並列スクリプト言語 Xcrypt によるジョブ分割実行に基づく並列化ツール

曽川 景介　平石 拓　安部 達也　三宅 洋平　岩下 武史　中島 浩

独立な部分問題に分割可能な問題においては，各部分問題を解く処理の同時実行が可能であり，それにより問題全体を解く処理を並列化できる．特に，1 つの部分問題を解く処理を 1 つのプログラム実行に対応させるジョブレベル並列実行により，元の逐次プログラムを変更することなく並列計算環境を手軽に活用することができる．我々は，ユーザが問題の分割方法と結果の結合方法をそれぞれ関数として与えるだけで，このような並列化を行うツールを開発した．本ツールは，計算中の部分問題を分割して複数のアイドルな計算資源に割り当て，その部分問題を最初から（並列に）実行させるという動的負荷分散を行う．これにより，均等な負荷にあらかじめ分割することが困難な問題においても計算資源を有効に活用する．本研究では，相同性探索と SAT ソルバの既存の逐次実装に対して本ツールを適用し十分な並列効果が得られることを確かめた．本ツールは我々が開発中のジョブレベル並列スクリプト言語 Xcrypt 上に実装しており，バッチキューイングシステムの利用を前提としている多くのスーパーコンピュータ環境においても動作する．

1　はじめに

近年のスーパーコンピュータや研究室レベルの計算クラスタ等の並列計算環境を有効活用するため，アプリケーションの実装を並列化することが必要不可欠になってきている．一般に，並列化を行うためには，MPI や OpenMP などを利用してプログラム自身を並列化することが多い．しかし，プログラムを並列化するためには十分な知識と手間が必要である．この問題は，特にクラスタのような分散メモリ環境においてはさらに顕著になる．

プログラムで解かせたい問題が独立な部分問題に容易に分割可能な場合，それぞれの部分問題に対応する入力を生成し，各入力に対してプログラムを同時実行する（ジョブレベル並列）ことで，もとの（逐次）プログラムを一切書き換えることなく処理を並列化することができる．無論この方法はプログラム自身の並列化に比べ，それぞれのプロセスが処理の効率化のための情報を共有できないことや並列化そのもののコストが大きいことにより，性能は低くなる．しかし，大規模な並列計算環境において極小な実装コストで実行時間を短縮できるという意味で，この方法は有用であると考えられる．

そこで我々は，ユーザがプログラムへの入力を解きたい問題の部分問題への入力となるように分割する処理，およびそれらの出力結果を統合してもとの全体問題の結果を得るための処理をそれぞれ Perl 関数として定義することで，上記のような並列化を自動的に行うツールを開発した．

本ツールは，SAT 問題のような各部分問題の負荷が均等になるように静的に問題を分割することが困難な問題に対応するため，動的負荷分散機構も備える．我々は，過剰な分割を行わず，かつアプリケーションプログラムに一切手を加えずに実現可能な動的負荷分散手法として，最初は計算資源（CPU コア等）の数に合わせて粗粒度な分割をしておき，一部の部分問題が完了して対応する計算資源がアイドルになれば，

A Job Division-based Parallelization on the Parallel Script Language Xcrypt

Keisuke Sogawa, 京都大学情報学研究科システム科学専攻, Department of Systems Science, Graduate School of Informatics, Kyoto University.

Tasuku Hiraishi, Tatsuya Abe, Yohei Miyake, Takeshi Iwashita, Hiroshi Nakashima, 京都大学学術情報メディアセンター, Academic Center for Computing and Media Studies, Kyoto University.

別の計算資源で実行中の部分問題を再分割しアイドルな計算資源に割り当てるという手法を採用した．この手法は，既に実行中の計算を最初から並列に実行しなおすことで全体の実行時間を短縮することを狙ったものである．同じ計算が重複して行われるコストがあるが，各部分問題の負荷の差が大きい問題においては有効にはたらくことが期待できる．

本研究では，部分問題の負荷の正確な見積が容易な問題として相同性探索，見積が困難な問題として SAT ソルバを取り上げ，それぞれの既存の逐次実装に対して本ツールを適用し，本ツールが上記の動的負荷分散機構も含め有効にはたらくことを確かめた．

本ツールは我々が開発中のジョブレベル並列スクリプト言語 Xcrypt [10] [8] [9] のモジュールとして実装しており，バッチキューイングシステムの利用を前提とした多くのスーパーコンピュータ環境においても動作する．

