Lisp マシンエミュレータ Celis の並列化 — Bridget ゲームへの適用

天海良治 *　　竹内郁雄 †

概要

1980 年代半ばに開発・市販された Lisp マシン TAO/ELIS の C 言語によるエミュレータ Celis は，最近の Intel や AMD CPU 上ですでに当時の約 330 倍の速度で動作している．当時の技術水準により，エミュレータ部分を含めても使用するメモリは 140MB 程度であり，現在の PC の GB オーダーのメモリには数十台分が十分に入る．最近の PC のマルチコアの性能向上は目覚しく，16 コアのマシンが容易に入手できるようになった．そこで，各コアで並列に Celis を走らせる実装をほとんどシェルプログラミングで行った．これにより，一昨年のプログラミング・シンポジウムで報告した Bridget というゲームの AI を，現時点で 4 台 72 コア（73 並列 Celis）で，4 手先読みで走らせることが可能になった．本報告では並列化 API や実装の工夫などについて述べる．

キーワード Lisp マシンエミュレータ，並列 $\alpha\beta$ 探索，シェルプログラミング，Bridget

1　はじめに

Lisp マシン TAO/ELIS はタグアーキテクチャの専用 VLSI と入出力を担当するフロントエンドプロセッサからなるハードウェアと，50K ステップのマイクロプログラムで記述された Lisp, オブジェクト指向，論理型を融合したマルチパラダイム言語 TAO で構成されている．実機での走行は 20 年ほど前に終焉を迎えたが，マイクロプログラムを C 言語に変換し，フロントエンド部を PC 上の Unix (FreeBSD や Linux) に置き換えたエミュレータでその後も我々のプログラミング環境として生き長らえている．C 言語で書かれた ELIS の意味で Celis と呼んでいる．当時の ELIS の CPU はシングルコア，16MHz のクロックと 128MByte のセル容量，128KByte のハードウェアスタックというスペックであって，最新 PC の CPU で軽々と実行できる．

今回，CPU コアを有効活用すべく，複数の Celis プロセスで並列計算を実行する基本機能を提供するフレームワーク（以下 **para-base** と呼ぶ）を開発したので報告する．TAO 言語で記述した Bridget ゲームの着手の評価タスクを並列に実行する例で報告するが， **para-base** 自体は別の言語処理系を並列化するのにも利用可能である．

2　para-base の概要

言語 TAO は ELIS ハードウェアを前提に設計され，マルチプロセスワークステーションとして単体で動作していた．よって，TAO 自体にはマルチコア上で走る機能はない．また，TAO には TCP/IP スタックは実装されていたが，エミュレータではネットワーク部分の保守を取り止めた[1)] これらの制約から，並列実行に欠かせない排他制御や TAO 間での通信は言語の外部に頼ることとなる．

これらの機能を言語から独立した形で構成し，並列実行アプリケーションを記述するフレームワークとして構成するため，簡潔性を重視して以下のような方針を定めた．

1. 1 台のプライマリと複数のセカンダリのスタートポロジ構成とする．
   プライマリとセカンダリはどちらもアプリケーションレイヤと **para-base** レイヤからなる．プライマリがセカンダリに計算依頼を出して，

* プログラマ amagai@nue.org
† 東京大学名誉教授 nue@nue.org

1) 当初は FreeBSD の BPF, Linux の AF_PACKET, SOCK_RAW でイーサフレームを TAO に取り込むことでネットワークも動作していた

その結果を受け取る Remote Procedure Call モデルを採用する．セカンダリ間の直接の通信は発生しない．

2. プライマリとセカンダリを疎遠にする．
   他の Celis プロセスに関する情報を減らして管理コストを削減し，システムの堅牢性を高める．複数ホスト環境でプライマリが最低限必要とする情報はセカンダリが存在する複数のホスト名とセカンダリの総数のみ（ホストごとの台数はなくてもよい），セカンダリが必要とするのはプライマリ Celis が走行しているホスト名のみである．
3. プライマリとセカンダリ間の通信は shell script 操作によるファイル経由で実現する．排他制御もファイルの書き込みと読み込みを排他的にする Unix コマンドで実現する．言語外にプログラムを別途用意する必要はない．通信データの到着は polling で確認することとする．
4. これらの方針から，タスクの粒度は中〜大とする．

図 1 に概要を示す．

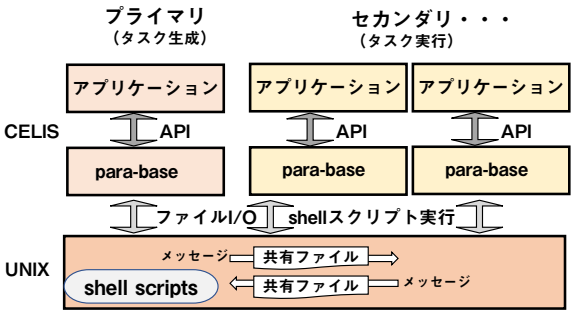

図 1　システム概念図

セカンダリで並列に実行される関数の実行単位をタスクと呼ぶ．セカンダリにタスクを割り当てる，などと言う．タスクを実行するプログラムは事前に用意してセカンダリアプリケーションとして配布しておく．プライマリが用意するタスクはその引数にあたる．実行の環境（Bridget なら盤面など）は同期して着手を打っていくことですべてのセカンダリが同じ状況を各自で構築する．

TAO で実装しているので，S 式で記述した関数本体をセカンダリに送ってそれを実行してもらうことも容易に実現できるが，今回はそこまで動的なプログラムの差し替えは不要なので，引数だけを送る．

