2000E を SC，SR2201 を SR と略記することにする.

表 1 使用した計算機の概要

	共有メモリ型	分散メモリ型
名称	SUN SPARCcenter 2000E	Hitachi SR2201
CPU	SuperSPARC II	PA-RISC 1.1 + PVP-SW
周波数	85MHz	150MHz
CPU 数	20	256
OS	Solaris 2.4	HI-UX/MPP
library	Solaris thread	HI-UX/MPP MPI

4.2 PSC データに対する実測

上記の仕様で実装されたプログラムで，PSC97(1997 Parallel Software Contest)[5]で出題されたデータ 9 種を実測した．まず，例題 PSC1～PSC5 については，逐次版を用いて 100μsec～700μsec 程度で実行できるので，並列版による実測は行わなかった．これらのデータにおける効率 γ_i の変動係数☆は 2.3～5.3 でばらつきが大きく，充分な枝刈りと変数の釘付けが可能である.

次に例題 PSC6～PSC9 については，並列版で実測した速度向上比を図 4～7 に示す．PSC6 は，すべての荷物について γ_i が一定であり (変動係数は 0)，限定操作が一切行えないという特徴がある．また，PSC7～9 の変動係数は 0.1 程度である．荷物の個数 N は，PSC6 は 42，PSC9 は 500，その他は 100 である．逐次版による実行時間は，表 2 に示した.

4.3 ランダムデータに対する実測

PSC の例題に対する実測結果を見ると，全体の傾

☆ 効率 γ_i の標準偏差 σ を平均値 μ で割ったもの.

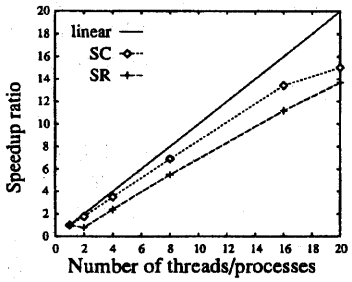

図 4 例題 PSC6 での速度向上比

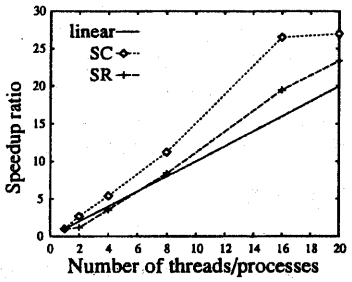

図 5 例題 PSC7 での速度向上比

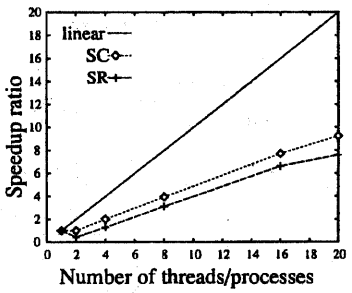

図 6 例題 PSC8 での速度向上比

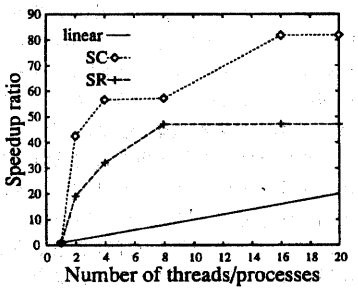

図 7 例題 PSC9 での速度向上比

向として SC のほうがより良い性能を示している．しかし，PSC の例題だけでは試行数が少ないので，変動係数$\simeq 0.1$，$N = 100$ の例題をランダムに 100 個作成した．このうち，SC での逐次の実行時間が 100sec 以上の例題 10 個について，並列版での実行時間を測定した．図 8 は，例題 10 個の実行時間の平均をとり，

表 2 逐次版による実行時間 (秒)

	PSC6	PSC7	PSC8	PSC9
SC	255.0	135.0	185.0	171.0
SR	302.0	117.0	159.0	96.0

その速度向上比を示したものである．

その結果，すべての実測値で SC が良い性能を示すわけではなく，図 9 のように，SR のほうが良い性能を示している例 (以後，ORG1 と呼ぶ) も見受けられた．以下では，この性能の違いについて考察する．

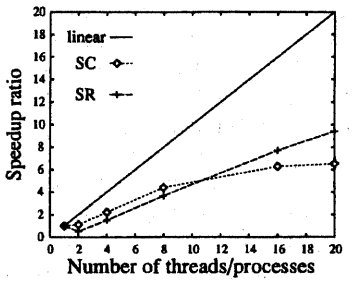

図 8 例題 10 個の平均実行時間による速度向上比

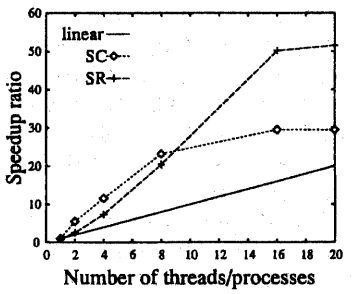

図 9 例題 ORG1 での速度向上比

5. 実測結果の考察

5.1 探索空間の性質の相違

一般に，木探索問題において走査される空間の広さは，入力値 (knapsack 問題では，荷物の重さや価値，ナップサックの収容力) に依存するだけでなく，どのような順序で枝を開くかによって著しく変動する．本稿のプログラムは「開くことのできる枝のうち，最も左に位置するものから開いていく」というアルゴリズムを採用しているので，その順序で開くことを前提に走査空間について考察する．

