入れ子関数を利用した動的負荷分散

八 杉 昌 宏[†,††] 田 畑 悠 介[†]
小 宮 常 康[†] 湯 淺 太 一[†]

探索問題などの樹状再帰的な細粒度並列計算においてプロセッサ間でタスクを授受して効率よく負荷分散を行うには，実行中のタスクから lazy に並列実行可能なタスクを分割・生成し抜き取れるようにした遅延タスク生成などの負荷分散方式が有効である．仕事（ work ）の転送可能なひとまとめであるタスクの抜き出しはできるだけ呼出しの根元から行うことで，ほぼ最小限のタスクの生成・授受で各プロセッサに仕事が充分に割り当てられる．しかし，そのためには関数の呼出し中に通常はアクセスできない呼出し元の変数にアクセスできる必要がある．本発表では，GNU C コンパイラなどが C 言語の拡張機能として提供する入れ子関数を利用して，呼出し元の変数にアクセスすることで，遅延タスク生成に基づく負荷分散を行うプログラムが記述できることを示す．また，バックトラックに相当する動作の記述も可能であることを示す．ただし本記述方式では，オリジナルの遅延タスク生成とは異なり，タスクは，継続を処理するものではなく，関数中に示された並列実行可能部分を処理するものとする．本発表では，共有メモリ型並列計算機上で，提案する記述方式のプログラムを実行した結果についても述べる．

Dynamic Load Balancing by Using Nested Functions

Masahiro Yasugi,[†,††] Yusuke Tabata,[†] Tsuneyasu Komiya[†]
and Taiichi Yuasa[†]

To realize efficient dynamic load balancing by transferring tasks among processors in tree-recursive fine-grained parallel computing such as searching problems, load balancing schemes which lazily create and extract a task by dividing the present running task, such as *Lazy Task Creation* (LTC), are effective. By extracting a task, which is a piece of work transferable to other processors, around the *root* of the invocation tree as much as possible, each processor can have sufficient amount of work with the almost minimum number of task transfers. However, it requires a way to access to caller's variables which are usually not accessible during an invocation of a function. In this presentation, we show that we can describe a program with LTC-based load balancing where caller's variables are accessed by using *nested functions* provided as an extension to C by the GNU C compiler. We also show that we can describe a behavior corresponding to *backtracking.* Although a task is created to process a *continuation* in the original LTC, in our scheme, a task is created to process the part for parallel execution specified in a function. In this presentation, we also show the result of the execution of the proposed-style program on a shared-memory parallel computer.

1. は じ め に

探索問題などの樹状再帰的な細粒度並列計算においてプロセッサ間でタスクを授受して効率よく負荷分散を行うには，実行中のタスクから lazy に並列実行可能なタスクを分割・生成し抜き取れるようにした遅延タスク生成[8] などの負荷分散方式が有効である．

ある量の仕事 (work) を転送可能な 1 個の形にしたものをタスク (task) と呼ぶ．タスクの分割・抜き出しはできるだけ呼出しの根元から行うことで，ほぼ最小限のタスクの生成・授受で各プロセッサに仕事が充分に割り当てられる．しかし，そのためには関数の呼出し中に通常はアクセスできない呼出し元の変数にアクセスできる必要がある．

本論文では，GNU C コンパイラなどが C 言語の拡張機能として提供する入れ子関数[1),10)] を利用して，呼出し元の変数にアクセスすることで，遅延タスク生成に基づく負荷分散を行うプログラムが記述できるこ

† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
†† 科学技術振興事業団さきがけ研究 21「情報と知」領域グループ
"Information and Human Activity", PRESTO, Japan Science and Technology Corporation(JST)

とを示す．また，バックトラックに相当する動作の記述も可能であることを示す．ただし本記述方式では，オリジナルの遅延タスク生成とは異なり，タスクは，継続を処理するものではなく，関数中に示された並列実行可能部分（スレッド）を処理するものとする．本発表では，共有メモリ型並列計算機上で，提案する記述方式のプログラムを実行した結果についても述べる．

論文の構成は以下の通りである．2 章では，論文の背景として，我々が進めている並列処理のための協調機能を持つ拡張 C 言語の研究と，GCC での入れ子関数について述べる．3 章では，本論文で例題として取り上げる，単純な樹状再帰的計算である Fibonacci 数の計算と，より複雑でバックトラックが望まれるペントミノ全解探索問題について述べる．4 章では，関連研究として動的負荷分散を行う言語処理系についてその高水準マルチスレッド言語，目的言語，実装手法について述べる．5 章では，Fibonacci 数の計算を動的負荷分散するために，提案方式ではどのような目的言語のプログラムにするのかについて述べる．6 章では，ペントミノ全解探索の動的負荷分散について同様に述べる．7 章では記述したプログラムの性能評価を行う．8 章では今後の課題として，分散メモリ型上での実装・評価，高水準言語上での構文，拡張 C 言語の実装手法について述べる．

2. 背　　景

2.1 並列処理のための協調機能を持つ拡張 C 言語

計算機の高性能化のために並列計算機は魅力的である．並列アーキテクチャは，共有メモリ型アーキテクチャ，分散メモリ型アーキテクチャなど様々であり，その違いを吸収しつつ信頼性・再利用性・実行効率の高いソフトウェアを開発するには，並列処理のための高水準プログラミング言語が必要となる．

高水準言語コンパイラの実装には言語階層を用いると便利である．特に C 言語を経由することで，低レベルの最適化やレジスタ割り付け，アセンブリコード生成などの処理を C コンパイラに担当させることができる．このように C 言語を実装用言語として用いることで，処理系の共通に利用できる部分を増やし，処理系実装の労力を減らすことができる．

しかしながら，C 言語は，本来逐次処理用の言語であるため，並列処理のための実装用言語として用いるには記述性・実行効率ともに問題がある．C 言語の仕様/処理系では他のプロセッサというものの存在が基本的にあまり考慮されていないため，自プロセッサでの決められた処理を集中して進めるのは得意であるが，他のプロセッサとの協調はあまり得意ではない．他のプロセッサとの協調のためには，プロセッサ間での通信・同期の機能，他のプロセッサからの非同期の依頼に応える機能，自プロセッサでの処理の内容や順序を固定化せず他のプロセッサに合わせて調整する機能が求められる．そのような機能のために，通常は通信ライブラリやスレッドライブラリなどの並列ライブラリを用いる必要がある．通信ライブラリは，分散メモリ環境における通信・同期などに用いられるが，高速な並列アーキテクチャを生かしきれないという問題がある．また，スレッドライブラリは，同期の機能や，自プロセッサで進められる処理の流れ（スレッド）を複数持ち，他のプロセッサに合わせて処理順序を変える機能などを持つが（データの通信は共有メモリを利用），同期の度にライブラリ関数を呼び出すオーバヘッドや，スレッドを生成するオーバヘッドが大きく細粒度スレッドの実装には適さない．

