論 文

Technical Papers

今すぐ使える並列処理

– GXP と Ibis による並列プログラミング –

Easy achievement on parallel processing

with GXP and Ibis

横山 大作
Daisaku Yokoyama

東京大学新領域創成科学研究科
School of Frontier Sciences, The University of Tokyo
yokoyama@logos.k.u-tokyo.ac.jp, http://www.logos.ic.i.u-tokyo.ac.jp/~yokoyama/

keywords: parallel programming, interactive shell, divide-and-conquer

1. はじめに

近年，計算機は急激に低廉化，高性能化しており，複数台の計算機を手元に置いている方も多いことと思われる．数台から数十台程度の規模の PC クラスタを運用されている方も多いだろう．また，最近ではノート PC にすらデュアルコア CPU が乗るようになっており，数 CPU 程度の共有メモリ並列マシンも身近なものとなっている．このように，我々の身近には並列計算ができる環境がごく当たり前のものとして整い始めている．大量のデータ処理，計算量の大きなアルゴリズムを用いた計算などを，これらの並列計算環境を用いて高速に実行したいという欲求は当然である．

実際の場面でどのような計算に時間がかかるのか考えてみると，データのふるい分け，パラメータスイープ，n-fold 検定，機械学習などなど，個々に独立した大量の仕事が存在するような計算を行っている場合も多い．これらを複数の計算機に分担させることは「理論上は」簡単である．しかし，実際には面倒な作業が必要になる．例えば，10 台の計算機の全てで，与える引数の一部分を変更しながらスクリプトを実行したい，という場合に，10 個の対話シェル窓を立ち上げてそれぞれにコマンド文字列をマウスでペースト，などという作業をしようとしてはいないだろうか? あるいは，昨日までは node10 から node19 までを使ってよかったが，今日から node30 以降を使ってくれ，と管理者から要求されたとき，スクリプトの中身をいちいち書き直したりはしていないだろうか? このような場面において効果を発揮するのが，並列対話型シェル GXP[Taura 04] である．GXP は多数の計算機に一度にログインし，それらの上で一斉にプロセスを立ち上げたり，出力結果をまとめたりといった作業を簡単に，スピーディに実現できる対話シェルである．前述のような面倒な作業でお困りの方には，試してみる価値のあるツールである．

また，多数のタスクを処理する際により複雑な制御が必要となる場合，並列プログラミング言語を用いて自分で並列実行用のプログラムを書く必要に迫られることもある．並列プログラミングのためには様々な計算モデル，プログラミング言語が用意されている．pthread ライブラリ (POSIX 標準準拠のスレッドライブラリ) によるマルチスレッドプログラミング [Kleiman 98]，MPI[mpi] によるメッセージパッシングなどは有名であるが，これらを使いこなすためには多くの知識が必要になることが多く，プログラミングを専門としない方が気軽に使うにはやや敷居が高い部分も多い．行列計算などの科学計算分野であれば，比較的扱いやすいフレームワークやライブラリが存在しているが，人工知能分野ではより非定型的な処理を必要とするアルゴリズムも多い．そのようなプログラムのうち，ID3 などの決定木学習のような divide-and-conquer 型のアルゴリズムを対象に，逐次プログラムとほぼ同様の記述を行うだけで並列計算を可能にする Satin[Nieuwpoort 00] という言語処理系が存在する．Satin は Java を拡張した並列プログラミング言語処理系であり，Ibis というより大きな並列計算用パッケージの一部である．

本稿では，GXP のインストールから簡単で実用性の高い並列処理を実現するまでを前半部分で紹介する．また後半は，Ibis をインストールし，Satin を用いて分枝限定法 (branch-and-bound method) などの並列プログラムを書くまでを示すことにする．

2. GXP

GXP は，Grid と呼ばれる多数の計算機が複雑なネットワークで接続された環境を想定して構築された並列対話シェルである．多数の計算機を，わずかな前準備と設定のみで，レスポンスよく制御できる点に大きな特長がある．

2·1 インストールの前に

GXP が動作するためには，GXP を起動する手元のホスト (以下「起動ホスト」と呼ぶ) と，ログインして使いたいホスト群 (以下「遠隔ホスト」) の全てで，

- Unix プラットフォーム (相当)
- Python インタプリタ (Ver. 2 以上)

の 2 つが動作していることが必要である．最近の Linux のディストリビューションであれば，Python インタプリタ [pyt] は最初からインストールされていることが多い．FreeBSD などの他の Unix プラットフォームでも Linux と同様の手順でインストールして使うことができる．また Windows でも，Unix 環境を再現するツール群である cygwin[cyg] をインストールすることで GXP を (一部制限はあるが) 使用することが可能になる．

また，遠隔ホストへは ssh を用いてログインできるようになっていなければならない．Unix マシンならば当然備わっている機能であるが，Windows マシンの場合には cygwin の ssh/sshd をインストールする必要がある．