本論文の構成は以下の通りである．まず 2 章で本ツールの仕様を説明し，次に 3 章で実装について説明する．次に 4 章で性能評価の結果を示し，5 章で議論を行い，最後に 6 章でまとめと今後の課題を述べる

2 仕様

本章では本ツールの外部仕様を,「クラスタシステム上での並列プログラミングコンテスト 2009」 [11] の非数値部門の課題においてサンプルプログラムとして与えられた，相同性探索の逐次実装プログラムを例として用いて説明する．

2.1 例題

上記の相同性探索プログラムは，クエリ配列 $q_1, q_2, \ldots, q_m$ とデータ配列 $d_1, d_2, \ldots, d_n$ を入力として受け取ると，各 q_i（$1 \leq i \leq m$）に対して最も類似度が高いデータ d_{max_i} を $d_1, d_2, \ldots, d_n$ の中から選択し，(q_i, d_{max_i}) の組を出力する[†1]．プログラム全体の出力は $(q_1, d_{\mathrm{max}_1}), (q_2, d_{\mathrm{max}_2}), \ldots, (q_m, d_{\mathrm{max}_m})$ となる．

入力 $q_1, q_2, \ldots, q_m$ および $d_1, d_2, \ldots, d_n$ はそれぞれクエリ配列，データ配列がこの順に並べられたテキストファイルとして与えられる．出力も $(q_1, d_{\mathrm{max}_1}), (q_i, d_{\mathrm{max}_2}), \ldots, (q_m, d_{\mathrm{max}_m})$ を表現するテキストファイルとして返す．

このプログラムの計算をジョブレベルで並列化するための手段としては，クエリ配列の分割あるいはデータ配列の分割が考えられるが，ここでは，結果の統合がより単純なクエリ配列の分割を考える．探索プログラムの実行ファイルが `homology` クエリ配列のファイルが `query.txt`，データ配列のファイルが `data.txt`，結果はファイル `output.txt` に出力されるとすると，プログラムを実行するコマンドラインは，

```
homology query.txt data.txt output.txt
```

となる．これは以下の手順でジョブ並列化しても同じ結果が得られる．

1. テキストファイル `query.txt` を（含まれるクエリの数が端数を除いて同数になるように）分割して，`query1.txt`，...，query$p$.txt を生成する．
2. コマンド
   ```
   homology query1.txt data.txt output1.txt
   ...
   homologyqueryp.txt data.txt outputp.txt
   ```
   を同時に実行する．
3. 2. の全ての実行が完了するのを待つ．
4. 2. で得られた各出力結果（`output1.txt`，...，output$p$.txt）を連結する．

2.2 ユーザ記述

本ツールのユーザは，以下の情報を記述することでプログラムをジョブ並列化して実行することができる．

1. プログラムを通常実行するためのコマンドライン
2. 計算資源の数（並列数）
3. 動的負荷分散の有無
4. 実行中のジョブが再分割されるまでの最低実行

†1 実際のコンテストでは，最高スコアのデータが同スコアで複数存在する場合はその全てを出力する，クエリとデータの組だけでなくそのスコアやそのスコアの根拠となるクエリ，データの成分も出力するなど詳細な仕様が設定されているが，ここでは簡単化して説明する．

```
%template = (
  'id' => 'job_homology',    # 任意の名前
  'exe' => 'homology',       # 実行ファイル
  'arg0' => 'query.txt',     # 以下，コマンドライン引数
  'arg1' => 'data.txt',      #
  'arg2' => 'output.txt',    #
  'nparallel' => 8,          # 並列数
  'redivide' => 0,           # 再分割（動的負荷分散）の有無
  'kill_parent' => 0,        # 再分割時の元のジョブの中止の可否
  'redivide_time' => 0,      # 再分割までの最低実行時間（秒）
);
```

図 1　相同性探索におけるジョブテンプレートオブジェクトの定義

```
sub divide
{
  my ($parent,$ndiv) = @_;
  my @children = ();
  my @divided_query_files = ();
  my $datapath = $self->{arg0_1};
  @divided_query_files = divide_query_file($parent->{arg0});
  for( my $i = 0; $i < $ndiv; $i++ )
  {
    my %newjobtmpl = (
    'id' => $parent->{id}.'_'.$i,
    'exe'  => $parent->{exe},
    'arg0' => $divided_query_files[$i],
    'arg1' => $parent->{arg1},
    'arg2' => $parent->{arg2} . $i,
    );
    push(@children, \%newjobtmpl);
  }
  return @children;
}
```