そこで，本研究では，並列処理に適した実装用言語として並行協調拡張 C 言語 XC-cube を設計・開発している．また，どのような設計が望ましいか，またどのような記述性や性能の向上が可能となるのかを研究している．現在の基本設計方針は以下のようになっている：各プロセッサのためのコードを記述するための言語として用いる．C 言語と同じく，汎用言語，低水準言語として柔軟性を保ちつつ，並列処理に適するように拡張し，アセンブリコードを直接生成しなくては実装が困難だった高水準言語のための実装用言語としても利用できるようにする．そのため並列アーキテクチャの通信・同期機能を直接サポートする[14]．また，他のプロセッサからの要求と自プロセッサの処理の多重実行に必要な機能をサポートする．

自プロセッサでの処理の内容や順序を固定化せず他のプロセッサに合わせて調整する機能を提供するために，我々は，次節で述べるように入れ子関数のクロージャを利用することを提唱している．また，クロージャ生成コストを抑えた拡張 C 言語の実装を進めている．

2.2 入れ子関数のクロージャの利用

一般に，次に実行できることが複数あったとしても，どれから先にやるのかを決めておくこくことで，高速な実行が可能となる．つまり，得意なやり方で集中して実行できるように，実行手順を固定化しておきたい．実行手順の固定化は，一つのプロセッサのみで計算をするのであれば問題ない．最終結果をさえ求められればよいので，実行手順を実行中に柔軟に変更可能としても得るものはなく，逆にいえば，実行手順を固定化しても失うものはないためである．

ところが，並列処理のために，複数のプロセッサ間で協調をする場合は，事情が異なる．他のプロセッサとの同期によって実行手順を変更したり，他のプロセッサから協力の申し出に対して仕事の一部を分け与えたりできることなどが重要となるためである．そのために，ここでしたいことは「普段は固定化した通りの実行手順で高速に実行するが，いざというときには実行手順を見直すための仕掛けを準備する」というものである．例えば公共工事の「見直し」のように，一度計画通りはじめた仕事であっても完全には手順を固定化せず周りと協調して途中で手順を見直せるようにするものである．そのように，元々設定した目標を中途変更できるようなプログラムを記述できるような言語設計を行う．

このため，各関数には，半固定化された高速な実行手順以外に「普段は使わないが，いざというときのための関数」を含ませておく．この関数中の関数（入れ子関数）が呼び出されたときに「元関数がどういうふうにやりかけであったか」が分かるようになっていればよい．これで，普段は半固定された手順で高速に実行することを可能とするが，いざというときは，この入れ子関数を呼び出すことで，やりかけの関数でさえ見直しを可能とできる．

入れ子関数を利用することで，従来の C 言語では記述できなかった「他のプロセッサなど外部と協調した自プロセッサでの実行順序の調整」が可能となる理由は次のようなものである．従来の C 言語では，関数呼出しを行っている間，同じプロセッサからであっても呼出し元に眠っている変数にアクセスすることはできない．このため，一度関数呼出しを行ってしまうと，呼出し中の間ずっと，呼出し元に戻った後の残りの処理について外部と協調した調整を行うことができない．入れ子関数は関数内で入れ子に定義される関数であり，定義により，入れ子関数本体とその定義時の環境を合わせて組としたクロージャ（closure）が生成される．入れ子関数の関数ポインタで生成したクロージャを利用することで呼出し元にアクセスする手段が提供できる．

入れ子関数を利用することで，プログラムに並行性・協調性を追加することが容易となり，本論文で述べる動的負荷分散の他，マルチスレッド[18]，マイグレーション・チェックポインティング，例外処理，GC でのスタックスキャンなども実現できる．

2.3 GCC での入れ子関数の仕様と実装

GCC は拡張機能として入れ子関数の機能を提供している[1),10)]．GCC ではローカル変数の宣言が可能な場所での定義が可能であり，その名前はローカル変数と同様に定義されるブロック内で有効である．入れ子関数は定義が行なわれた時点の環境に従って外側の関数の仮引数やブロックのローカル変数やラベル，他の入れ子関数にアクセスすることもできる．C 言語の場合は入れ子関数のポインタを取得して関数呼出しの引数としたりグローバル変数に代入するなどの手段によって入れ子関数を定義した関数（環境）の外で入れ子関数を呼び出すことができる．ただし，ブロック内のローカル変数へのポインタをそのブロックの実行完了後に使用してはならないように，ブロック内の入れ子関数もブロックの実行完了後に使用してはならない．

GCC の入れ子関数の実装ではスタティックリンクを用いている．入れ子関数を定義した関数が，入れ子関数名の有効範囲で入れ子関数を呼び出す場合には，スタティックリンクを特別な追加の引数として，入れ子関数本体の処理を行うコード（命令列）を呼び出せばよい．

関数ポインタの取得において通常の関数の関数ポインタの実体は，その関数本体の処理を行うコードの先頭のアドレスと考えて良いが☆，入れ子関数の関数ポインタの実体を単なるコードのアドレスとするのは簡単ではない．入れ子関数の関数ポインタにより，入れ子関数本体と定義された時の環境の組であるクロージャが利用できなくてはならず，単なるコードのアドレスでこの組を表す必要がある☆☆．

GCC ではこの問題をトランポリン[1)]と呼ばれる機構を使うことによって解決している．トランポリンとは入れ子関数を定義した際のフレームポインタの値をスタティックリンク用のレジスタ（特別な追加の引数）にセットしてから入れ子関数本体のコードのアドレスへジャンプする数命令の短いコードのことであり，スタック上に動的に生成される．そのスタック上のトランポリンのコードのアドレスを関数ポインタとすることによって入れ子関数のポインタでクロージャを利用することができ，ほとんどのアーキテクチャにおいて動作する．この手法によって GCC は入れ子関数を実現しているが，スタック上にコードを動的に生成することや，アーキテクチャによってはプロセッサの持つ命令キャッシュを明示的にフラッシュする必要があることなどのオーバヘッドが発生する．

☆ ただし，評価で述べるように，PowerPC では通常の関数ポインタであってもコードと環境の組になっている．

☆☆ 通常の関数の場合も定義時の環境でグローバル変数等にアクセスできるが，その環境はコンパイルとリンクの際にコードに埋めこまれる．

3. 例　　題

樹状再帰的で不規則な細粒度並列計算として，本論文でとりあげる例題について説明する．

一つは，定番であり，説明の必要はないと思われるが，

$\mathrm{fib}(1) = \mathrm{fib}(2) = 1$

$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$ for $n > 2$

で定義される Fibonacci 数列 $\mathrm{fib}(n)$ の第 n 項を再帰的な計算で求めるというものである．実用上の意味はないが，樹状再帰的で不規則な細粒度並列計算を行うものであり，また各関数の純粋な仕事がほとんどないため呼出しや並列処理に関するオーバヘッドなどの測定に向いている．