GXP のシステムをインストールする必要があるのは，起動ホストのみである．遠隔ホストにはログインした際に GXP が自動的にシステム自身をコピーして起動するので，遠隔ホスト側に何かを前もって準備する必要はない．なお，GXP のシステムがコピーされるのは~/.gxp_tmp ディレクトリ内であり，遠隔ホスト側に永続的に GXP を勝手にインストールしてしまう，というわけではないので安心していただきたい．

以下，Linux と Windows の 2 つの場合におけるインストール方法を紹介する．

2·2 Linux へのインストール

GXP は紹介サイト [GXP] からダウンロードできる．本稿作成時の最新リリース版は version 2.0.4 である．gxp-2-0-4.tar.gz をダウンロードし展開すると，gxp-2-0-4 というディレクトリの中にシステム一式が展開される．以下では，gxp-2-0-4 のディレクトリを gxp という名前のディレクトリであるとして説明する．

gxp ディレクトリの中の gxp.py というファイルが GXP 本体である．gxp ディレクトリ中に gxp.sh という起動スクリプトのひな形があるので，この中身を

```
#!/bin/bash
exec python /<install dir>/gxp/gxp.py "$@"
```

と環境に応じて書き換え，ファイル名を gxp と変更し，実行属性を付けて PATH の通った適当なディレクトリ (/usr/local/bin など) に置いておくと，

```
$ gxp
```

とタイプするだけで GXP が立ち上がるようになる．

2·3 Windows へのインストール

Windows の場合，cygwin 環境が必要になる．cygwin のインストールは公開サイト [cyg] よりインストーラをダウンロードし，その指示に従えばよい．

Python は Windows 用のインストーラも存在するが，cygwin 環境内にインストールされる Python を用いた方が GXP の動作は安定しやすい．両者ともインストールされていても，cygwin 側のデフォルト状態では cygwin 内の python インタプリタを使うはずであるので問題はない．

Windows を起動ホストとして使用するときには，Linux の場合と同様に GXP 本体をインストールする．Windows を遠隔ホストとして使用するためには，cygwin の sshd をサービスとして立ち上げ，ログインできるようにする必要がある．cygwin の cygrunsrv と openssh の 2 つのパッケージをインストールし，ssh-host-config コマンドを実行すると sshd 立ち上げの準備ができる．/etc/sshd_config を環境に合わせて設定し，sshd をサービスとして立ち上げれば Windows マシンに外部から ssh でログインすることが可能になる．詳細については cygwin のドキュメントを参照していただきたい．なお，WindowsXP ではデフォルトの状態でファイアウォールが動作しており，sshd の使用するポート (22 番) を使用できるよう設定する必要があることに留意されたい．

2·4 ssh の設定

GXP を起動する前の準備として，遠隔ホストへ ssh でログインする際，パスワードなどを聞かれないでログインできるように設定しておく必要がある．これは具体的には，公開鍵による認証で起動ホストから遠隔ホストへログインできるように設定しておかなければならない，ということである．起動ホストで公開鍵を作り，それを全ての遠隔ホストにあらかじめ配っておく，という方法が一番わかりやすいやり方だろう．認証時にパスフレーズによる暗号化を用いている場合は，パスフレーズを一時的に記憶する ssh-agent というツールを用いればよい．詳しくは ssh の公開鍵認証に関する設定ドキュメントや解説ページ [Robbins] などを参照してほしいが，

```
localhost$ ssh remotehost.remote.domain
remotehost$
```

のように，パスワード，パスフレーズなどを入力しないでログインできる状態になっていれば，GXP のための準備は完了である．

2·5 GXP の起動と，簡単なコマンド実行

gxp を起動すると，以下のようなプロンプトが表示される．

```
$ gxp
GXP started on home.domain.name
GXP[1/1/1] %
```

これは GXP がコマンド入力待ちをしている状態である．終了するには quit と打つか，単に Ctrl-D を入力すればよい．

ここで，GXP を使い始める前の留意点であるが，GXP

のコマンド入力部は必要最低限の機能しか持っておらず，コマンド履歴の呼び出しや編集などはできない．そのため，Emacs の shell-mode などの中で GXP を起動することを強くおすすめする．

さて今，node000.cluster.net というホストにログインし，2 台で計算を行いたいとする．このとき，以下のように打ち込む．

```
GXP[1/1/1] % edges -> node000.cluster.net
0.005 sec
GXP[1/1/1] % explore
home : reached
node000 : reached
0.965 sec
GXP[2/2/2] %
```

最初の edges コマンドで，node000 というホストが存在し，そこへログインして使いたい，ということを GXP に登録する．次の explore で，実際に node000 へのログインを行っている．ここでは無事にログインに成功した．GXP プロンプトの 3 つ組の数字のうち，一番右の数字が現在ログイン中のホスト数を表しており，explore の後は home と node000 の 2 個のホストが使用可能になったことがわかる．

それでは，2 つのホストでコマンドを実行してみる．

```
GXP[2/2/2] % e hostname
node000
home
0.144 sec
GXP[2/2/2] %
```

e というのが GXP のコマンドであり，e 以降を「通常のシェルコマンド」として使用する全てのホストで一斉に実行するものである．ここでは，全てのホストで”hostname” コマンドを実行している．それぞれのホストでのコマンド実行結果，node000 と home という文字列が集められて表示されている．この実行では，起動ホストと遠隔ホストの区別はない．