図 2　相同性探索における分割関数の定義

時間

5. ジョブ再分割時，分割元のジョブを中止するかどうか
6. 実行する問題を分割する方法
7. 実行した結果を統合する方法

このうち，1.–5. は「ジョブテンプレートオブジェクト」（Perl のハッシュオブジェクト）のメンバ，6. と 7. はそれぞれ `divide`，`merge` という名前の Perl 関数として定義する．2.1 節の例題におけるコマンドラインの定義例を図 1，分割関数 `divide`，統合関数 `merge` の定義例をそれぞれ図 2，図 3 に示す．

```
sub merge
{
  my @jobs = @_;
  my @output_files = ();
  my $parent = $jobs[0]->{parent};
  foreach $job (@jobs) {
    push (@output_files, $job->{arg2});
  }
  # 出力ファイルを結合して，親ジョブの出力ファイル名で保存
  concat_file ($parent->{arg2}, @output_files);
}
```

図 3　相同性探索における統合関数の定義

`divide` 関数は，分割元のジョブに対応する「ジョブオブジェクト」への参照および分割数を引数として受け取り，入力ファイルの分割などの必要な処理を行い，分割実行のためのコマンドラインの情報を持たせたジョブテンプレートオブジェクトを分割数だけ生成し，その参照の列を返す．ここで，ジョブオブジェクトは，テンプレートオブジェクトにシステムがジョブの管理情報を追加のメンバとして加えたオブジェクトである．詳細は文献 [9] を参照されたい．

`merge` 関数は，分割実行された全てジョブに対応するジョブオブジェクトの参照の列を引数として受け取り，それらの出力ファイルの結合など必要な処理を行うことで，分割前のジョブを実行した場合と同じ結果になるようにする（返り値はない）．

2.3　システムの動作

本システムの動作は以下の通りである．

1. ユーザが記述したジョブテンプレートオブジェクトおよび分割関数 `divide`，統合関数 `merge` の定義を Perl の変数・関数定義として評価する．
2. ジョブテンプレートオブジェクトからジョブオブジェクトを生成する．
3. 生成したジョブオブジェクトとテンプレートオブジェクトのメンバ `nparallel` の値を引数として `divide` 関数を呼び出し，その返り値であるテンプレートオブジェクト列からジョブオブジェクト列を得る．
4. そのジョブオブジェクト列に対応する全てのジョブを非同期に実行する．
5. 全てのジョブの終了を待ち合わせる．

6. ジョブオブジェクト列を引数として merge 関数を呼び出す．

さらに，動的負荷分散オプション（図 1 の redivide メンバで指定）が有効になっていた場合，以下の処理を定期的に行う監視スレッドが起動する．

1. 実行中のジョブ数が nparallel の設定値より小さいかをチェックする．
2. もし小さければ，実行中のジョブのうち，現時点までの実行時間が redivide_time の設定値より長いジョブがあるかをチェックする．
3. もしそのようなジョブがあれば，その中から 1 つを選択する（複数ある場合は，それまでの分割回数が少ないもの，実行時間が長いもの，の順で優先的に選択する）．
4. 選択されたジョブに対応するジョブオブジェクトを引数として divide 関数を呼び出し，その返り値から得られるジョブオブジェクトに対応する全てのジョブを非同期に実行する．ただし，得られたジョブオブジェクトの数が空き計算資源の数（nparallel の値と実行中のジョブ数の差）より多い場合は，まず空きの数だけ実行し，残りは資源が空き次第実行する．
5. kill_parent の設定が有効になっていれば，divide 関数の引数となっていたジョブオブジェクトに対応するジョブ（分割元のジョブ）を中断する．
6. 再分割して実行した全てのジョブの終了を待ち合わせ，その後 merge 関数を実行するためのスレッドを起動する．

kill_parent が無効と設定されていた場合，同じ計算に相当するジョブが，非分割実行と分割実行で重複して実行されることになり，どちらかの実行が完了すればもう片方の実行は不要になる．そのため，各ジョブの終了時および merge 完了時に，不要になったジョブが存在しないかをチェックし，存在すればそれを中断するという処理が新たに追加される．