以下，まず 1 台のホスト（PC）のマルチコア環境での **para-base** について述べ，次にこれを複数ホストに拡張する（6 章）．

3　para-base の詳細

アプリケーションは，プライマリでタスクを用意して，空いているセカンダリにそのタスク実行を依頼し (dispatch)，返事を受け取ることを繰り返す．タスク数も実行時間も不定なので，この依頼と返事の受け取りは個別に非同期的に実行できるようにする．これと別に，実際に手を打つなど，全てのセカンダリの状態を一致させるといった依頼 (sync) もある．この場合は全セカンダリの依頼実行完了をプライマリが確認する必要があるので，依頼を出したら返事が揃うまで待つ同期的な動作となる．また，すべてのセカンダリ，あるいは都度結成されるセカンダリのグループに共通する情報など，返事を期待しない広報 (Bulletin Board) で十分なものもある．これら，dispatch, sync, BB がプライマリの基本的動作になる．

セカンダリはプライマリからの依頼を受けてタスクを実行する．タスク実行はアプリケーションが用意した **para-base** 内の関数をセカンダリが実行する．

表 1 と表 2 にプライマリとセカンダリの動作をまとめた．プライマリの動作の各タイプごとにセカンダリも同じ対応するタイプのように動作する．次節以降で各々について説明する

表 1　プライマリの動作

タイプ	セカンダリ	ブロック	結果受領
sync	全て †	する	する
dispatch	いずれか 1 個 ‡	しない	polling
BB	全て，またはグループ †	しない	しない

† 全て，またはグループのセカンダリが同じ引数を受け取る

‡ セカンダリは個別の引数を受け取る

表 2　セカンダリの動作（返事）

タイプ	返事
sync	戻り値
dispatch	戻り値 and/or 要望送信
BB	なし

3.1　プライマリの動作

プライマリの動作は対象となるセカンダリ，返事を待つためにブロックするかでタイプがわかれる．sync タイプの動作は全セカンダリに対して同じ動作を同じ引数で依頼する．全セカンダリが動作を完了するまでブロックする．dispatch タイプは 1 つのセカンダリだけを対象とする．アプリケーションは並列に実行させるタスクごとに引数の異なる dispatch タイプの依頼を生成する．依頼の返事は polling で受け取る．dispatch 自体は返事がなくても直ちに戻る．依頼と結果受領を分け，どちらもブロックしない動作とすることで複数のタスクの効率的な実行を実現しているわけである．

BB タイプはプライマリからの情報提供である．動作はブロックしないし返事も必要としないが，セカンダリは必要なアクションをとることがある．

3.2　セカンダリの動作

セカンダリアプリケーションは sync タイプと dispatch タイプの動作用の関数を用意する．セカンダリの **para-base** がプライマリから受け取った引数でこれらの関数を実行する．関数の戻り値がプライマリに返される．sync タイプでは，すべてのセカンダリが同じ引数を受け取る．実行完了するまでプライマリの実行はブロックされる．実行を完了した時点では全セカンダリの状態が一致していることが期待されている．

dispatch タイプでは，実行ごとに引数が異なるし実行時間も異なることが想定される．関数の戻り値はプライマリに送られ，順次受け取られる．dispatch タイプの関数の実行中には要望送信の関数でプライマリにデータを送ることができる．要望送信関数の実行はブロックしない．送ったデータはプライマリに順次受け取られる．

プライマリから BB タイプで送られた情報は，アプリケーション関数の実行中に随時受け取ることができ，必要なアクションを起こす．このアクションは主に探索カットなどの性能向上のためのものである．BB の情報受領はプライマリには通知されない．

表 3　Bridget 用のコマンド

コマンド	説明
プライマリ sync 用 接頭辞`:S-`がつく	
`:S-initial-moves`	初期化
`:S-move`	着手通知
`:S-undo`	着手取消
`:S-stop`	評価打ち切り（8.1.5 節）
`:S-quit`	即刻評価打ち切り（8.1.5 節）
`:S-ping`	問い合わせ (6.1 節)
`:S-expression`	引数の S 式評価
プライマリ dispatch 用 接頭辞`:D-`がつく	
`:D-next-moves`	候補手評価依頼
`:D-contact-moves`	候補手列挙依頼
`:D-reach-cut`	候補手列挙依頼
セカンダリ用	
`:ack`	了解
`:acks`	集約済み了解（8.1.2 節）
`:evaluate`	評価値返答
`:request`	要望
`:nop`	メッセージなしを示す

3.3　動作の実装

sync, dispatch, BB の 3 つの動作の実装について述べる．以下，アプリケーションと **para-base** を区別するときには，プライマリ AP，プライマリ PB，セカンダリ AP，セカンダリ PB と記述する．

プライマリとセカンダリの間はメッセージと呼ぶ形式で通信する．

- メッセージは *seq*（連番）と *body* からなる
- *body* は *command* と引数（なくてもよい）からなる
- *seq* はプライマリ PB が用意する．メッセージの発行ごとに 1 増加する
- *command* は API によって定まる．引数はプライマリ AP が用意する