なお，e と同様の文法の l というコマンドがあり，以降のシェルコマンドを「起動ホストのみで」実行する．

さて，次に node0XX.cluster.net という複数台のホストにログインしてみる．

```
GXP[2/2/2] % edges -> node001.cluster.net
GXP[2/2/2] % edges -> node002.cluster.net
```

のように一つずつ追加していくことも可能であるが，名前に規則性がある場合は

```
GXP[2/2/2] % cluster node0??.cluster.net
```

のようにすると一度に設定することが可能である．cluster コマンドを使うと，GXP が「知っている」ホストのうち，node0??.cluster.net というパターンにマッチするもの全てを使用するように GXP に登録される．

GXP は，edges でユーザに指示されたホストだけでなく，計算に参加しているホストが/etc/hosts や NIS によって知っているホスト名も収集して記憶している．cluster コマンドでは，このようにして記憶したホスト名からのパターンマッチを行っている．

なお，ここでのログインは，起動ホストから遠隔ホストそれぞれへ順番にログインを行う，というものではない．GXP は edges や cluster コマンドで「使用したい」とユーザに指定されたホストについて，tree 状の経路に沿ってログインを試みる．つまり，起動ホストは限られた数のホスト (子ホストとする) にまずログインし，子ホストそれぞれからさらに孫ホストへとログインしていく，という動作を行う．この経路の選定は GXP が自動的に行い，ログイン前に計画されている．explore を行う前に `show plan` コマンドを実行すると，これからどのようにログインしようと GXP が計画しているのか，がわかる．

```
GXP[1/1/1] % show plan
explore/duplore will reach 6 nodes. 5 are new
node000
 --> node001.cluster.net
  --> node002.cluster.net
  --> node005.cluster.net
 --> node003.cluster.net
 --> node004.cluster.net
GXP[1/1/1] %
```

例えばこの実行例では，起動ホスト node000 から node001，3，4 へとログインし，001 からさらに 002，5 へとログインしようと計画されている．ただし，この tree 構造をユーザが意識する必要はない．

cluster コマンドで指定した後で explore を行うと，

```
GXP[2/2/2] % explore
<... 中略...>
Failed logins:
  node008.cluster.net -> node037.cluster.net
9.927 sec
GXP[65/65/65] %
```

のように，獲得したかったホストにログインが行われる．ログインは前述のように tree 状の経路で実行され，かつ複数ログインが並行して行われるため，多数のノードを獲得する場合でも数秒から 10 秒程度でこの過程は終了する．また，この図では node037 が故障中のためにログインに失敗している．このような場合でも，GXP は全体の動作を止めることなく，可能な限りのホストにログインする．explore の時にあらかじめ計画されていたログイン経路は explore の実行中は変化しないので，node037 経由でログインしようと予定していたホスト群にはログインされない．このような場合には，さらにもう一度 explore を行うと，ログインできなかったホスト群に対して他の経路を経由するように自動的に計画が変更されるため，最終的には稼働中のホスト全てにログインすることができる．

なお，共有メモリ並列計算機で，複数のシェルを一度に立ち上げることも可能である．4CPU の計算機で 4 個のシェルを立ち上げるときは

```
GXP[1/1/1] % dup 4 local_hostname
GXP[1/1/1] % duplore
GXP[4/4/4] %
```

という手順をとる．詳細は GXP サイトのマニュアルを参照されたい．

2·6 実行ホスト選択

```
GXP[65/65/65] % e hostname | grep node04
node043
node046
<... 中略...>
GXP[10/65/65] %
```

上図では，全てのホストで”hostname | grep node04” の実行が行われている．node04?という名前以外のホストではこのコマンドは失敗する．実行が成功するのは，node04?という名前を持つ 10 個のホストのみであり，それぞれのホストから node04?という出力が 1 行ずつ行われて GXP のシェルに現れている．

GXP プロンプトの 3 つ組の数字のうち，一番左の数字は，直前のコマンド実行が成功したホストの数を示している．ここでは成功は 10 個のみである．

```
GXP[10/65/65] % smask
GXP[65/10/65] % e hostname
node041
<... 中略...>
GXP[10/10/65] %
```

1 行目，smask というコマンドは，直前にコマンド実行が成功したホストのみを，それ以降のコマンド実行用のホストとして「選択」する．GXP プロンプトの 3 つ組真ん中の数字は，コマンド実行用に現在選択されたホストの数を表しており，smask によって 65 ホストから 10 ホストに選択が絞られたことが分かる．この状態で再び e コマンドを実行すると，今度は選択された 10 ホスト上でのみ hostname が実行されることになる．

smask によるホスト選択を繰り返し，さらにホストを絞り込んでいくことが可能である．“smask -” を実行した場合は，成功と失敗が反転され，直前に実行失敗したホストのみが選択される．絞り込みをやめ，全てのホストが選択された状態に戻すには”rmask” を用いる．