3 実装

本ツールは，我々が開発中のジョブ並列スクリプト言語 Xcrypt [10][8][9] のモジュールとして実装した．

3.1 ジョブ並列スクリプト言語 Xcrypt

Xcrypt は Perl をベースとしたジョブ並列スクリプト言語であり，NQS [12] や SGE [7]，Torque [1] などのバッチスケジューラ環境におけるジョブ投入および終了待ちをそれぞれ submit 関数，sync 関数により行うことができる．エンドユーザは，バッチスケジューラごとのインターフェースの違いによらず（導入されていない研究室のクラスタ環境等においても），同一の Xcrypt プログラムを動かすことができ，ジョブスクリプト生成等の煩雑な作業からも解放されるため，手軽にジョブ並列処理を記述することができる．

Xcrypt におけるジョブは，ジョブクラスという組み込みクラスのオブジェクトとして抽象化されている．Xcrypt は，このジョブクラスを拡張することで自身の機能をモジュラーに拡張できる機構を備えている．クラス拡張の実装においては，通常のメンバ，メソッドの追加以外に before，after という特別な名前のメソッドを定義することで，それぞれ非同期に実行されるジョブ実行の前処理，後処理を追加することができる．また，watch という名前のメソッドを追加することで，バックグラウンドで定期的に実行される処理を追加することができる．

本ツールは，このような Xcrypt モジュールの 1 つとして実装した．

3.2 モジュールの定義

before メソッド，after メソッド，watch メソッドの定義をそれぞれ図 4，図 5，図 6 に示す．before メソッドでは分割関数 divide の呼び出しおよび生成された分割ジョブの投入，after メソッドでは統合関数 merge の呼び出しや，同等の計算を行っている再分割されたジョブが存在しないかのチェック（存在した場合は中断）を行う．

watch は，ユーザにより定義された経過時間を過ぎても実行中の部分問題ジョブがあり，計算資源がアイドルであるならば，そのジョブを再度分割してジョブとして実行する．この際，元のジョブを停止するかどうかは，テンプレートオブジェクトのメンバとして定義された値に従うものとする．

```
sub before
{
  my $self = shift;
  $kill_parent = $self->{kill_parent};
  $redivide = $self->{redivide};
  if( defined($self->{divide_func}) )
  {
    $divide_func = $self->{divide};
    #ユーザ定義関数を呼び出してテンプレートの列を得る
    my @children_ = &{$divide}($self, $self->{ndivide});
    my @children = ();
    foreach my $child (@children_)
    {
      %{$child}->{generation} = 1;
      %{$child}->{redivide_on} = $self->{redivide_on};
      %{$child}->{redivide_exit_on} = $self->{redivide_exit_on};
      push(@children, &prepare_submit(%{$child}));
    }
    push(@dc_jobs, \@children);
    @{$self->{children}} = @children;
    $self->{exe0} = 'sleep 1 #';
  }
}
```

図 4　分割統治モジュールの before 関数の定義

```
sub after
{
  my $self = shift;
  if(!defined($self->{del_flag}) and defined($self->{dynamic_children}))
  {
    foreach my $c (@{$self->{dynamic_children}})
    {
      if(defined($c->{finish_time})){next;}
      # 正常終了したジョブに実行中の子ジョブが存在するときはジョブを停
止する
      if( &jobsched::get_job_status($c) eq 'queued' or
       &jobsched::get_job_status($c) eq 'running' )
      {
        $c->{del_flag} = 1;
        &jobsched::qdel($c);
      }
    }
  }
  if( defined($self->{merge}))
  {
    # ユーザ定義関数を呼び出して結果を結合する
    &{$self->{merge}}($self, \@dc_jobs);
  }
}
```

図 5　分割統治モジュールの after 関数の定義

4　評価

2 章で例として取り上げた相同性探索と，SAT ソルバの既存の逐次実装の一つである Minisat [5] を用いて本ツールの性能評価を行った．後者の例は，負荷が均等になるような問題の分割が困難な問題の例である．そのため，動的負荷分散（`redivide` オプション）を有効にすることで実行時間が改善されるかを評価した．

評価には，京都大学の T2K オープンスーパコンピュータを用いた．評価環境の詳細を表 1 に示す．Xcrypt の機能により，(分割された）各ジョブに対応して 1 つの NQS ジョブが（`qsub` コマンドで）投入される．本評価に用いた NQS キューには，この評価用のジョブ以外は投入されないようにした．