- *command* は仕事内容の指示の 1 語である．Bridget 用のコマンドを表 3 に示す
- プライマリが送信したメッセージは FIFO キューに追加される．キューはプライマリと全セカンダリで共有される．キューの実体は Unix 上の通常ファイルで 1 メッセージを 1 行の S 式で表わす．ファイル名は `toSecondaries` である
- メッセージには sync 用と dispatch 用の 2 種類がある．種別は *command* によって決まる
  1. sync 用はすべてのセカンダリが同じメッセージを受け取る
  2. dispatch 用はいずれか 1 台のセカンダリだけがそのメッセージを受け取る
- セカンダリからプライマリへの返事も同じ形式のメッセージで別のキュー（ファイル名は `toPrimary`）に返す（ただし *seq* は未使用）．
- キュー追加時は，まず TAO プログラムから Celis プロセスごとに用意された一時ファイル（テンポラリファイル）に書き込み，次に一時ファイルの内容を shell script で排他ロックでファイルに追加する．ファイルロック機能のない TAO から最小のロック期間で安全に書き込める．キューからのデータを取り出しはこの逆に shell script で排他ロックでファイルの先頭行を取り出し，一時ファイルに書き込む．TAO は一時ファイルをゆっくり読むことができる

3.4 プライマリ側

プライマリは次のように動く．

- プライマリ AP は API を呼んでプライマリ PB にメッセージ作成と `toSecondaries` への書き込みを依頼する．
- sync 用メッセージを送る場合，すべてのセカンダリが同じメッセージを受け取るので，プライマリ PB は `toSecondaries` に排他ロックで 1 行だけ書く．*seq* は変わらないので，各セカンダリが 1 回だけ受け取る．
  次に，プライマリ PB は `toPrimary` に ACK メッセージ（*command* が`:ack`）が書き込まれるのを polling で待つ．届いた ACK メッセージの個数がセカンダリ数と同じになったら `toSecondaries` をクリアし，API はリターンする．
- dispatch 用メッセージは返事を待たずに API がリターンする．dispatch 用メッセージは同時に複数書き込んでよい．
- プライマリ AP は dispatch 用メッセージの返事（実行結果）を確認用 API で得ることができる（`toPrimary` から 1 行読み込む）．返事がまだなければ `nil` が返る（TAO の副値でセカンダリが値として `nil` を返したかは分かる）．返事が揃ったかはプライマリ AP が確認する．

3.5 セカンダリ側

すべてのセカンダリは独立に次のように動く．

- セカンダリ PB は排他ロックで以下を実行する（shell script で実行）
  1. `toSecondaries` にデータがあれば先頭 1 行のメッセージを自分用の一時ファイルにコピーする（受け取る）
  2. sync 用メッセージであった場合はメッセージはそのままファイルに残す．
     dispatch 用メッセージの場合はファイルから取り除く（先頭 1 行を削除する）
  3. `toSecondaries` が空ならしばらく待機して再び見にいく（いまは sleep 0.4 秒）
- 受け取ったメッセージの *seq* がすでに受け取ったことのある *seq* ならそのメッセージは無視する．`:nop` コマンドも無視する
- 受け取ったメッセージの *seq* を記録し，*command* の指示に従ってセカンダリ AP の関数を実行する．関数はコマンドごとに定められた純粋仮想関数[2)]の実装としてセカンダリ AP が用意する
- 実行後の動作はメッセージの種類で異なる
  1. sync: セカンダリ PB が 1 個の ACK メッセージを `toPrimary` に書き込む
  2. dispatch: セカンダリ AP が API 経由で 1 個の評価メッセージ（Bridget の場合 *command* が `:evaluate`）を `toPrimary` に書き込む

2) 名前と引数プロトタイプが決まっていて実装をアプリケーションが用意する関数

```
secTmp=$1 # セカンダリごとの一時ファイルパス
L=$(cd $(dirname $0); pwd)  # このスクリプトの配置ディレクトリ
. ${L}/def.sh  # TOS=${L}/99_pool/toSecondaries

tmpfile=${secTmp}.tmp
trap "/bin/rm -f $tmpfile; exit 0" 0 1 2 15

if test "$2 " != "FLOCK "; then
  flock -x ${TOS}.L $0 ${secTmp} FLOCK # ロックをかけて自スクリプト実行
elif test -s ${TOS}; then
  # 先頭行を一時ファイルに書き込む。awk でコマンド部分を取り出す
  set `head -1 ${TOS} | tee ${secTmp} | awk '{gsub("[(]", ""); print $1}'`
  case "$1" in
    :D-*) # dispatch タイプコマンド
       sed 1d ${TOS} > ${tmpfile} # 先頭行を削除
       cp ${tmpfile} ${TOS}
    ;;
  esac
else
  echo ":nop nil" > ${secTmp} # 依頼なし セカンダリには :nop が届く
fi
```

図 2　shell script 例（toSecondaries のピックアップ）

すべてのセカンダリはメッセージを eager にとりにいく．タスクを完了したセカンダリはキューにタスクがある限り，直ちに次のタスクに取り掛かる．タスクの割り当てやスケジューリングといったセカンダリ間の調整のメカニズムは存在しないが，効率的に仕事が進む．

`toSecondaries` から先頭メッセージを受け取る shell script を図 2 に示した．タイプの判定のため，S 式を文字列として分解してコマンド名を切り出している．shell script は Unix (RedHat 系 Linux) に標準的に備わっているコマンドのみで書かれている[3)]．

3.6　広報コマンド

広報はプライマリが同一内容をセカンダリ，あるいはセカンダリのグループに発信する．セカンダリが受け取ったかどうかはプライマリにはわからない．広報にはセカンダリのグループを指定できるチャネル名がある．プライマリとセカンダリはそれぞれチャネル名を指定して広報の発信および，受領を行う．予約のチャネル名として `common` がある．全てのセカンダリを対象とする場合に使用する．プライマリが同じチャネルに発信すると，古いものは上書きされる．

実装では広報は共有ファイルの読み書きで，チャネルはファイル名になっている．

3) Celis は Mac M2 の arm64 ネイティブバイナリでも動作しているが，flock コマンドが標準で備わっていないなど，まったくそのままでは **para-base** は動かない