2·7 高度なコマンド実行

一斉実行の e コマンドは，次のような文法を持つ．

```
e commandL {{ commandC }} commandR
e commandC
```

commandL，C，R は，パイプやリダイレクトなどを含む通常のシェルコマンドである．1 行目の書式の場合，commandL と commandR は起動ホストで実行され，commandC は現在選択されている全てのホストで一斉に実行される．commandL の標準出力は複製されて全てのホスト上の commandC の標準入力に接続される．commandC の標準出力は行を単位として非決定的にマージされて，起動ホスト上の commandR の標準入力に接続される．複数の commandC の出力が行の途中で混ざることはない．commandL と R はそれぞれ省略することが可能であり，両方ともに省略した場合は 2 行目の書式のように略記することが可能になる．これまでに見てきた実例は，この略記法による実行であった．

```
GXP[65/65/65] % e {{ uname -r }} sort | uniq
2.6.8-2-386
2.6.8-3-686-smp
```

例えばこの実行例では，全てのホストの uname -r の結果を集め，何種類の出力があったのかを調べようとしている．

2·8 簡単な並列処理

ここまでの説明で，全てのホストに同じ情報を配り，同一のプログラムを実行して，結果を集めることができるようになった．最も簡単な並列処理が可能になったことになる．

それぞれのホストで少しずつ異なる処理をさせるためには，GXP が設定する環境変数を用いると簡単である．代表的な環境変数を以下に示す．

- GXP_NUM_EXECS: 実行選択されているホスト数
- GXP_EXEC_IDX: 実行選択されているホスト内で自ホストに付けられた番号．(0, 1, ... GXP_NUM_EXECS - 1) の範囲．
- GXP_NUM_HOSTS: ログインされているホスト数
- GXP_HOST_IDX: ログインされているホスト内での自ホストの番号

タスク数がわかっていれば，タスク番号の GXP_NUM_EXECS による剰余を取り，GXP_EXEC_IDX と一致したタスクのみを処理する，といった戦略を採ることが可能になる．

タスクの処理時間が全て同じで，計算に参加しているホストの能力も均質なときは，このような静的な均等分割でうまく負荷が分散される．しかし，タスクの処理時間が大きく異なる場合などには，マスタワーカ型の処理を行わせると効率のよい並列計算ができる．

マスタワーカ型とは，マスタが全てのタスクを管理し，ワーカは自分が暇になるとマスタのタスクを一つずつもらって処理する，という動作を繰り返す並列処理手法である．このような処理を簡単に実現するために，GXP には

```
mw commandM {{ commandW }}
```

というコマンドが用意されている．commandM はマスタに相当するコマンドとして起動ホスト上で実行され，commandW はワーカであり，選択された全てのホスト上で実行される．commandM の標準出力は commandW の標準入力にブロードキャストされ，commandW の標準出力はマージされて commandM の標準入力に接続される．

mw コマンドを用いると，マスタワーカ型の処理は例えば以下のように簡単に実現できる．

```
マスタ:
while (1) {
  標準入力から 1 行，ワーカの ID を読み込む
  if (タスクがなくなっていた)
    全てのワーカ ID がそろうまで待ち，
    終了を表す行を出力して終了
  else
    タスクを 1 つ取り出し，
    (ワーカ ID，タスク) ペアの行を標準出力に書き出す
}
```

```
ワーカ:
ワーカ ID を標準出力に書き出す
while (1) {
  標準入力から 1 行，(ワーカ ID，タスク) ペアを読み込む
  終了指示だったら終了
  自分のワーカ ID の行でなければ無視
  自分のワーカ ID だったら，タスクを処理し，
    処理が終了したらワーカ ID を書き出す
}
```

このように，GXP を用いることで，独立した多数の仕事を複数の計算機に適切に分散させて並列計算する，という処理が極めて簡単に実現できるのである．

3. Ibis

Ibis は Java をベースにした並列計算のためのプログラミング環境であり，メッセージ送受信や Remote Method Invocation など，多くの機能を提供している．Satin は Ibis のパッケージに含まれるプログラミング言語処理系であり，Ibis の基盤技術を用いてより高度な計算モデルを提供している．

3·1 インストール

Ibis は全て Java によって実装されているため，計算に参加するホストには全て Java がインストールされている必要がある．Satin プログラムのコンパイルには JDK が必要なため，Sun の Web ページ [jav] より使用 OS に合致した JDK をダウンロードし，指示に従ってインストールを行っておく．Ibis の動作のためには JDK 1.4 以降のバージョンの Java が必要となる．本稿のプログラムは java version 1.5.0_08 で動作することを確認した．

Ibis の配布サイト [ibi] 内，Download のページから，2006 年 9 月 15 日にリリースされたバージョン 1.4 をダウンロードする．このバージョンから，ソース形式ではなく Java のバイナリ形式で配布が行えるように修正が施され，Ibis のインストールが極めて簡単になった．ここでは”Download binary release” を選択して ibis-1.4.zip をダウンロードしておく．