表 1　評価環境

CPU	AMD Quad Core Opteron 8350 Barcelona 2.2GHz×4（計 16 コア）
メモリ	32GB
OS	Redhat Enterprise Linux AS V4
バッチスケジューラ	富士通 NQS [12]

4.1　相同性探索

評価に用いた問題は，クエリ数 2496 件，データベースのレコード数 2500 件で逐次実行で 7070 秒かかるものである．分割した各問題の負荷は，キューの数にほぼ比例すると考えられるため，動的負荷分散オプションは無効にした場合のみを評価した．分割数を 2，4，8，16，32，64 として得られた，逐次実行に対する速度向上のグラフを図 7 に示す．分割数が 32 以下のときは，実行時間が十分長いため，問題の分割や結果の統合のオーバーヘッドはほとんど問題にならず，分割数に応じてほぼ理想的な速度向上が得られている．しかし，分割数 64 のときはやや速度向上が制限された．これは分割された各ジョブの問題サイズが小さくなり，分割・統合のオーバーヘッドや各ジョブの負荷バランスの崩れが顕在化したためであると考えられる．

4.2 SAT

Minisat での評価には，ベンチマークとして SAT Competition[6] において出題された問題のうち，Crafted（人工的に難しくした問題），Random（ランダムに生成した問題），Application（実応用に現れる問題）に分類されるものの一部を用いた．

Minisat の入力は，論理式 $f(x_1, x_2, \ldots, x_n)$ を所定の書式で表現したテキストファイルである．問題の分割は，論理式中で最も出現回数が多い変数 x_i を選択し，$f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$ および $f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ をそれぞれ表現するテキストファイルを生成することで行った．逐次実行，最初に問題を 8 分割し動的負荷分散を無効にした場合，最初に問題を 8 分割した後再分割も行った場合（`redivision` 有効，`kill_parent` は無効）のそれぞれの実行時間を問題ごとに測定した結果を図 8 に示す．

SAT な（論理式を満足する割り当てが存在する）問題においては分割したジョブの 1 つが解を偶然発見することがあるため，分割により極端に実行時間が短縮する場合がある．

`kill_parent` を無効にしているので，計算中の問題を再分割した場合においても分割前の計算もそのまま継続して，どちらか早く終了したほうの結果を採用する．そのため，再分割を無効にした場合よりも実行時間が長くなることはない．UNSAT な問題で最初の分割におけるジョブ間の負荷の違いが大きかった場合においては，再分割により実行時間が大きく改善された．例えば，8:Application (UNSAT) においては 1.44 倍（1312s/1002s）の速度向上が得られた．

5 議論

本ツールの機能として提供するジョブ分割自体は，手動で行ったとしてもそれほど大きな手間ではない．本ツールの有用性は，Xcrypt をベースに用いることでバッチキューイングシステム環境に対応したことと，動的負荷分散機能を提供していることにある．

予め計算資源数より多い数の部分問題に分割しておけば，動的負荷分散機構は不要であるとも考えられる．しかし，分割数が少ないと負荷不均衡が起こりやすく，分割数を増やしすぎると分割のオーバーヘッドが大きくなるというトレードオフを解決し，適切な分割数を実行前に決定しておくのは困難である．

Cilk 言語 [2] では，Lazy Task Creation [4] という手法により分割のコストをタスクスティール時まで遅延することで上記の問題を解決しているが，プログラムの書き換えが必要であるため，本ツールで採用することはできない．

Tascell [3] では，ワーカは最初は逐次計算を行い，分割が必要になると初めて自らの残りの計算の一部をタスクとして生成する．本ツールの手法は，Tascell の手法において残りの計算の一部を並列に実行するのではなく，これまでの計算を取り消した上で並列に実行するようにしたものと理解することができる．

6 まとめと今後の課題

本研究では，並列スクリプト言語 Xcrypt を用いてジョブレベルの並列実行ツールを開発した．また，空き計算資源がある場合に実行中のジョブを分割して最初から並列に実行しなおすという動的負荷分散手法により負荷不均衡な問題に対応した．これにより，ユーザは問題の分割方法と統合方法を記述するだけで，簡易的な計算の並列化を行うことができ，並列計算資源を有効に活用することができる．今後の課題としては記述コストの評価が挙げられる．また，本論文では基本的な性能評価しか行わなかったため，今後，動的負荷分散機構なしで分割数を変えて実行した場合，より高並列な環境での評価を行いたい．

謝辞 本研究の一部は，文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」の支援による．

参 考 文 献

[1] Cluster Resources Inc.: TORQUE Resource Manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`.