もう一つはペントミノパズルとして知られている探索問題をとりあげる．ペントミノとは 1×1 の正方形 5 つを辺同士で連結した 12 種類のピースのことで，例えば，十字型やコ字型のピースがある．ペントミノパズルは，このピースを 6×10 の長方形のボードに全て納める解を見つけるという問題である．ここでは，すべての解を求めるペントミノ全解探索を例題として用いる．一般に，この種のパズルはバックトラック法を用いて解く．バックトラック法は，ピースがボードにうまく置けるかどうかを判定し，うまく置けるようなら，配置データを書き換えて，さらに再帰的に探索を続けるが，その置き方についての探索を終えたら，配置データを書き戻してピースを取り除き元に戻す．しかし，このやり方は，配置データを一つのプロセッサのみが変更／バックトラックすることを前提としており，並列処理でこれを行うのは難しい．多くの並列処理では，より深く探索を続ける度に配置データをコピーする必要があった．配置データコピーについては差分のみを保持するなどコピーを仮想的に行うことでそのコストを小さくできるが，そうすると逆に配置データへのアクセスが遅くなるなどの問題があった．

4. 関 連 研 究

4.1　高水準マルチスレッド言語

高水準マルチスレッド言語としては，Scheme 言語に `future` 構文を導入した Multilisp 言語[4]，C 言語に `spawn` 構文などのスレッド機能を導入した Cilk 言語[3]，Java 言語に `par` 構文などのスレッド機能を導入した OPA 言語[15] などがある．

4.2　目的 (中間) 言語

目的言語として直接アセンブリ言語を用いるコンパイラも考えられ，オリジナルの遅延タスク生成[8] で用いられている．Cilk や OPA のコンパイラでは C 言語を目的言語とし，アセンブリ言語のコードを出力するための中間言語として用いている．ただし，一部の機能の高速化のために GCC[10] の拡張機能を利用している．我々が研究している拡張 C 言語 XC-cube は C 言語ではうまく記述できない協調機能を C 言語に追加した言語で GCC と同様の入れ子関数などが利用できる．

C-- 言語[6),9] は，C 言語ではうまく記述できないという問題を，C 言語を拡張するのではなく，アセンブリ言語へと近づけるというアプローチをとっている．我々が入れ子関数で提供しようとしている並行性・協調性の機能と同様に，マイグレーション・チェックポインティング，例外処理[9]，GC でのスタックスキャン[6] などが実現できる．また，stack walking を用いれば本論文で述べる動的負荷分散，あいるはマルチスレッド[18] も実現できると考えられる．

4.3　実 装 手 法

もともとの遅延タスク生成[8] は，論理的なスレッド生成を意味する `future` 構文の実行において，`future` の対象となる式の評価実行をそのまま行い，`future` の後の継続をスタックに保存し，この継続を他の idle なプロセッサがタスクとして盗むこともできるようにすることで負荷分散を行う方式である．継続は LTQ という deque で管理され継続は盗まれなかったときは自分でそのまま実行する．ここで，自プロセッサは LTQ の tail 側に対して継続の push/pop を行い，他のプロセッサは LTQ の head 側から継続を取り出してタスクとして盗む．このような手法は，基本的にはアセンブリ言語のプログラムでスタックを直接操作する必要がある．Cilk の処理系[3] でも遅延タスク生成を用いているが，C 言語を目的言語としていることから，継続を C のスタックとヒープに二重に保存している．C のスタックは C 言語が管理するものなので，出力する C のプログラムでは，ヒープ上に継続を置くようにする．継続は，自分でそのまま実行するか，idle なプロセッサがタスクとして盗むかどちらか一方なので，相互排他が必要である．それにはロックを用いるか[8]，Dekker のアルゴリズム等を利用する必要がある[3]．

一方，メッセージパッシング型遅延タスク生成[2] では，idle なプロセッサは継続（タスク）を直接盗むのではなく，まずタスクの要求を行い，要求されたプロセッサが，自分で継続からタスクを生成して要求したプロセッサに返す．この方式では，相互排他が不要となる上，callee save レジスタに保存されている継続

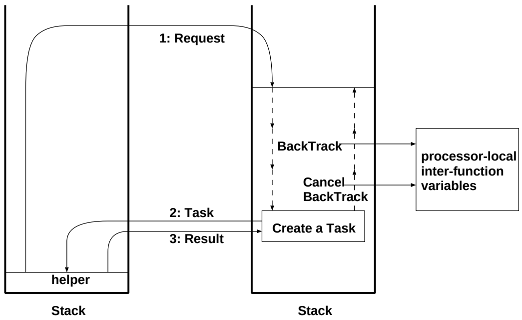

図 1　提案するタスクスティールプロトコル
Fig. 1　Proposed task steal protocol.

の情報も利用して，タスクの生成を行うことができる[12),17)]．また，OPA 言語の処理系における LTC[16)] のように，要求されてはじめて (より lazy に) 継続をヒープ上に作るようにすることもできる．

一方，継続ではなく，`future` 構文で論理的に生成されようとするスレッドを，lazy にタスクとして生成・授受する方法も考えられる．Leapfrogging[13)] はそのような方式である．この方式でもタスクは lazy に生成されるので，遅延タスク生成の一種と考えられる．ただし，未処理の継続ではなく未処理のスレッドからタスクを生成するという違いとなる[☆]．Leapfrogging では，未処理のスレッドをタスクとして盗まれたプロセッサはスレッドの結果を待つ状態になると，そのスレッドから派生したスレッドを盗み返す．これにより，逐次実行の場合より，スタックを余計に消費することはなくなる．今回提案する方式は，基本的にはこの Leapfrogging に基づいている．ただし，Cilk と同様に，生成したスレッドの完了は同じ関数内で必ず同期がとられるものとし，盗まれなかったスレッドを自プロセッサで実行する場合は，関数内に inline して実行するものとした．なお，Leapfrogging と同様の負荷分散を行うものには，SHE[5)] がある．

5. 単純な樹状再帰的計算の動的負荷分散

提案する動的負荷分散では次の方式を用いる．

- メッセージパッシング型[2)] である．
- Leapfrogging[13)] のように，未処理のスレッドから lazy にタスクを生成する
- Cilk と同様に，生成したスレッドの完了は同じ関数内で必ず同期がとられるものとする．
- 盗まれなかったスレッドは関数内 inlining で実行する．
- タスク生成のための情報は，LTQ からではなく入れ子関数で呼出し元に遡って見つける．