4　API

プライマリ用 API には，コマンドに対応した動作起動用 API に加えて，dispatch タイプのタスクの追加，dispatch タイプ動作の結果を受領する API と広報の発信用 API がある．

セカンダリ用 API には，要望メッセージ送信と広報受領の API がある．タスクの実行結果は関数戻り値で返すので API はない．

API は関数呼び出しで，API 関数の引数部分はそのまま相手に送られる．引数部分については **para-base** は内容にはまったく関知しない．このため Bridget 以外にも応用可能である．

これらを表 4 にまとめた．`peek-primary` は 8.1.5 節で述べる．

5　アプリケーション

5.1　計算実行の流れ

Bridget でのプライマリ AP の概略の動作を API 呼び出しを中心として図 3 に示す．`(!x y)` は `x` への代入を意味する．`quit` と `stop` は 8.1.5 節で述べる．

セカンダリでは，プライマリからのメッセージの待ちとコマンド解釈をセカンダリ PB が実行する．非同期の評価実行時の動作を図 4 に示す．ここで，`(exit x)` は `x` を戻り値として関数を終了することを意味する．

表 4　API

対象	API	説明
プライマリ sync	コマンドに対応	primary-move など
プライマリ dispatch	コマンドに対応	primary-next-moves など
	primary-add-next-moves	候補手評価依頼追加
プライマリ	primary-pickup-eval-result	結果を受領
プライマリ BB	bb-send-channel	チャネルに発信
	bb-send-common	全セカンダリに発信
セカンダリ	request-to-primary	要望送信
	peek-primary	最新依頼を確認（8.1.5 節）
セカンダリ BB	bb-glance-at-channel	チャネルから受領

6　複数ホストへの拡張

ここまで述べてきた 1 台ホストでの並列方式を複数のホストに拡張する方式を述べる．プライマリを 1 台とするのは変更しない．プライマリが走行しているホストと同じホストで走行しているセカンダリを同居セカンダリ，プライマリと別のホスト（遠隔ホスト）で走行しているセカンダリを遠隔セカンダリと呼ぶ．

```
(initial-move) ; 盤面初期化
(loop outer
  候補手選定
  (next-moves 候補手 ...) ; 候補評価依頼
  (loop inner
    (!返事 (pickup-eval-result))
    要望あり 要望に答え continue inner
    結果あり
      結果の吟味
        必勝手が得られたら (quit) exit inner
        枝刈り情報の広報
        必要なら (add-next-moves) で候補追加
      結果のカウント
        残なしなら着手選定 exit inner
    結果なし ; nil が返ってきた
      タイムアウト発生
        (!結果 (stop))
        結果の吟味 着手選定 exit inner
    )
   (do-move) ; 選定した着手を通知
   勝負決定 or 駒なしなら exit outer
   )
```

図 3　プライマリ AP 動作の概略

```
引数で候補手受け取り
(loop
  広報確認; 枝刈り情報など
  (peek-primary) ; 打ち切り指令確認
    quit 指令あり (exit nil)
    stop 指令あり (exit 現在の評価値)
  評価実施
  必要なら (request-to-primary) で要望発行
  評価終了 (exit 評価値)
  )
```

図 4　セカンダリ AP 評価動作の概略

6.1　方式

1 台環境ではキューとして toSecondaries と toPrimary の 2 つのファイルをプライマリと全てのセカンダリで共有していた．シンプルに拡張するため，toSecondaries ファイルを全てのホストに置く．toPrimary ファイルはプライマリが走行しているホストにのみ置く．

まずセカンダリからプライマリにメッセージを送信する場合を述べる．toPrimary に追加するのはセカンダリだけだが，プライマリがそのファイルを先頭から削除しながら読み込んでいく．プライマリは 1 台なので，追加を以下の手順とする．(1) 遠隔セカンダリ PB はメッセージを書き込んだ一時ファイルを Linux の scp コマンドでプライマリホストの同じパスに送る．(2) プライマリホストにあるデータ追加用の shell script を ssh で実行する，(3) 同居セカンダリ PB は 1 台環境と同じ動作でよい．

shell script は 1 台環境で使用していたものと同一でよい．プライマリは toPrimary のメッセージが

どのホストから届いたかにはまったく影響されない．

次にプライマリからセカンダリにメッセージを送信する場合を述べる．`toSecondaries` は遠隔ホストごとにある．sync コマンドの場合は内容を自ホストと遠隔ホストのすべての `toSecondaries` に書き込めばよい．遠隔ホストの場合は上記と同様に追加用の一時ファイルに書かれた内容を Linux の `scp` コマンドですべて遠隔ホストに送り，ssh で追加用 shell script をリモート実行すればよい．dispatch コマンドは分割して自ホストと遠隔ホストの `toSecondaries` に scp と ssh で分配する．遠隔ホストに分散した `toSecondaries` の同期は不要である．

以上の拡張は，アプリケーションレベルでの変更はまったく不要である． **para-base** レベルでも shell script 実行部分を scp と ssh shell script に変更すること，dispatch コマンドの場合のみ，ホストを意識して分割する部分だけの修正となる．

この方式で新たに必要となった情報はプライマリには遠隔ホストのホスト名，遠隔セカンダリにとってはプライマリが走行しているホストの名前だけである．ホストの CPU 性能やホストごとのセカンダリの台数は性能向上のため（8.1.1 節）に使用するが，複数ホストへの拡張に必須の情報ではない．