図 10 と図 11 は，それぞれ PSC9 と ORG1 の最適解を示したものである (黒い四角は「入れる」，白は「入れない」を表す)．両者を比較すると，ORG1 では早い段階 (57 番目の荷物) で「入れない」が出現しているので，根に近い部分で最適解を生成する分岐が起

こっている (PSC9 は，先頭から 80%を過ぎたところで初めて「入れない」が出現している)．このことから，ORG1 における走査空間は PSC9 に比べて，より幅の広い木であったことが予想される．よって，分枝・限定操作が逐次化されず，並列に処理が可能と考えられる．この場合には，分散メモリ型が良い性能を示している．逆に，PSC9 における走査空間は縦に細長く，分枝・限定操作が逐次化されると考えられる．この場合には，共有メモリ型が良い性能を示している．

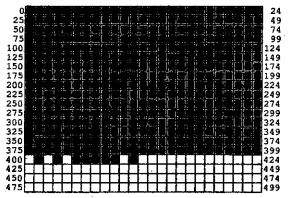

図 10　PSC9 の最適解

図 11　ORG1 の最適解

5.2　稼働率による考察

次に，プログラムの実行時間の内訳を解析する．

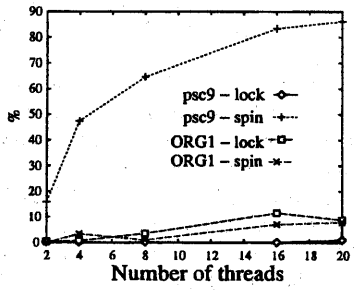

図 12　共有領域へのアクセス時間の全体に占める割合 (SC)

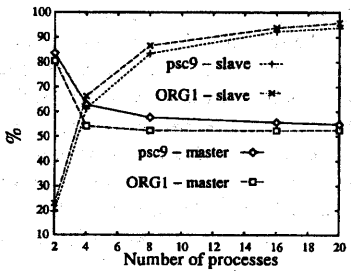

図 13　通信待ち時間の全体に占める割合 (SR)

図 12 は SC(共有メモリ) で計測したものである．“lock” は mutex_lock/unlock に挟まれた部分の実行時間が総実行時間に占める割合を示したもので，これはグローバルリストでのセルの操作 (ポインタ付け替え) の時間的割合を示している．また “spin” は，スピンウェイトしながら変数の値を監視している時間で，具体的には，空のグローバルリストに誰かがセルを追加するのを待つ時間である．また，図 13 は SR(分散メモリ) で計測したもので，マスタ/スレーブそれぞれで，通信時間の占める割合を示したものである．

SC(共有メモリ) においては，PSC9 で全実行時間のうちスピンウェイトが占める割合がかなり高くなっており (図 12)，グローバルリストが空になっている時間が長いことが分かる．これは，走査する空間が縦に細長くて，少数のスレッドが部分問題を抱えこんでしまっている状態であると考えられ，5.1 での予想を裏付ける結果となっている．逆に，ORG1 ではスピンウェイトの時間がほとんどなく，すべてのスレッドが並列に分枝・限定処理を進められたと考えられる．

また，SR(分散メモリ) において PSC9 では 8 プロセスの時点で (図 7)，ORG1 では 16 プロセスの時点で (図 9)，速度向上の曲線が頭打ちになってしまっていた．図 13 によると，両データの場合とも，8 プロセスの時点でマスタの通信時間の割合が下げ止まっており，スレーブからの要求がマスタの処理能力の限界まで達している可能性が考えられる．従って，SR についてはこれ以上プロセス数を増やしても，現在の方式をとる限りでは速度の向上は見られないと考えられる．

5.3　分枝回数による考察

図 14 は，各スレッド/プロセスが処理した分枝操作の回数の最大値と最小値の比を示したものである．

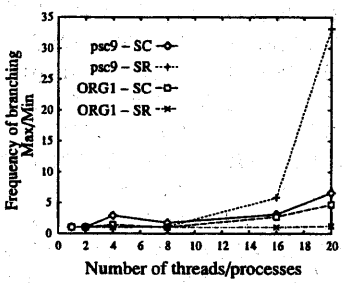

図 14　分枝数のばらつき

この図において，PSC9 を SR で実行した時の分枝処理数にプロセス間でのばらつきが見られる．ばらつきを押えるには，各プロセスにおいてまとめて行う分枝・限定の操作数を減らして (本実装では 4096 回)，グローバルリストへ部分問題を戻す頻度を高めればよいのであるが，スレーブの処理における通信時間が占める割合はすでに 90%を超えており (図 13)，これ以上通信頻度を増やすのは得策でないことが分かる．