メッセージパッシングと，入れ子関数で呼出し元に遡って見つける様子を図 1 に示す．LTQ がなくなっていることが特徴の一つである．プログラムの一部を図 2 に示す．フラグのセットのみ (`restp = 1`) でスレッドが作られ，そのスレッドの (入れ子関数内での) タスク生成か，自分で inlining するときに (`restp = 0`) としている．

```
int fib(proc_env pr,
        int (*bkf)(req_buf_t *), int k){
  int s = 0;
  int restp = 1;
  int ret = 1;
  void **saved_req_port_p;
  int result_buf;
  int result_port;
  int bk(req_buf_t *req_buf_p){
    if(ret) ret = bkf(req_buf_p);
    if(ret) return 1;
    if(restp){
      task_buf_t *tbp =req_buf_p->task_buf_p;
      int *tpp        =req_buf_p->task_port_p;
      restp = 0;
      /* save info for "request back" */
      saved_req_port_p =req_buf_p->req_port_p;
      /* result not available yet */
      result_port = 0;
      /* put a task */
      tbp->f = tf_fib;
      tbp->sender = pr->myid;
      tbp->a.t1.result_buf_p = &result_buf;
      tbp->a.t1.result_port_p = &result_port;
      tbp->a.t1.k = k-2;
      /* transfer the task */
      finish_write_before_write();
      atomic_write_int(*tpp, 1);
      return 1;
    }
    return 0;
  }
  if(k <= 2) return 1;
  else {
    restp = 1; /* thread creation */
    POLL(pr->req_port, bk);
    s += fib(pr, bk, k-1);
    if(restp){
      restp = 0; /* inlining thread */
      s += fib(pr, bk, k-2);
    }else{
      /* wait for thread completion */
      while(atomic_read_int(result_port) == 0)
        /* request back a task and run*/
        steal_run_task(pr,saved_req_port_p,1);
      start_read_after_read();
      s += result_buf;
    }
    return s;
  }
}
```

図 2　Fibonacci の動的負荷分散コード
Fig. 2　load balancing code for Fibonacci.

☆ これらの中間的なものとして，Cilk などでは `spawn` の後から `spawn` の同期がとられるところ (`sync`) までの部分継続を盗むことも考えられる

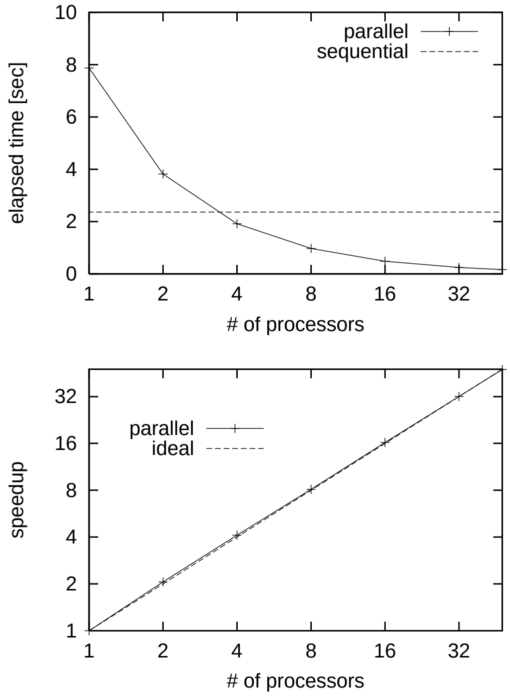

図 3　SPARC ベースのマルチプロセッサ上での Fibonaccti 数の実行時間と台数効果

Fig. 3　Elapsed time and speedup of Fibonacci number on SPARC-based Multiprocessor.

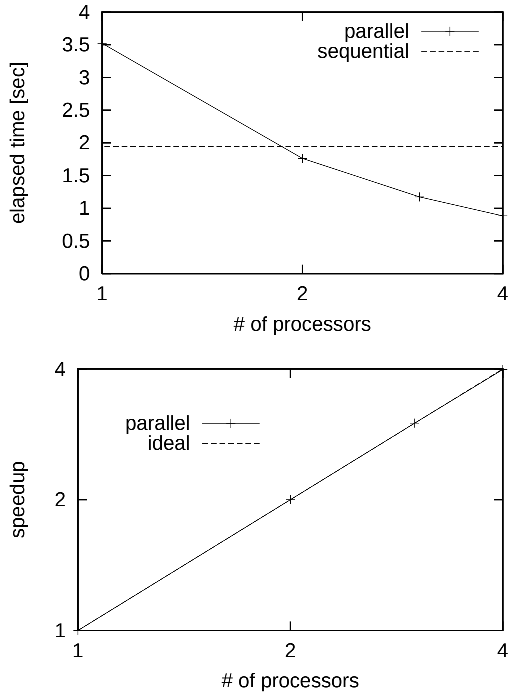

図 4　PowerPC ベースのマルチプロセッサ上での Fibonacci 数の実行時間と台数効果

Fig. 4　Elapsed time and speedup of Fibonacci number on PowerPC-based Multiprocessor.

6. 探索問題の動的負荷分散

単純な樹状再帰的計算の動的負荷分散に加えて

- タスクを生成するために，一時的にバックトラックする．

という機能が加わっている（図 1）．これによってペントミノ全解探索問題では，配置情報のコピーをタスク生成時のみに限定することが出来ている．ペントミノ全解探索問題の動的負荷分散コードを付録 A に示す．

コピーを減らすことで，コピーのコスト削減や，キャッシュヒット率向上が達成できると考えられる．

Cilk でも子の間で同じ領域を再利用可能とするための SYNCHED が用意されているが[11]，親子間はコピーしなくてはならない．また，SICStus Prolog では，並列実行時に束縛情報のコピーをさけるために，盗むプロセッサは盗まれるプロセッサに割り込みをかけて，スタック全体をコピーし，盗んだプロセッサ自身が必要なだけバックトラックをしてから盗んだ branch を実行するようになっている．

7. 評　　価

図 3，**図 4**，**図 5** には，種々の共有メモリ型並列計算機上で，fib(36) の Fibonacci 数計算の逐次プログラムの実行と，負荷分散を行う並列プログラムの並列実行を行った結果を示す．用いた共有メモリ型並列計算機の仕様/環境はそれぞれ以下のものである．

- Sun Ultra Enterprise 10000 （250MHz UltraSPARC-II, 1MB L2 Cache, 64CPUs）Solaris 7, gcc 2.95.2 `-O2 -mcpu=ultrasparc`
- IBM RS/6000 SP の 1 ノード（332MHz PowerPC 604e 命令 32KB/データ 32KB L1 cache，0.25MB L2 cache, 4CPUs）AIX Version 4.3, gcc 2.95.2 `-O2 -mcpu=powerpc`
- PC(1GHz Pentium-III, 2CPUs), Solaris 8, gcc 2.8.1 `-O2`

負荷分散を可能とした並列プログラムの 1CPU 上での実行時間を逐次プログラムの実行時間と比較すると，SPARC, PowerPC, Pentium それぞれの場合について，3.33 倍，1.81 倍，2.24 倍となる．SPARC のオーバヘッドが大きいのは，クロージャ生成のオーバヘッド，入れ子関数と元の関数で共有される変数のレジスタ割り付け阻害のオーバヘッドが高いためだと思われる．クロージャ生成のオーバヘッドについてはトランポリンに関する命令キャッシュのフラッシュを伴うこ

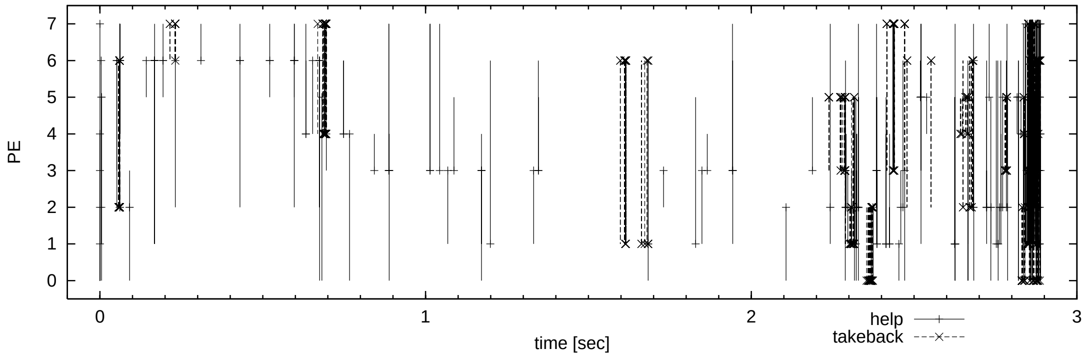

図 9　8 プロセッサ上のペントミノ全解探索でのタスク転送記録
Fig. 9　Record of task transfer by Pentomino search on 8 processors.