プライマリのホスト名を通知するため，sync タイプコマンド`:S-ping` を導入した．複数ホスト全体で何台のセカンダリを使用するかは事前に決め，その台数のセカンダリを立ち上げたあと，`:S-ping` をプライマリホスト名をパラメータとして送信する．この時点でセカンダリがホストに何台あるかは不明だが，sync タイプなのでメッセージは各ホストに 1 つずつ送ればよい．`:S-ping` を受け取ったセカンダリはプライマリホスト名を記録し，返事として自ホスト名を返す．プライマリは返事からセカンダリが走行しているホストごとのセカンダリ台数を得ることができる．

明らかなように，セカンダリ AP が実行するタスクの個数はそのホストに分配されたタスクの個数となるので，CPU が空いてしまう状況が起りうる．これを緩和する方式について 8 章で述べる．

7 並列分散システムとしての性質

複数の計算機での並列実行が実装できたが，並列分散システムとしての性質をまとめる．

1. 耐故障性 (fault tolerance)
   (a) 現状では考慮していない．プライマリだけでなく，セカンダリが 1 プロセスでも止まると全体が止まる．
   (b) 死活監視，冗長化，退行運転のどれも実装していない
2. 透過性 (transparency)
   セカンダリの総数やホスト台数，同居しているセカンダリの台数はアプリケーションの記述や実行に影響しない．複数ホストでの実行時の効率は **para-base** レベルで工夫する
3. 可伸性 (scalability)
   4 ホスト，72 セカンダリでは特に shell script の起動やファイルの読み書きといったものにボトルネックは現れていない．
4. 安定性 (stability)
   4 ホスト，72 セカンダリでの数日間の連続実行では徐々に遅くなるといったことはない．通信の実装が shell script とファイルなのでメモリリークといったものもない．

8 性能向上のための方式・機能

8.1 para-base レベル

8.1.1 ホスト単位のタスクスケジューリング

コア数や CPU 性能の異なる複数のホストにセカンダリが存在している場合，タスクの割り当てを調整することで全体の実行時間を削減できる可能性がある．

これまでセカンダリはその総数だけを管理していたが，タスク割り当てのためホストごとのセカンダリ数，実行中タスク数，待ちのタスク数の管理を導入する．またホスト性能を事前に定義しておく[4)]

このため，セカンダリからのタスクの返事にホスト名を同梱する．メッセージにホスト名を追加するのは **para-base** レベルで実行するので，アプリケーションには変更はない．

4) 現在 `1.0` – `2.5` 程度の実数でその相対的な性能を表すこととしている．値の大きいほうが高性能．厳密な値ではない

現状のスケジューリングは，(1) 速いホストからコアの空きを埋めていく．(2) すべてのホストのコアの空きがなくなったらホストごとに
(/ (* 性能 セカンダリ数) (1+ 待ちタスク数))
を求めこの比率でタスクを割り当てる，としている．

8.1.2 Ack 集約

プライマリが発行する sync タイプコマンドの Ack は返ってきた個数だけが必要なので，セカンダリのホストの **para-base** で Ack 送信を延期し，個数が揃ったところで Ack 個数を返すようにした．小さなデータを送る scp や ssh による遠隔 shell script 起動が削減できた．

セカンダリは自分が実行しているホストに何台のセカンダリがいるかを知らないので，プライマリからの sync タイプコマンドにそのホストのセカンダリ数を同梱した．セカンダリ PB は Ack をすぐに送らずにホストごとのファイルに貯めていく．個数が揃ったセカンダリ PB がプライマリに送信する．プライマリと同居しているセカンダリについてはファイル書き込みだけなので個別に Ack を返している．

8.1.3 タスクリスケジューリング

ホストに割り当てたタスクはそのホストのセカンダリが実行するが，運が悪いと実行待ちのタスクをもっているホストと空きセカンダリがあるホストが同時に出現することがある．これを解消するため，プライマリは実行待ちのタスクをキューの先頭から引き戻して空きのあるホストに再割当てする方式を実装した．実行が始まっていタスクを別ホストに移動するタスクマイグレーションは TAO では困難だがリスケジューリングは shell script レベルでも容易に実現できる．再割当てされたタスクは直ちに実行が開始される．もともとタスクの実行開始の順番は保証していないので，問題はない．

8.1.4 ファイル書き込みの検知

メッセージはファイル経由でメッセージの到着は polling で検知していたが，polling の時間間隔は全体の実行時間に影響を与える．この到着検知の遅れを解消するため，ファイルへの書き込みを TAO に通知する機構を実装した．これは **para-base** レベルではなく，Celis 本体の修正となった．具体的には Linux の `inotify(7) monitoring file system events` 機構を利用して TAO の入力 tty と監視対象ファイルパスを結びつけ，ファイルへの書き込みクローズで入力待ちから戻る仕様とした．

8.1.5 打ち切り動作

計算を並列で実行して結果をすべて受け取るのは基本であるが，Bridget のような対戦ゲームでは例えば必勝手が得られたら他の実行中の結果や依頼済みの評価はすべて不要になる．また対人間でゲームを進める場合，計算機には 1 手 30 秒のような時間制限を設けたい場合もある．これをアプリケーションで実現するには例えば図 5 のようにする．`quit` を打ち切り，`stop` を timeout とする．

```
プライマリ
 quit / stop を bb-send-common で広報
 返事を確認（正常終了と同じ）
セカンダリ BB で受領
 quit 実行中は打ち切り、新規依頼は直ちに return
 stop 実行中は区切りまで実行、新規は直ちに return
```