5.4 Global Low の更新頻度の影響

Global Low は枝刈りの目安となる値であり，できる限り真の最適値に近い大きい値をもつことが効果的な枝刈りにつながる．しかし，必ずしも最新の最大値に維持されていなくても，最適解は正しく得られる．

SR(分散メモリ) の実測に用いたプログラムでは，各スレーブの内部で分枝・限定操作を 4096 回行う毎にマスタと通信し，Global Low を更新するようにした．このため，1 回の分枝・限定操作の度に Global Low を更新する SC(共有メモリ) 用のプログラムに比べ，枝刈りがうまくされないのではないかと推測できる．特にプロセス数が増えた時に，あるプロセスで生成された良い Global Low が，他のプロセスに配布されるまでに相当な時間がかかるのではないかと考えられる．

表 3 は，SR(分散メモリ) 上で 20 プロセスで問題を解く際に，Global Low の更新間隔 (1 回の通信あたりに行う分枝操作数) を変えると，総分枝数がどう変わるかを示したものである．例題 PSC9 では，Global Low の更新間隔を広げると，全体の分枝操作数も増えているが，逆に，例題 ORG1 ではわずかながら減少している．SR では，プロセス数が増えるに従って実質的な更新間隔が増えるため，開かなければならない枝の数が増えることがある．このことが，PSC9 での台数効果の頭打ちの原因になったとも考えられる．

表 3 Global Low の更新間隔と総分枝数の関係 (SR)

	8	64	512	4096
PSC9	2484948	2478197	3128356	3372438
ORG1	21407077	21229144	21635217	18440834

6. 関連研究

Knapsack 問題を実用的な時間で解くための手法として，分枝限定法と並んでよく利用されるのが動的計画法である．動的計画法にもとづき knapsack 問題を並列に解く試みが，文献[6]で行われている．

また，今回 knapsack 問題の例題に使用したデータは，PSC で提供されたものと，独自に乱数で作成したものを用いたが，本来は共通のデータセットがあれば理想である．これについては，(単純)knapsack 問題については見つけられなかったが，複数 knapsack 問題については，ベンチマークとして使用できるデータが WWW 上[7)]～[10)]に紹介されている．

7. 結論および課題

本稿では，knapsack 問題を分枝限定法を用いて並列に解くプログラムを用いて，共有メモリ型と分散メモリ型の並列計算機の性能を比較した．

多くの場合，knapsack 問題の走査空間は，木の左側に偏った縦に細長い空間であることが多い．その場合には，共有メモリ型並列計算機が良い性能を出すことが分かった．しかし，走査すべき空間が幅方向に広い例題に対しては，分散メモリ型並列計算機が良い性能を示すことがあることも確認された．

また，枝刈りの目安として使用される共有変数 Global Low については，本稿の分散メモリ型用のプログラムのように更新頻度を低めると，枝刈りがうまく進まず余計な分枝操作をすることになり，結果として台数効果が現れない場合があることが分かった．

今後は，共有メモリ型/分散メモリ型それぞれについて，複数の計算機で実測を行い，要素技術 (mutex_lock の性能や，内部ネットワークの性能など) の違いによる性能特性について検討する予定である．

8. 謝　辞

本稿のプログラムを作成するにあたり，もとになるプログラムを提供していただいた，電子技術総合研究所の児玉祐悦氏に感謝します．また本研究に対して議論に参加いただき多くのアドバイスをいただいた，新情報処理開発機構 島田潤一研究所長，並列分散システムソフトウェアつくば研究室・並列分散システムアーキテクチャつくば研究室・並列応用つくば研究室の皆様に感謝します．

参考文献

1) S. Martello, P. Toth, "KNAPSACK PROBLEMS – Algorithms and Computer Implementations", Wiley-Interscience Series in Discrete Mathematics and Optimization (1989).
2) 今野浩, 鈴木久敏, "整数計画法と組合せ最適化", 日科技連 (1982).
3) A. Pritchard, "A Template for Sequential and Parallel Branch-and-Bound Tree Search", `http://pete.cs.caltech.edu/Archetype/Papers/BnB/report.html` (1994).
4) 吉松敬仁郎, 安浦寛人, "並列部分問題探索法による組合せ最適化問題の並列処理", 情報処理学会ハイパフォーマンスコンピューティング研究会報告, Vol. 96, No. 22, pp. 49–54 (Mar. 1996).
5) `http://proton.is.s.u-tokyo.ac.jp/psc97/`
6) A. Goldman, D. Trystram, "An efficient parallel algorithm for solving the knapsack problem on the hypercube", *Proc. 11th International Parallel Processing Symposium* (1997).
7) `http://www.gsia.cmu.edu/afs/andrew/gsia/Srinivas/milind/mknapinfo.txt`
8) `http://www.gsia.cmu.edu/afs/andrew/gsia/Srinivas/milind/mknap1.txt`
9) `http://www.gsia.cmu.edu/afs/andrew/gsia/Srinivas/milind/mknap2.txt`
10) `http://mscmga.ms.ic.ac.uk/jeb/orlib/mknapinfo.html`