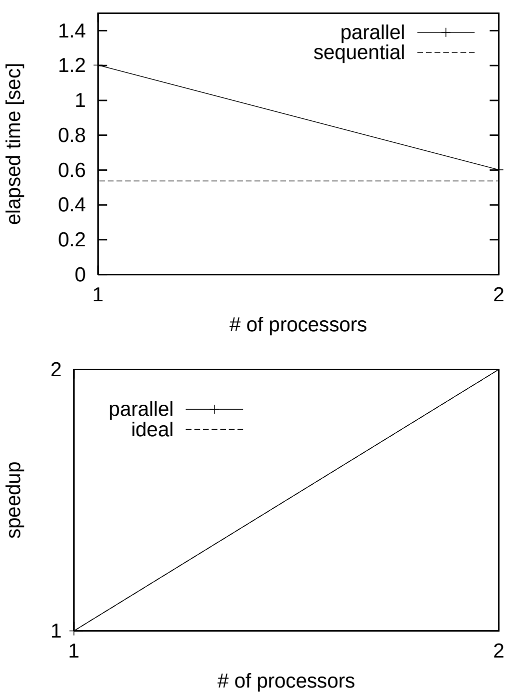

図 5　Pentium ベースのマルチプロセッサ上での Fibonacci 数の実行時間と台数効果
Fig. 5　Elapsed time and speedup of Fibonaccti number on Pentium-based Multiprocessor.

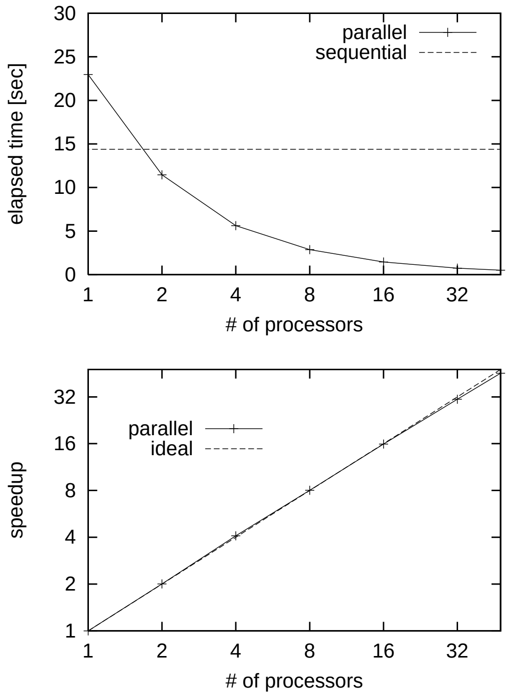

図 6　SPARC ベースのマルチプロセッサ上でのペントミノ全解探索の実行時間と台数効果
Fig. 6　Elapsed time and speedup of Pentomino search on SPARC-based Multiprocessor.

とが挙げられる．PowerPC のオーバヘッドが比較的小さいのは，PowerPC ではもともと関数ポインタをコードのアドレスではなく，コードとその環境（定数ポインタのための）を表すデータ構造へのポインタとしているため，クロージャ生成時のスタック上への動的コード生成が不要となるためである．Pentium の場合にそれほど悪くないのは，命令キャッシュの明示的なフラッシュが不要な点，もともとレジスタが少ないために，レジスタ割り付け阻害のオーバヘッドが目立たない点が考えられる．

また，図 3，図 4，図 5 のように並列プログラムの台数効果については，全ての計算機について理想的な台数効果が得られた．例えば Ultra Enterprise 10000 上では 48CPU の使用時に 47.8 倍の台数効果が得られた．これは，提案する負荷分散方式が十分に機能していることを示している．

図 6，**図 7**，**図 8** には，種々の共有メモリ型並列計算機上で，ペントミノ全解探索の逐次プログラムの実行

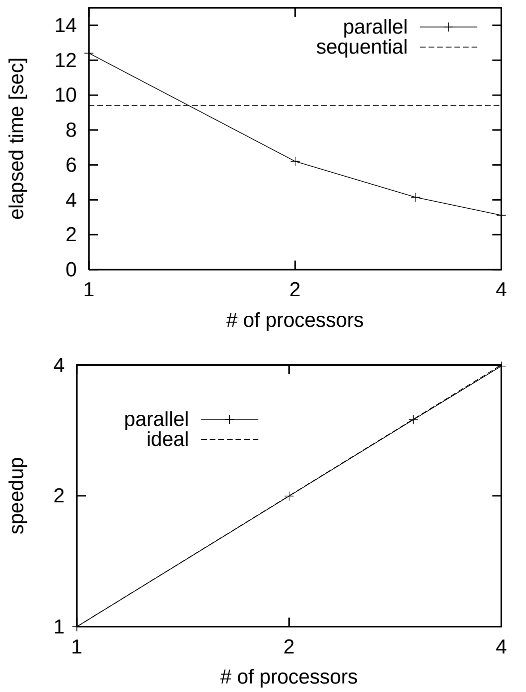

図 **7** PowerPC ベースのマルチプロセッサ上でのペントミノ全解探索の実行時間と台数効果

Fig. 7 Elapsed time and speedup of Pentomino search on PowerPC-based Multiprocessor.

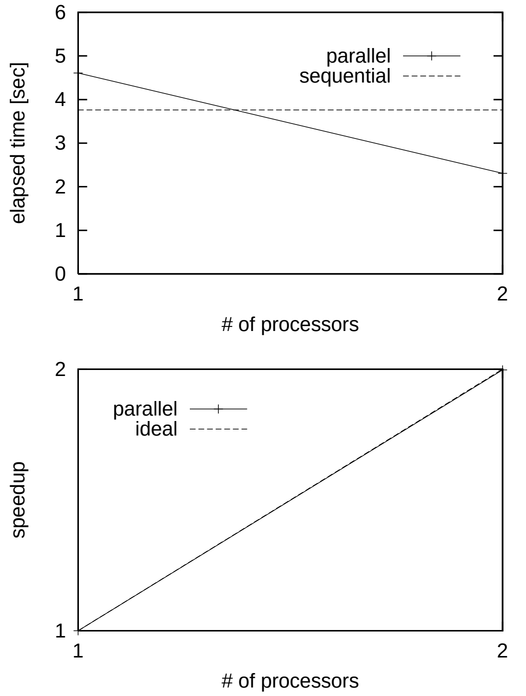

図 **8** Pentium ベースのマルチプロセッサ上でのペントミノ全解探索の実行時間と台数効果

Fig. 8 Elapsed time and speedup of Pentomino search on Pentium-based Multiprocessor.