図 5　アプリケーションでの打ち切り動作

今回は，打ち切りを **para-base** で支援するため，プライマリに sync タイプのコマンド `stop` `quit` と対応する API，セカンダリにプライマリからの最新指令（`toSecondaries` の先頭行）を見る `peek-primary` を追加した．これらを使ったときに打ち切り動作は図 6 のようになる． **para-base** が支援することで，実行前の依頼をすべて取り消すことができる．また，プライマリは sync タイプの API で停止できるので打ち切りの完了が **para-base** で保証される．また途中結果もまとめて得られる．例外処理の `throw–catch` の形といえる．実装は既存の sync タイプ動作と小さな機能の API 追加だけであった． **para-base** の基本動作は十分な拡張性を備えているといえる．

```
プライマリ
 primary-stop/primary-quit 発行（sync タイプ）
 PB: 実行前の依頼を取消（タスク管理簿調整）
 ack を待つ
   stop  区切りまでの評価値がリストで得られる
   quit  戻り値不要
 PB: タスク管理簿をクリア
セカンダリ peek-primary で最新指令受領
 quit 実行中は打ち切り，今後新規はこない
 stop 実行中は区切りまで実行，今後新規はこない
```

図 6　para-base 支援の打ち切り動作

8.2 チューニング（設定 Tips）

複数ホストでの並列動作は ssh とファイル読み書きを多用するので，実際に稼働させる場合に若干の配慮をするとよい．今回は次のような設定で実行した．

1. メッセージ用ファイルの置場をメモリファイルシステムにおく
   例えばディレクトリ`/run/user/`*uid*`/`はログインユーザごとに tmpfs 上に作成される．実体は主記憶であって読み書きが高速である．SSD を傷めることもない．
2. ssh 設定で永続コネクションを利用
   `ssh_config` の `ControlMaster` を設定することで，確立した ssh 接続を使いまわすことができる．TCP の three-way ハンドシェイクや ssh の認証をすべて省略できる．
3. 最速のホストにプライマリを置く
   ホストの CPU コア数だけセカンダリを用意するとプライマリ専用のコアはない．最速のホストに置けば影響が少ないことが実測された．

9 para-base を用いた並列 $\alpha\beta$ と Bridget の概要

ゲーム木探索では $\alpha\beta$ 法を使うが，これをいかに並列化するかについては過去にたくさんの研究がある（横山 [1] に分かりやすい解説がある）．これらの研究では，最善の子の系列 (Principal Variation, PV) が（反復深化などの技術により）探索木の左のほうにあることが前提となっている．並列でなくても，このほうが探索カットが多くなる．

しかし，Bridget の場合，我々の局面評価関数が未熟であり，PV が探索木の左のほうにあることがあまり期待できない．しかも，ゲーム木は最大深さ 28 と浅いのに，分岐数が中盤くらいまで 500 を越える（竹内 [3]）．この中で PV を見つけるのは至難の業である．

このような事情から，与えられた局面での候補手（第 1 レベル候補手と呼ぶ）を高々 20〜30 手に絞ってから，先読みを行う．なお，第 2 レベル以下の候補手は高々 20 程度に絞る．評価関数が未熟なため，これでも一発での必勝手を見逃すことがある．

このように PV の効果があまり期待できない状況なので，採用した並列化 $\alpha\beta$ 法（実際にはいわゆる negative $\alpha\beta$ 法，ここでは探索カットは α カットのみ）は，1 手先読みの非常に浅い先読み評価だけでソートした，ほぼ均質に近い並列化を行う．具体的には，第 1 レベル候補手それぞれについて，第 2 レベル候補手の 2 個または 3 個のリスト（バルクと呼ぶ）のリストを作り，それぞれの第 1 レベル候補手の範囲でバルク単位の並列探索を行う．

プライマリは，多数のセカンダリにゲーム木の部分木の探索をタスクとして与えるが，同じ第 1 レベル候補手に関して探索を行っている複数のセカンダリは，第 1 候補手を表す文字列（例えば，`(87 . DW)` なら`"87DW"`）をチャネル名とする BB を共有する並列探索グループに属していることになる，その BB チャネルを通して，ローカルな α 値の更新を行い，必要な α カットを行う．ただし，BB チャネルを共有しているとはいえ，ローカルな α 値の更新は必ずプライマリが行う（こうしないと正しい制御ができない）．

これらの工夫により，72 コアを用いて妥当な時間内に 4 手先読みが可能になった．全 174 種類の初手に対しての自己対戦は，1 試合平均 12 分程度で行えているが，この自己対戦を通じてプログラムの改良を行っている．

なお，$\alpha\beta$ 法のプログラムは，プライマリ用とセカンダリ用で異なる．セカンダリの中では，**para-base** AP は呼ぶものの，木探索が逐次処理となるからである．

さて，この $\alpha\beta$ 法のプログラムを Bridget 以外のゲームにも使えるようにすべきである．実際，プログラムの中で，ゲームに依存しているのは，局面評価関数，チャネル名を作るための候補手の名前，候補手選択，候補手の数の絞り込みに使う終盤や序盤などの判定のみなので，これらをゲーム依存のプログラムモジュール（いわば，`Bridget-pAI` モジュール，従来の `Bridget-AI` の並列拡張）へのコールバックにすることによって実現する．

さらに，任意の候補手レベルで並列探索セカンダリグループを形成して，並列探索を可能にすれば，さらに汎用性が高まる（現在実装中）．

なお，この **para-base** を用いて，実際に（$\alpha\beta$ 法を使わない）n-queen プログラムを，簡単な拡張で並列 n-queen プログラムにして走行させることに成