Ibis のインストールは，ダウンロードしてきた zip ファイルを適当なディレクトリに展開し，展開した ibis-1.4 ディレクトリのフルパスを環境変数 IBIS_HOME に設定するだけである．Unix，Windows のどちらの場合もこれだけでよい．また，Java をインストールした際に環境変数 JAVA_HOME は適切に設定されているはずであるが，もしこれが未設定の場合は，Java のインストールされたディレクトリを設定しておく必要がある．

なお，Satin プログラムをコンパイルする際には，Java における make コマンド相当のコマンド，”ant”[ant] をインストールしておくとコンパイルが容易になる．

3·2 Divide-and-conquer の並列計算

Satin が提供するのは，divide-and-conquer 型と呼ばれる種類の並列計算を簡単にプログラミングできるような仕組みである．

```
public long fib(long n) {
    if (n < 2) return n;
    long x = fib(n - 1);
    long y = fib(n - 2);
    return x + y;
}
```

上はフィボナッチ数を求めるプログラムを Java で記述したものである．このアルゴリズムは，fib(n) を求めるために，より小さなタスク fib(n - 1) と fib(n - 2) に分割し，それぞれの結果を求めて後で統合することで，全体の結果を得るという，divide-and-conquer の構造を持っていると考えてもよい．

ここで，fib(n - 1) と fib(n - 2) の計算は独立している．そこで，この 2 つの計算を並行に実行し，全体の計算時間を短縮させることができる．Satin は，このような並列計算を簡単に書けるようなプログラミング言語である．

```
public long fib(long n) {
    if (n < 2) return n;
    long x = fib(n - 1); //[1]
    long y = fib(n - 2); //[2]
    sync();
    return x + y;
}
```

fib を Satin によって並列プログラムとして書き直すと，概念的にはこのようになる．以前のプログラムに対して，sync() というコードが追加されただけである．ここで，fib という関数を「並列に実行できるもの」と Satin に伝えてコンパイルを行う．実行時には，[1] と [2] の fib 関数の呼び出しは，並列に実行される可能性があるもの，としてローカルなワークキューに入れられる．ワークキューに入れられたタスクは 1 つずつ取り出され，計算される．sync() 関数は，それ以前の並列実行タスクの実行が終わるまで待つという動作を行う，Satin が提供する関数である．ワークキューに入れられた [1] と [2] の fib 関数の計算が終わると，sync() 関数での待機が終了し，次の行の実行，[1] と [2] の結果を足し合わせて return する，という計算が実行されることになる．

ある計算機において，ワークキューにタスクがなくなってしまったとする．この時この計算機は，計算に参加している他の計算機にランダムに問い合わせを送り，ワークキューに計算されずに残っているタスクを分けてもらおうとする．このような”work stealing” 機構が働くことで，最初は 1 つの計算機で実行されていた fib() 関数は，次第に他の計算機に「盗まれ」て，並列に計算され

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

図 1 数独の問題例 (Wikipedia[wik] より)

るようになる．work stealing の際には，なるべく古いタスクから盗む，という方式が採用されている．これは，古いタスクの方がより多くのタスクを含んだ，より大きなタスクであることが多い，という divide-and-conquer の特徴を生かすための工夫である．

なお，ここに記述した Satin によるフィボナッチ数のプログラムは動作の概念を示すだけのものであり，実際にプログラムをどのように変更すればよいかは次の章で詳しく説明する．また，このプログラムの完全な並列化版は，サポートページ内に ibis_fib.zip として公開されているので参考にしていただきたい．コンパイルや実行の方法は 3･4 節を参照のこと．

3･3 並列プログラムへの書き換え

この章以降では，少し複雑になった divide-and-conquer プログラムとして，数独 [wik] の解の数え上げプログラムを例に，Satin による並列プログラムへの書き換え方法を示す．

数独 (スウドク) はペンシルパズルと呼ばれる種類のパズルであり，近年世界的に人気が高まっている．数独はニコリの登録商標であるため，ナンバープレースとも呼ばれるが，海外では Sudoku と呼ばれることが多いようである．図 1 に数独の例を示す．9×9 の升目に 1 から 9 の数字を埋めるのだが，同一行内，同一列内，太線で囲まれた 3×3 のブロック内には同じ数字が入ってはならない，というルールである．

ここでは，与えられた問題に対して，数独の制約を満たすような解がいくつ存在するかを，可能解を完全探索することで求めるプログラムを考える (ただし，同一解の排除や対称解の排除は行っていない)．なお，Ibis は Java のクラス継承と interface，及び serialize 機構を利用する．これらの機構については本稿では説明しないので，Java の中級以上を対象とした入門書，解説書など [Arnold 01, Bloch 01] を参考にされたい．