と，負荷分散を行う並列プログラムの並列実行を行った結果を示す．負荷分散を可能とした並列プログラムの 1CPU 上での実行時間を逐次プログラムの実行時間と比較すると，SPARC, PowerPC, Pentium それぞれの場合について，1.60 倍，1.32 倍，1.23 倍となった．PowerPC と Pentium で逆転しているのは，一つの関数内での仕事の量が増えているため，クロージャ生成のオーバヘッドより，入れ子関数と元の関数で共有される変数のレジスタ割り付け阻害のオーバヘッドのほうが重要となるためだと考えられる．

また，図 6，図 7，図 8 のように並列プログラムの台数効果については，全ての計算機についてほぼ理想的な台数効果が得られた．例えば Ultra Enterprise 10000 上では 32CPU 使用時には，30.9 倍の台数効果，48CPU の使用時に 45.4 倍の台数効果が得られた．Fibonacci 数計算とくらべてやや落ちるのは，枝刈りの結果，せっかく授受したタスクの仕事量が少なすぎることもあるためと考えられる．

ここで，図 **9** には，8 プロセッサ上のペントミノ全解探索でのタスク転送記録を示す．横軸が時刻で縦軸がプロセッサ番号である．図中 `help` はスタックが空になったプロセッサが他のプロセッサからタスクの供給を受けた場合を示し，`takeback` はタスク（スレッド）の完了待ちになったプロセッサが，タスクを盗んだプロセッサから仕事の一部をタスクとして取り返す場合を示す．実行の初期・中期ではときたま仕事の取り返しが繰り返されている以外は，タスクの生成・授受は最小限に抑えられているといえる．実行の終盤では，残り仕事量が減ってくるためタスクの生成・授受が活発なるが，残り仕事量も減っていくためそのうち実行が完了することになる．

8. 今後の課題

今後の課題には以下のものがある．

- 分散メモリ型上での評価を行う予定である．Leapfrogging の制約の緩和すること，タスクのバッファリングすることで，タスク授受の遅延隠蔽が可能かを試す予定である．
- 高水準言語 OPA 言語など高水準言語で提案手法を利用するとして言語仕様をどうするかという問題がある．Scheme の dynamic-wind[7] のような機能があればよい．
- 拡張 C 言語 XC-cube の実装として．クロージャ生成の軽量化のための実装を進めている．

9. おわりに

本論文では，入れ子関数を利用することで，バックトラックにも対応できる動的負荷分散が記述可能であることを示した．また，共有メモリ型並列計算機上で提案する記述方式のプログラムの評価を行い，数台のプロセッサを用いれば逐次プログラムと比較して速度向上が得られること，並列プログラム実行においては理想的な台数効果が得られることを示した．

謝辞　Ultra Enterprise 10000 の使用を快く許可していただきました東京大学の平木先生に，謹んで感謝の意を表します．

参　考　文　献

1) Breuel, T. M.: Lexical Closures for C++, *Usenix Proceedings, C++ Conference* (1988).
2) Feeley, M.: A Message Passing Implementation of Lazy Task Creation, *Proceedings of International Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, Lecture Notes in Computer Science, No. 748, Springer-Verlag, pp. 94–107 (1993).
3) Frigo, M., Leiserson, C. E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol. 33, No. 5, pp. 212–223 (1998).
4) Halstead, Jr, R.H.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Parallel Lisp: Languages and Systems* (Ito, T. and Halstead, R. H.(eds.)), Lecture Notes in Computer Science, Vol. 441, Sendai, Japan, June 5–8, Springer, Berlin, pp. 2–57 (1990).
5) Ito, T.: Sound Parallelization Framework for Parallel Scheme Programming, *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, World Scientific, pp. 3–40 (2000).
6) Jones, S. P., Ramsey, N. and Reig, F.: C--: a Portable Assembly Language that Supports Garbage Collection (1999).
7) Kelsey, R., Clinger, W. and Rees, J.: Revised^5 Report on the Algorithmic Language Scheme, *ACM SIGPLAN Notices*, Vol. 33, No. 9, pp. 26–76 (1998).
8) Mohr, E., Kranz, D.A. and Halstead, Jr, R.H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
9) Ramsey, N. and Reig, F.: A Single Intermediate Language That Supports Multiple Implementations of Exceptions (2000).
10) Stallman, R. M.: *Using and Porting GNU Compiler Collection*, Free Software Foundation, Inc., for gcc-2.95 edition (1999).
11) Supercomputing Technologies Group: *Cilk-5.1(Beta1) Reference Manual*, MIT Laboratory for Computer Science (1997). http://theory.lcs.mit.edu/~cilk.
12) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pp. 60–71 (1999).
13) Wagner, D. B. and Calder, B. G.: Leapfrogging: A Portable Technique for Implementing Efficient Futures, *Proceedings of Principles and Practice of Parallel Programming (PPoPP'93)*, pp. 208–217 (1993).
14) 八杉昌宏, 高田潤, 田畑悠介, 小宮常康, 湯淺太一: 共有メモリ向けプリミティブとその GCC を使った実現, 情報処理学会論文誌：プログラミング, Vol. 43, No. SIG 1 (PRO 13), pp. 118–132 (2002).
15) 八杉昌宏, 馬谷誠二, 鎌田十三郎, 田畑悠介, 伊藤智一, 小宮常康, 湯淺太一: オブジェクト指向並列言語 OPA のためのコード生成手法, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 11 (PRO 12), pp. 1–13 (2001).
16) 馬谷誠二, 八杉昌宏, 小宮常康, 湯淺太一: ループ内スレッド生成に対応した Lazy Task Creation, 並列処理シンポジウム (JSPP2001) 論文集, pp. 157–164 (2001).
17) 田端邦男, 田浦健次朗, 米澤明憲: C プログラムにおける Lazy Task Creation, 情処研報 97-PRO-14(SWoPP'97), pp. 79–84 (1997).
18) 田畑悠介, 八杉昌宏, 小宮常康, 湯淺太一: 入れ子関数を利用したマルチスレッドの実現, 情報処理学会論文誌：プログラミング, Vol. 43, No. (PRO 14) (2002). (採録予定).