功している．

10　性能測定

徹底的な性能測定とは言えないが，ここでは，bug63-と呼んでいる特定の局面での並列化の効果を示す（40 コアで十分な問題）．図 7 に示した bug63-は黒の手番だが，見てすぐ分かるように，黒はいかにもバラバラでブリッジできそうにない形をしている．しかし，ここでは（87 ． DW）が必勝手であり，にわかには信じがたいかもしれないが，次の手番で白が 4 手先読みで敗戦を認める．

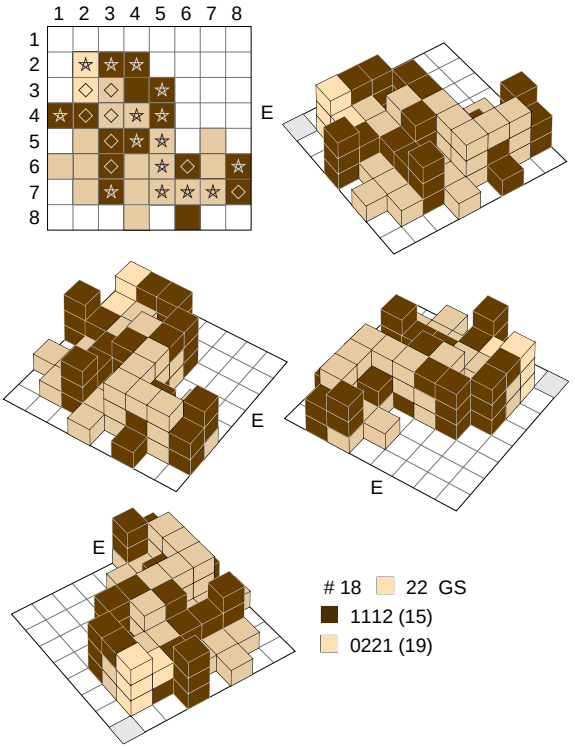

図 7　bug63-の局面

第 20 手目で白が敗戦を認めるのは 2 段階を要する．まず，通常の第 1 レベルの候補手 23 個がすべて負けにつながることを知り，次に Emergency mode に入り，パスをしたら，黒が何を打つかを調べる．その手を妨害する候補手で 4 手先読みを行う．これでも負けということで PASS を選ぶ．図 9〜11 ではこの 2 段階が見てとれる．最上段のプライマリの実行状況の青緑色の部分が Emergency mode 中であることを示している．

表 5　CPU 時間測定 先読み深さ 4

台数	経過時間	CPU タイム			
		primary	secondary		
			合計	平均	率
1	1282.00	1.96	1279.40	1279.40	99
8	410.34	2.01	3172.02	396.50	97
16	197.66	2.48	2726.48	170.41	86
24	156.04	2.84	2897.91	120.75	77
40	130.96	5.75	2980.18	74.50	57
56	123.98	6.62	3246.72	57.98	47
72	123.26	7.93	3508.53	48.73	40

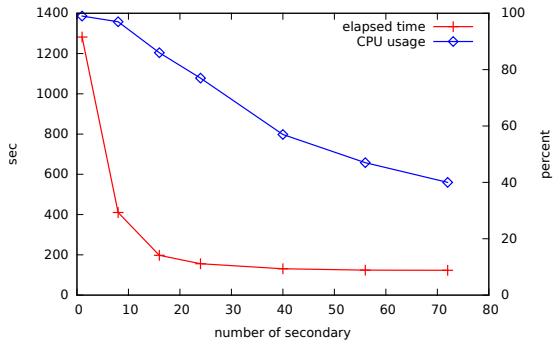

図 8　経過時間と CPU 使用率

台数効果を示すものとして，コアの走行状況を示すダイアグラムを図 9〜11 に示す．それぞれ，同一ホストのコア 8 個，複数ホストのコア 40 個，コア 72 個の走行状況である．詳細は省略するが，横軸は経過時間（秒），2 段目からがセカンダリの状況で，橙色がタスクの実行，青色の部分が 1 つのタスクの終わり部分，縦の長い黒細線が次の着手を示している．コア 8 個の場合は，コアの使用率がほぼ 100%であり，無作為なスケジューリングがほぼ完璧に機能していることが分かる．タスクを eager に取りにいく構造なので，連続したタスクの実行間にすきまはない．コア 40 個の場合は，ところどころにつっかい棒のように律速をしているコアがある．そのため，全体としてのコア使用率が 57%に下がっている．この傾向は 72 コアでさらに顕著になる．

コア使用率を上げて，台数効果を上げるためには第 9 章で言及した，現在実装中の「任意の候補手レベルでの並列探索セカンダリグループの形成」が必要であろう．つまり，つっかい棒になっている探索をダイナミックに並列分解することである．

セカンダリ Celis プロセスの台数を変化させたときの 4 手先読みの測定結果を表 5 と図 8 に示す．単位は秒である．18 手まで打った状況から勝負がつくまでの経過時間を計測した．20 手までで勝負がつく．表 5 には CPU 消費時間を示した．user と sys の合計で，子プロセス（shell script 実行）の CPU 時間も含んでいる．率は経過時間に占めるセカンダリの平均 CPU タイムの割合である．

セカンダリ台数 1 から 24 は 1 台のホストで，それ以上は複数台のホストでの分散並列である．ホストの構成と諸元を表 6，表 7 に示す．プライマリはもっとも性能のよい n139 で実行した．

いずれの場合も台数の増加にともなって実行時間が減っている．複数台にまたがった場合も効果が認められる． **para-base** はスタートポロジを採用したので，一般には中央のプライマリが性能上のボトルネックになることがあるが，Bridget のプライマリはセカンダリ台数が増えてもネックになるような負荷は見られない．なお，この実行中にはタスクのリスケジューリングは発生しなかった．