[2] Frigo, M., Leiserson, C. E., and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices (PLDI '98)*, Vol. 33, No. 5(1998), pp. 212–223.

[3] Hiraishi, T., Yasugi, M., Umatani, S., and Yuasa, T.: Backtracking-based Load Balancing, *Proceedings of the 14th ACM SIGPLAN Sympo-*

sium on Principles and Practice of Parallel Programming (PPoPP 2009), February 2009, pp. 55–64.

[4] 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(1991), pp. 264–280.

[5] Niklas Eén, N.: Minisat, 2009. `http://minisat.se/`.

[6] SAT09 Conference: SAT Competition, 2009. `http://www.satcompetition.org/2009/`.

[7] Sun Microsystems Inc.: The Grid Engine project. `http://gridengine.sunsource.net/`.

[8] 平石拓, 安部達也, 三宅洋平, 岩下武史, 中島浩: 高生産並列スクリプト言語 Xcrypt の開発, 情報処理学会夏のプログラミングシンポジウム *2009* 報告集, 2009, pp. 67–73.

[9] 平石拓, 安部達也, 三宅洋平, 岩下武史, 中島浩: 柔軟かつ直観的な記述が可能なジョブ並列スクリプト言語 Xcrypt, 先進的計算基盤システムシンポジウム（*SACSIS2010*）, 奈良市, 2010, pp. 183–191.

[10] 平石拓, 岩下武史, 中島浩: シームレスな高生産並列スクリプト言語の実現に向けて, 第 *174* 回計算機アーキテクチャ・第 *119* 回ハイパフォーマンスコンピューティング合同研究発表会（*HOKKE-2009*）, 札幌市, 2008, pp. 175–180.

[11] 東京大学情報基盤センター, 筑波大学計算科学研究センター, 東京工業大学学術国際情報センター, 京都大学学術情報メディアセンター, PC クラスタコンソーシアム: クラスタシステム上での並列プログラミングコンテスト, 2009. `https://www2.cc.u-tokyo.ac.jp/procon2009-1/`.

[12] 富士通株式会社: HPC ミドルウェア Parallelnavi. `http://jp.fujitsu.com/solutions/hpc/products/parallelnavi.html`.

```
sub watch
{
  if( $redivide_flag == 1 && defined(@dc_jobs))
  {
    for(my $i = 0; $i < scalar(@dc_jobs); $i++)
    {
      if(defined($dc_jobs[$i])){
        foreach my $c (@{$dc_jobs[$i]})
        {
          if( ! defined($c->{finish_time}))
          {# 再分割時間を超えた実行中のジョブを再分割する
            if( ! defined($c->{dynamic_children})
            and (time() - $c->{start_time} > $redivide_time ))
            {
              my @children_ = &{$divide}($c, 1);
              my @children = ();
              foreach my $child (@children_)
              {push(@{$c->{dynamic_children}},&prepare_submit(%{$child}));}
              my $all_children = get_all_children();
              if( ! defined( @{$all_children}[$c->{generation}] ) )
              {@{$all_children}[$c->{generation}] = \@children;}
              push(@children, \@{$c->{dynamic_children}});
              # 再分割時に分割前のジョブを停止する
              if( $kill_parent == 1)
              {
                if( &jobsched::qstat($c) eq 'queued' or &jobsched::qstat($c) eq 'running' )
                {$c->{del_flag} = 1;&jobsched::qdel($c);}
              }
            }
            # すべての子ジョブが終了している場合親ジョブを停止
            elsif( defined($c->{dynamic_children}) )
            {
              my $flag = 1;
              foreach my $child (@{$c->{dynamic_children}})
              {if(!defined($child->{'finish_time'})){ $flag = 0; }}
              if($flag ==1)
              {
                $c->{finish_time} = time();
                &jobsched::sync(@{$c->{dynamic_children}});
                $c->{del_flag} = 1;
                if( &jobsched::qstat($c) eq 'queued' or &jobsched::qstat($c) eq 'running' )
                {$c->{del_flag} = 1;&jobsched::qdel($c);}
              }
            }
          }
        }
      }
    }
  }
}
```