付　　録

A.1 入れ子関数を利用して動的負荷分散を行うペントミノ全解探索プログラム

```
#include <stdio.h>
#include <sys/time.h>
#include <math.h>
#include "xccmem.h"
#include "systhr.h"

#define MAXNPROC 64
#define DPRINT0

typedef struct _proc_locals *proc_env;
```

```
typedef struct _task_buf {
  void (*f)(proc_env, struct _task_buf *);
  int sender; /* optional */
  int why;    /* optional */
  union {
    struct {
      int *result_buf_p;
      int *result_port_p;
      int k, i;
      int a[12];
      int b[70];
    } t1;
  } a;
} task_buf_t;

typedef struct _req_buf {
  task_buf_t *task_buf_p;
  int *task_port_p;
  void **req_port_p; /* (struct _req_buf **) */
} req_buf_t;

struct _proc_locals {
  int n_proc;
  int myid;
  task_buf_t task_buf;
  int task_port;
  req_buf_t req_buf;
  void *dumy1[64];
  void *req_port;   /* (req_buf_t *) */
  void *dumy2[64];
  int a[12];        /* 0, if piece is unused */
  int b[70];        /* 0, if the board location
                           x+y*7 is empty */
  void *dumy3[64];
} pe[MAXNPROC];

void init_proc_locals(proc_env pr, int id,
                      int num_proc){
  int i;
  for(i=0;i<12;i++) pr->a[i] = 0;
  for(i=0;i<70;i++) pr->b[i] = 0;
  /* right side wall */
  for(i=6;i<70;i+=7) pr->b[i] = '\n';
  pr->n_proc = num_proc;
  pr->myid = id;
  pr->req_port = 0;
}

void tf_do_nothing(proc_env pr,
                   task_buf_t *task_buf_p){
}

void refuse_req(proc_env pr){
  /* transfer a null task to refuse request. */
  void *req_buf_p=atomic_read_ptr(pr->req_port);
  if(req_buf_p){
    req_buf_t *rbp = (req_buf_t *)req_buf_p;
    start_read_after_read();
    atomic_write_ptr(pr->req_port,0);
    rbp->task_buf_p->f = tf_do_nothing;
    finish_write_before_write();
    atomic_write_int(*(rbp->task_port_p), 1);
  }
}

/* For idle processor */
void steal_run_task(proc_env pr,
                    void **req_port_p, int why){
  pr->task_port = 0;
  pr->task_buf.why = why;
  /* request a task */
  pr->req_buf.task_buf_p  = &(pr->task_buf);
  pr->req_buf.task_port_p = &(pr->task_port);
  pr->req_buf.req_port_p  = &(pr->req_port);
  finish_write_before_write();
  while(cas_ptr(*req_port_p,
              0, (void *)(&(pr->req_buf))))
    refuse_req(pr);
  /* wait for a task */
  while(atomic_read_int(pr->task_port) == 0)
    refuse_req(pr);
  start_read_after_read();
  /* run the stolen task */
  (pr->task_buf.f)(pr, &(pr->task_buf));
}

int bv;

void *helper(void *p){
  proc_env pr=(proc_env)p;
  double r, q;
  int n;
  systhr_bind_to_proc(pr->myid);
  systhr_bind_to_proc(pr->myid);
  /* atomic-inc */
  do{n = bv;}while(cas_int(bv, n, n+1));
  for(r = sqrt(0.5 + pr->myid),
      q = sqrt(r + pr->myid), r -= (int)r;;
      r = r * 3.0 + q, r -= (int)r){
    int tg = (int)(pr->n_proc * r);
    if(tg != pr->myid){
      steal_run_task(pr, &(pe[tg].req_port), 0);
    }
  }
}

void init_parallel(int num_proc){
  int i;
  for(i=0; i<num_proc; i++)
    init_proc_locals(pe+i, i, num_proc);
  systhr_init(num_proc);
  systhr_bind_to_proc(0); bv = 1;
  for(i=1; i<num_proc; i++)
    systhr_create(helper, pe+i);
  while(atomic_read_int(bv) < num_proc)
    start_read_after_read();
}

#define POLL(req_port, bk) \
do{ \
 void *req_buf_p = atomic_read_ptr((req_port));\
 if(req_buf_p){ \
   req_buf_t *rbp = (req_buf_t *)req_buf_p; \
   start_read_after_read(); \
   atomic_write_ptr((req_port),0); \
   if((bk)(rbp) == 0){ \
     rbp->task_buf_p->f = tf_do_nothing; \
     finish_write_before_write(); \
     atomic_write_int(*(rbp->task_port_p), 1); \
   } \
 } \
}while(0)

/*
 A piece shape for some direction.
 A piece consists of fives squares.
 Position (x,y) is represented by integer x+y*7.
 Starting from the topmost-leftmost square of
 a piece, a piece shape is given by the relative
 positions of the other four squares in
 topmost-leftmost order.
 Here, "topmost" means "smallest y-coordinate",
 "leftmost" means "smallest x-coordinate", and
 "relative" means "relative to the previous
```

```
 square".
*/
typedef int pshape[4];

pshape
p1[2] ={{1,1,1,1}, {7,7,7,7}},
p2[8] ={{7,1,1,1},{4,1,1,1},{1,1,1,4},{1,1,1,7},
        {7,7,7,1},{7,7,6,1},{1,6,7,7},{1,7,7,7}},
p3[8] ={{6,1,1,1},{5,1,1,1},{1,1,1,5},{1,1,1,6},
        {7,1,6,7},{7,7,1,6},{7,6,1,7},{6,1,7,7}},
p4[4] ={{7,7,1,1},{1,1,5,7},{1,1,7,7},{7,5,1,1}},
p5[4] ={{2,5,1,1},{1,1,5,2},{1,6,7,1},{1,7,6,1}},
p6[4] ={{7,1,1,5},{5,1,1,7},{7,6,1,1},{1,1,6,7}},
p7[8] ={{7,1,1,6},{5,1,1,6},{6,1,1,5},{6,1,1,7},
        {1,7,1,6},{7,1,5,1},{1,5,1,7},{6,1,7,1}},
p8[4] ={{7,1,1,7},{5,1,1,5},{1,7,7,1},{1,6,6,1}},
p9[1] ={{6,1,1,6}},
p10[8]={{1,6,1,1},{1,1,5,1},{1,5,1,1},{1,1,6,1},
        {7,1,6,1},{1,6,1,6},{6,1,6,1},{1,6,1,7}},
p11[8]={{1,4,1,1},{1,1,7,1},{1,7,1,1},{1,1,4,1},
        {7,1,7,7},{7,6,1,6},{6,1,6,7},{7,7,1,7}},
p12[4]={{7,1,7,1},{1,5,1,6},{1,7,1,7},{6,1,5,1}};

typedef struct {
  int n;    /* number of different directions */
  /* pointer to the array of piece shapes */
  pshape *pss;
} piece;


piece ps_org[12] = {{2,p1},{8,p2},{8,p3},{4,p4},
                    {4,p5},{4,p6},{8,p7},{4,p8},
                 {1,p9},{8,p10},{8,p11},{4,p12}};

/* avoid symmetric solutions */
pshape p2d[8] = {{4,1,1,1},{7,7,6,1}};
piece ps[12] = {{2,p1},{2,p2d},{8,p3},{4,p4},
                {4,p5},{4,p6},{8,p7},{4,p8},