図 2 は，Satin によって並列化された数え上げプログラムである．このプログラムの完全版もサポートページにおいて ibis_sudoku.zip として公開されているので，適宜参考にしていただきたい．search() 関数は，部分的に解か

```
import ibis.satin.SatinObject;
import java.util.*;

interface Searcher
  extends ibis.satin.Spawnable {
  public int search(Board b);
} // [*1]

class Sudoku extends SatinObject
  implements Searcher { // [*2]
  public int search(Board b) {
    if (b が不正な局面になった) { return 0; }
    if (b が完全な解) { return 1; }

    int mpos
      = (b で，数字の可能性が最も絞られた座標)
    Vector<Integer> cand_v
      = (b の mpos に置くことが可能な数字の集合)
    int n_cand = cand_v.size();
    int ret[] = new int[n_cand]; // [*3]

    for (int i = 0; i < n_cand; i++) {
      Board next_b = (Board)b.clone(); // [*4]
      int id = cand_v.get(i);
      next_b.place(mpos, id); //新しい局面作成
      ret[i] = search(next_b); // [*5]
    }
    sync(); // [*6]

    int ret_num = 0;
    for (int i = 0; i < n_cand; i++) {
      ret_num += ret[i];
    }
    return ret_num;
  }

  public static void main(String[] args) {
    int[] initial_board = new int[] {...};
    Board b = new Board(initial_board);

    Sudoku s = new Sudoku();
    int n = s.search(b);
    s.sync(); // [*7]
    System.out.println(n);
  }
}

class Board
  implements Cloneable, Serializable { // [*8]
  ...
}
```

図 2 数独の解数え上げプログラム

れた局面 (Board) を受け取り，その局面から完全な解がいくつ導出されるかを求める関数である．問題局面 b を受け取ると，置くことができる数字の可能性が最も絞られた升目を一つ選び，そこに置ける数字を置いてみた新しい局面 new_b を作って，それぞれを引数に search() を再帰的に呼び出して探索する，という divide-and-conquer の構造を持っている．この関数 search() を並列実行可能であると Satin に伝える必要がある．このためには，並列実行可能な関数 search() を持つクラス Sudoku の継承関係を変更する．具体的には，このような関数を持つクラスは ibis.satin.SatinObject を継承し，ibis.satin.Spawnable を継承した interface を実装したものとしなければならない．[*1] で，実際に並列実行可能な関数をインタフェースとして定義し，Sudoku でこれを実装している [*2].

[*5] で並列に計算された search() の結果が出そろうまで待つために，[*6] の地点に sync() 関数が追加されている．search() が呼び出されてから sync() が終了するまでは，search() の返り値にアクセスすることはできない．これはつまり，

```
int ret = 0;
for (...) {
  ...
  ret += search(next_b);
  ...
}
```

のようなコードは書けないということになる．従って，複数の search() の結果を sync() まで全て別々に保持し続けるために，[*3] で呼び出す search() の数だけの配列を用意し，それに結果を返してもらうようにしている．

また，search() の引数も，sync() までは変わらないことが求められる．

```
for (...) {
  b に数字を置く
  ret[i] = search(b);
  b の数字を元に戻す
}
```

のようなコードを書くと，search() が呼ばれたときには b は変更されてしまっている可能性があり，正しい結果が得られない．そこで，新たに作られた next_b を search() に渡す必要がある [*4]．これは，search() の引数が int など値渡しされる型だけであれば気にする必要はないが，参照渡しされるクラスや配列を使う場合には考慮に入れる必要がある．

main 関数では問題の局面を引数に search() を呼ぶが，この後の [*7] に sync() を入れることを忘れてはいけない．sync() がないと，search() の結果を待たずに main 関数を終了してしまうことになる．

並列実行可能な関数の全ての引数と返り値は，Serializable でなければならない．ここでは，ユーザ定義のクラス Board が，java.io.Serializable を実装する必要がある [*8].

3·4 コンパイルと実行

Satin プログラムのコンパイルのためには ant を用いるのが便利である．

```
<project name="Sudoku" default="build" basedir=".">

<description>
Sudoku solver
</description>

<property environment="env" />
<property name="ibis" value="${env.IBIS_HOME}" />
<property name="satin-classes" value="Sudoku" />
<property name="build" location="build" />

<import
  file="${ibis}/build-files/apps/build-satin-app.xml"
/>
</project>
```

このような build.xml を作る．重要なのは satin-classes という property である．ここには javac によるコンパイル後に Satin が手を加えなければならないクラスの名を全て，カンマ区切りで列挙する．書かなければならないのは，main を含むクラス，並列実行や sync を起こすクラス，shared object(後述) を実装するクラスの 3 種類である．

```
$ ant clean build
```

とすると，build ディレクトリ中に並列実行用の java バイナリが生成される．なお，

```
$ ant clean compile
```

とすると，Ibis を組み込まない逐次実行版のバイナリが生成されるので，デバッグにはこちらを用いると都合がよい場合もあるだろう．

コンパイルされたバイナリの実行には，Ibis に付属するサポート用のスクリプトを使って

```
$ $IBIS_HOME/bin/ibis-run -ns localhost Sudoku
```

とする．(まぎらわしいが，IBIS_HOME の前の$は環境変数へのアクセスを表している) 引数の-ns は，Ibis のネームサーバがあるホストを指定する．ネームサーバとは，同一の並列計算に参加する Ibis プログラムを全て把握し，管理するためのプログラムである．ネームサーバのあるべきホストとして自ホストが指定されると，Ibis は計算プロセスの他にネームサーバも自動的に立ち上げる．ネームサーバ単体を

```
$ $IBIS_HOME/bin/ibis-nameserver
```