図 6　分割統治モジュールの watch 関数の定義

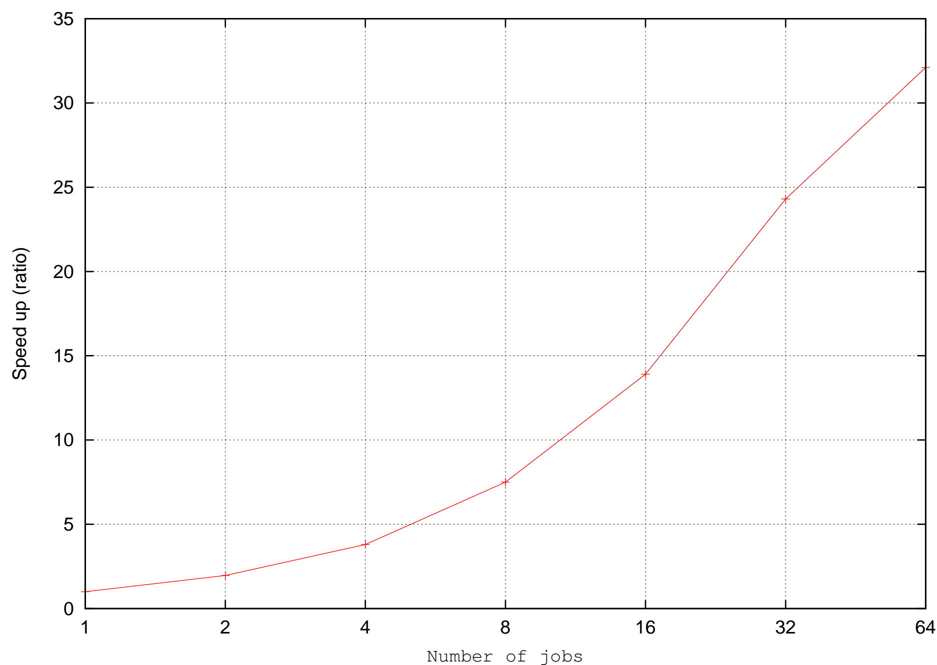

図 7　相同性探索におけるジョブ分割実行による逐次実行に対する速度向上

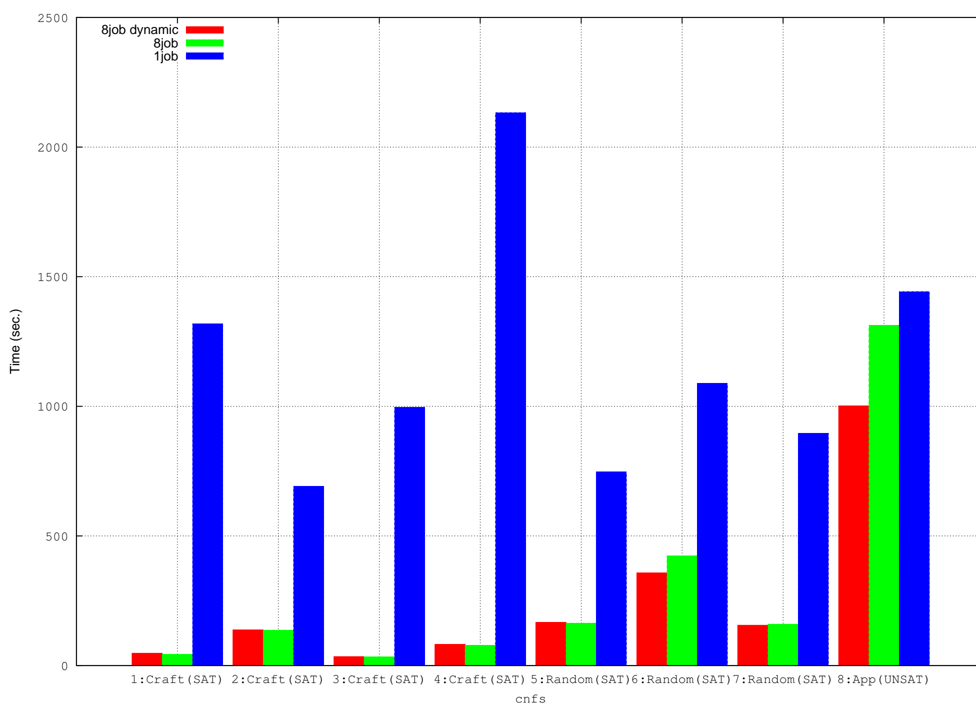

図 8　SAT の各問題ごとの並列度別の実行時間