                {1,p9},{8,p10},{8,p11},{4,p12}};

void show_b(proc_env pr){
  int i;
  for(i=0;i<70;i++)
    putchar(pr->b[i] ? pr->b[i] : ' ');
  putchar('\n');
}

int search(proc_env, int (*)(req_buf_t *), int);

/* examine the "i"-th piece for all directions
   at the first empty location "k" */
int try_piece(proc_env pr,
              int (*bkf)(req_buf_t *),
              int k, int i){
  int s = 0;          /* the sum of solutions */
  int d;              /* direction */
  int n = ps[i].n;    /* number of directions */
  int bk(req_buf_t *req_buf_p){
    int ret;
    int kk=k,l;
    int *pss = ps[i].pss[d];
    /* remove the piece (backtrack) */
    pr->a[i] = 0; pr->b[kk=k] = 0;
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = 0;
    ret = bkf(req_buf_p);
    /* put the piece (cancel backtrack)*/
    pr->a[i] = 1; pr->b[kk=k] = i+'A';
    for(l=0;l<4;l++) pr->b[kk += pss[l]] =i+'A';
    return ret;
  }
  for(d=0;d<n;d++){
    int kk=k,l;
    int *pss = ps[i].pss[d];
    /* room available? */
    for(l=0;l<4;l++)
      if((kk += pss[l]) >= 70 || pr->b[kk] !=0)
        goto Ln;
    /* put the piece */
    pr->a[i] = 1; pr->b[kk=k] = i+'A';
    for(l=0;l<4;l++) pr->b[kk += pss[l]] =i+'A';
    /* find the first empty location */
    for(kk=k; kk<70; kk++)
      if( pr->b[kk] == 0 ) break;
    /* recursive search */
    s += search(pr, bk, kk);
    /* remove the piece (backtrack) */
    pr->a[i] = 0; pr->b[kk=k] = 0;
    for(l=0;l<4;l++) pr->b[kk += pss[l]] = 0;
  Ln:
    continue;
  }
  return s;
}

struct timeval tv0;

void tf_backup_and_try_piece(proc_env pr,
                        task_buf_t *task_buf_p){
  int bk(req_buf_t *req_buf_p){ return 0; }
  int a_bak[12];
  int b_bak[70];
  int jj;
  int *saved_result_port_p;
  int *saved_result_buf_p;
  task_buf_t *tbp  = task_buf_p;
  /* backup */
  for(jj=0;jj<12;jj++) a_bak[jj] = pr->a[jj];
  for(jj=0;jj<70;jj++) b_bak[jj] = pr->b[jj];
  /* save info for "result back" */
  saved_result_port_p = tbp->a.t1.result_port_p;
  saved_result_buf_p  = tbp->a.t1.result_buf_p;
  /* get a task */
#ifdef DPRINT
  {
    struct timeval tv1;
    gettimeofday(&tv1, 0);
    printf("%9d %2d gets %2d %2d of %2d for %d\n",
           (tv1.tv_sec-tv0.tv_sec)*1000000
           +tv1.tv_usec-tv0.tv_usec,
           pr->myid, tbp->a.t1.k, tbp->a.t1.i,
           tbp->sender, tbp->why);
  }
#endif
  for(jj=0;jj<12;jj++)
    pr->a[jj] = tbp->a.t1.a[jj];
  for(jj=0;jj<70;jj++)
    pr->b[jj] = tbp->a.t1.b[jj];
  {
    int k = tbp->a.t1.k, i = tbp->a.t1.i, s = 0;
    if(pr->a[i] == 0)
      s += try_piece(pr, bk, k, i);
    *saved_result_buf_p = s;
  }
  finish_write_before_write();
  atomic_write_int(*saved_result_port_p, 1);
  /* restore */
  for(jj=0;jj<12;jj++) pr->a[jj] = a_bak[jj];
  for(jj=0;jj<70;jj++) pr->b[jj] = b_bak[jj];
}

int search(proc_env pr, int (*bkf)(req_buf_t *),
           int k){
  int s = 0;          /* the sum of solutions */
  int i = 0, si = 12;
```

```
  int ret = 1;
  void **saved_req_port_p[12];/* req_buf_t ** */
  int result_buf[12];
  int result_port[12];
  int bk(req_buf_t *req_buf_p){
    if(ret) ret = bkf(req_buf_p);
    if(ret) return 1;
    while(i<si){
      int ii = --si;
      if(pr->a[ii] != 0){
        /* avoid a trivial task */
        result_port[ii] = 1;
        result_buf[ii]  = 0;
      }else{
        int jj;
        task_buf_t *tbp =req_buf_p->task_buf_p;
        int *tpp        =req_buf_p->task_port_p;
        /* save info for "request back" */
        saved_req_port_p[ii]
          = req_buf_p->req_port_p;
        /* result not available yet */
        result_port[ii] = 0;
        /* put a task */
        tbp->f = tf_backup_and_try_piece;
        tbp->sender = pr->myid;
        tbp->a.t1.result_buf_p =result_buf+ii;
        tbp->a.t1.result_port_p =result_port+ii;
        tbp->a.t1.k = k;
        tbp->a.t1.i = ii;
        for(jj=0;jj<12;jj++)
          tbp->a.t1.a[jj] = pr->a[jj];
        for(jj=0;jj<70;jj++)
          tbp->a.t1.b[jj] = pr->b[jj];
        /* transfer the task */
        finish_write_before_write();
        atomic_write_int(*tpp, 1);
        return 1;
      }
    }
    return 0;
  }
  if(k == 70){
    /* a solution is found */
    /* show_b(pr); */
    return 1;
  }
  else {
    /* prune */
    if((k+7 >= 70 || pr->b[k+7] != 0) &&
       (pr->b[k+1] != 0 ||
         (k+8 >= 70 || pr->b[k+8] != 0) &&
         pr->b[k+2] != 0))
      return 0;
    while(i<si){
      int ii=i++;
      POLL(pr->req_port, bk);
      if(pr->a[ii] == 0)
        s += try_piece(pr, bk, k, ii);
    }
    /* i == si */
    while(i<12){
      int ii=i++;
      while(atomic_read_int(result_port[ii])==0)
        /* request back a task and run*/
        steal_run_task(pr,
                       saved_req_port_p[ii], 1);
      start_read_after_read();
      s += result_buf[ii];
    }
    return s;
  }
}

main(int argc, char *argv[]){
  int bk(req_buf_t *req_buf_p){ return 0; }
  struct timeval tv[2];
  int num_proc=2, ans;
  if(argc>1) num_proc=atoi(argv[1]);
  init_parallel(num_proc);
  gettimeofday(tv+0, 0); tv0 = tv[0];
  ans = search(pe+0, bk, 0);
  gettimeofday(tv+1, 0);
  printf("answers: %d\n", ans);
  printf("%lf\n", tv[1].tv_sec-tv[0].tv_sec
    +(tv[1].tv_usec-tv[0].tv_usec)*0.000001);
}
```