であらかじめ立ち上げておくことも可能である．また，ネームサーバは，計算に参加する全てのホストから TCP/IP ソケットで接続が可能なホスト上になければならない．つまり，ファイアウォールの内側にあるようなホストでネームサーバを立ち上げても，外部の計算プロセスは計算に参加できないということになる．ただし，ネームサーバが使用するポートは自由に設定できるので，ファイアウォールのポートを適切に開放すれば動作は可能である．

並列実行時には，複数の計算機において，同一のネームサーバを指定して ibis-run を実行すると，自動的に同

一の計算に参加し，タスクを盗んで並列計算を行うようになる．GXP を用いて起動するのであれば，

```
GXP[65/65/65] % cd /program/dir/
GXP[65/65/65] % e $IBIS_HOME/bin/ibis-run
                  -ns nameserver_name Sudoku
```

である．

3·5 Shared object

ここまでで示したように，Satin を用いると，比較的簡単なプログラムの書き換えを行うだけで，divide-and-conquer 型の並列計算が実現できる．このようなプログラムで十分な場面もあるが，アルゴリズムによっては，全ての計算機で共有されたデータを扱いたい場合もある．

例えば，分枝限定法で巡回セールスマン問題を解く場合を考えてみる．まず，都市間の距離情報を保持したテーブルは比較的巨大なデータを持つため，並列計算する関数にそのまま渡すと，タスクが盗まれるたびにシリアライズされて大量のデータを通信することになってしまう．このテーブルのデータを，プロセスが並列計算に参加するときに一度だけ取得し，あとはそれをプロセス内の全てのタスクが参照するようにすれば，通信コストが大幅に削減される．

また，分枝限定法では，探索途中に発見された暫定的な最良解の値を用いて見込みのない探索を打ち切ることができる．純粋な divide-and-conquer 型の並列計算では，このような暫定解の値を他のプロセスに送ることができなかった．そのため，限定操作を行うことができず，無駄な探索を続けてしまうことになる．

このような問題に対処するために，Satin では Shared object という仕組みが利用可能になっている．図 3 は，Shared object を用いて TSP を解くためのプログラムである．なお，この例題プログラムは [Wrzesinska 05] から引用した．

都市間の距離を保持する DistanceTable は，satin.so.SharedObject を継承して作られている [*1]．これは一度値が決まったら変更されないような Shared object である．DistanceTable は，main 関数で生成され，他の計算機には一度だけ転送される．他の計算機でこのオブジェクトの値を読もうとしたときには，転送されたローカルなオブジェクトの値を読むことになる．

一方，暫定最良解の値を保持する Min は，計算が進行するにつれて変更されていくような Shared object である．このように，書き込みが起きる Shared object については，satin.so.WriteMethodsInterface を継承したインタフェースを定義することで，どの関数で書き込みが起きるのかを Satin に知らせる必要がある [*2]．ここでは，Min.set() が書き込みを行うメソッドである [*3]．

Shared object への書き込みが起きると，その結果は「ベストエフォートで」全てのホストの Shared object に伝えられ，ローカルなオブジェクトの値を更新することになる．複数のホスト間では，更新の順序が違うことも

```
final class DistanceTable
  extends satin.so.SharedObject { // [*1]
  ...
}

interface MinInterface
  extends satin.so.WriteMethodsInterface {
  public void set(int val);
} // [*2]
final class Min extends satin.so.SharedObject
  implements MinInterface {
  int val = Integer.MAX_VALUE;
  public void set(int new_val) { // [*3]
    if (new_val < val) val = new_val;
  }
  public int get() {
    return val;
  }
}

public interface TspInterface
  extends satin.Spawnable {
  public int tsp(int hops, int[] path, int len,
                 DistanceTable dist, Min min);
}
public class Tsp extends satin.SatinObject
  implements TspInterface {
  public int tsp(int hops, int[] path, int len,
                 DistanceTable dist, Min min) {
    int[] mins = new int[NTOWNS];
    if (len >= min.get()) { // [*4]
      // Shared object の値を用いた枝刈り
      return len;
    }
    if (hops == NTOWNS) {
      min.set(len); // [*5]
      return len;
    }
    for (int city = path にない都市) {
      mins[i++] =
        tsp(hops + 1, (city を追加した path),
            len + dist.getDistTo(city),
            dist, min);
    }
    sync();
    return (最小の mins);
  }
}

public static void main(String[] args) {
  DistanceTable dist
    = new DistanceTable(NTOWNS);
  Min min = new Min();
  Tsp tsp = new Tsp();
  int result = tsp.tsp(0, new int[0], 0,
                       dist, min);
  tsp.sync();
  System.out.println(result);
}
```

図 3 Shared object を用いた TSP プログラム

あり得る．Shared object の値の更新は，タスクの実行が中断している安全なポイントのみ，つまり，並列実行可能な関数をタスクとして生成しようとしているとき，sync() でタスクが終了するのを待っているとき，タスクの実行が終了したときの 3 つのタイミングでのみ起きる．そのため，タスク実行中には好きなタイミングで (ロックなしで)Shared object の値を読んでよく，一貫性が失われた状態にはならない．例題プログラムでは，タスクである tsp() 関数内で自由に min を操作している [*4,*5]．

Shared object の更新はこのような緩やかな制約のみを満たすため，効率の良い実装を用いることができる．そして，分枝限定法の場合，このような緩い制約であっても，Shared object の値は安全な枝刈りの為だけに使用されるものであるので，アルゴリズムの計算結果の正当性には影響しない．このような Shared object を用いて，並列計算の効率を高められるような問題は多いと思われる．

なお，Shared object により厳しい制約を加えるための機構も用意されているが，本稿では触れないことにする．

3·6 計算効率向上のために

このように，divide-and-conquer 型やその応用的なアルゴリズムについて，Satin を用いれば手軽に並列プログラムを作成することができる．

作成した並列プログラムの計算効率を向上させるための重要なポイントに，計算粒度の調整がある．これまで見てきたプログラム例では，並列実行可能な関数はどんなに小さいタスクになっていても全て並列実行できるように設定していた．しかし，あまりに小さいタスクを盗んでいくと，計算時間に対して通信や同期にかかる時間が無視できなくなり，並列計算の効率が悪くなってしまう．タスクの大きさがある程度見積もれるようなプログラムであれば，ある程度小さいタスクについては，並列実行しない関数，つまり ibis.satin.Spawnable のインタフェースを実装していない通常の関数を呼ぶようにして，並列実行しないような工夫をするとよい．ただし，タスクの大きさを大きくしすぎると，今度はタスク総数が少なくなり，暇な計算機がタスクを見つけられず，計算効率が下がってしまう点にも注意が必要である．

4. お わ り に

本稿では，GXP を用いて多数の独立したタスクを簡単に複数の計算機に分散させて実行する方法と，Satin を用いて divide-and-conquer 型のアルゴリズムを簡単に並列プログラミングする方法を示した．

GXP は対話シェルであるので，並列計算のためだけではなく計算機クラスタなどの管理業務にも有用であることを申し添えておく．例えば，全ての計算機のログファイルから検索を行う，全ての計算機におかれたファイルを新しいものに置き換える，など，管理業務には GXP が得意とする操作が多い．

また，Satin の提供する並列計算の枠組みだけでは機能が不十分なアルゴリズムも存在する．Ibis にはより低レベルなプリミティブである RMI や，MPI のような計算モデルを提供する MPJ などのライブラリも含まれているので，必要に応じて使い分けていただきたい．

♢ 参 考 文 献 ♢

[ant] Apache Ant. http://ant.apache.org/

[Arnold 01] Arnold, K., Holmes, D., and Gosling, J.: プログラミング言語 Java, ピアソンエデュケーション (2001)

[Bloch 01] Bloch, J.: Effective Java プログラミング言語ガイド, ピアソンエデュケーション (2001)

[cyg] Cygwin Information and Installation. http://www.cygwin.com/

[GXP] GXP GRID/Cluster Shell. http://www.logos.t.u-tokyo.ac.jp/phoenix/gxp_quick_man_ja.shtml

[ibi] Ibis: Efficient Java-Based Grid Computing. http://www.cs.vu.nl/ibis/

[jav] Java Technology. http://java.sun.com/

[Kleiman 98] Kleiman, S., Smaalders, B., and Shah, D.: 実践マルチスレッドプログラミング, アスキー (1998)

[mpi] Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/

[Nieuwpoort 00] Nieuwpoort, van R. V., Kielmann, T., and Bal, H. E.: Satin: Efficient Parallel Divide-and-Conquer in Java, in *Euro-Par 2000 Parallel Processing*, No. 1900 in Lecture Notes in Computer Science, pp. 690–699, Munich, Germany (2000), Springer

[pyt] Python Programming Language – Official Website. http://www.python.org/

[Robbins] Robbins, D.: IBM 共通テーマ: OpenSSH キー (鍵) の管理: http://www-06.ibm.com/jp/developerworks/linux/011019/j_l-keyc.html

[Taura 04] Taura, K.: GXP : An Interactive Shell for the Grid Environment, in *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (2004)

[wik] Sudoku - From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Sudoku

[Wrzesinska 05] Wrzesinska, G., Maassen, J., Verstoep, K., and Bal, H. E.: Satin++: Divide-and-Share on the Grid, Submitted for publication (2005)

〔担当委員：××○○〕

19YY 年 MM 月 DD 日　受理

著 者 紹 介

横山　大作

2000 年東京大学大学院工学研究科情報工学専攻修士課程修了．2002 年同博士課程中退．同年より東京大学新領域創成科学研究科助手．2006 年同大学より博士号取得．博士 (科学)．並列計算量理論，並列プログラミングライブラリ，およびゲームプログラミングに関する研究に従事．情報処理学会，ソフトウェア科学会，IEEE-CS 各会員．