表 6　ホスト構成

台数	ホスト構成
1〜24	n139
40	n139 nryz
56	n139 nryz n1292
72	n139 nryz n1292 n129

表 7　測定ホスト

ホスト	CPU	コア数	OS
n139	Core i9-13900KF	P8 E16†	RHEL 9.0
nryz	Ryzen 9 5950X	16	RHEL 8.5
n1292	Core i9-12900	P8 E8	RHEL 9.0
n129	Core i9-12900	P8 E8	RHEL 9.0

† P performace コア，E efficient コア

11　むすび

通信機能のないシステムに外付けで構築した並列計算フレームワークについて述べた．ファイル経由の通信と shell script による実装で，ホスト 4 台 72 セカンダリの並列計算が実現できた．

参考文献

[1] 横山大作: 「激指」におけるゲーム木探索並列化手法．人工知能学会誌, Vol.26, No.6 (Nov. 2011) pp.648–654

[2] 天海良治: マイクロプログラムの静的変換による記号処理システムの移植とその性能評価. 電子情報通信学会論文誌, Vol.J84-D-1, No.1 (Jan. 2001) pp.100–107
https://www.nue.org/nue/tao/celis/celis.pdf

[3] 竹内郁雄，天海良治: 立体連結ゲーム Bridget のプログラミング（改）– 古典的手法再び. 第 63 回プログラミング・シンポジウム報告集, 2022 年 1 月（実際は 2022 年の山内記念賞受賞講演のために，改訂したもの）
https://www.nue.org/nue/nue-essays/nue-essays/Bridget-programming.pdf

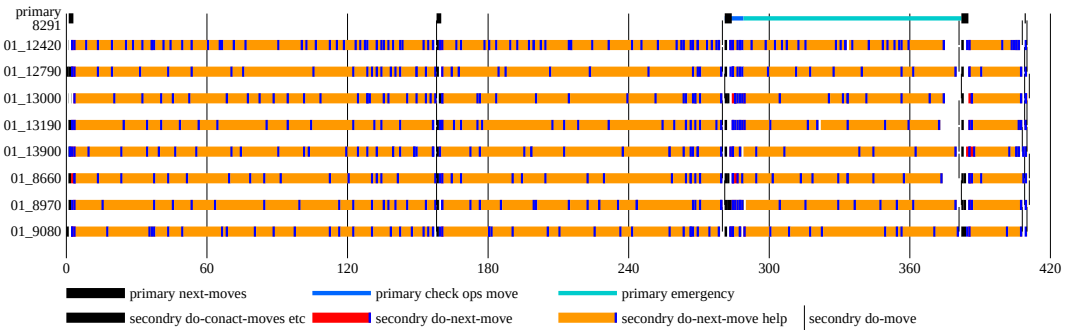

図 9　セカンダリ 8 台 ホスト 1 台

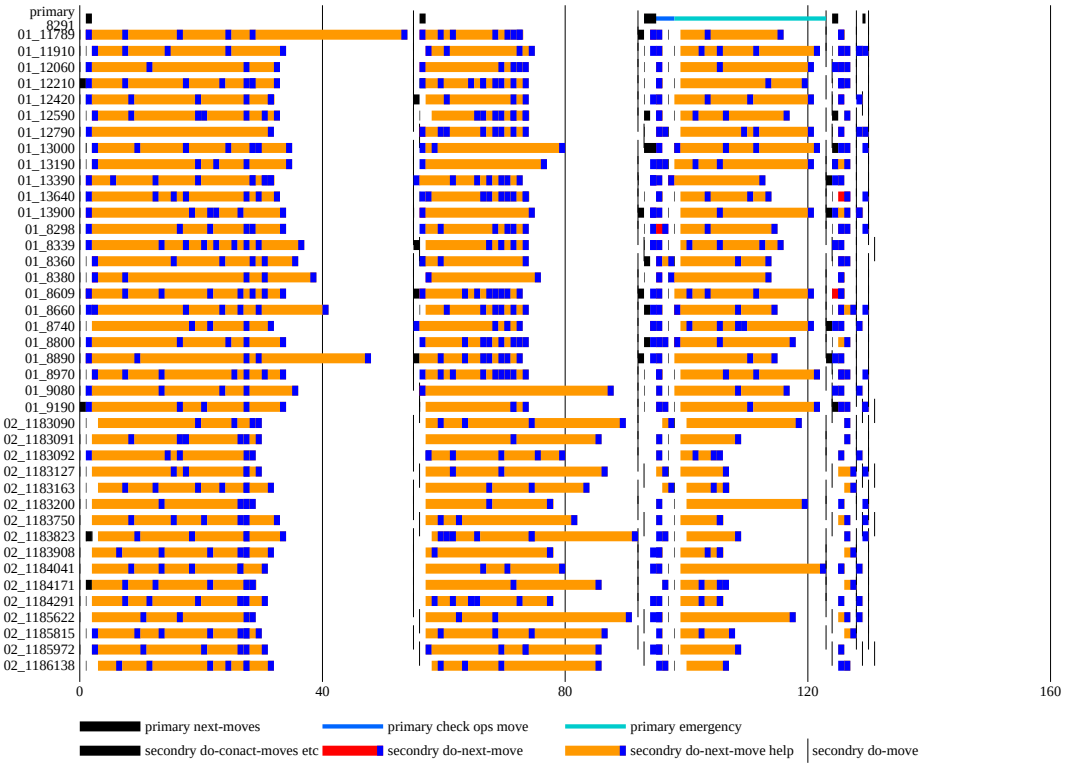

図 10　セカンダリ 40 台 ホスト 2 台

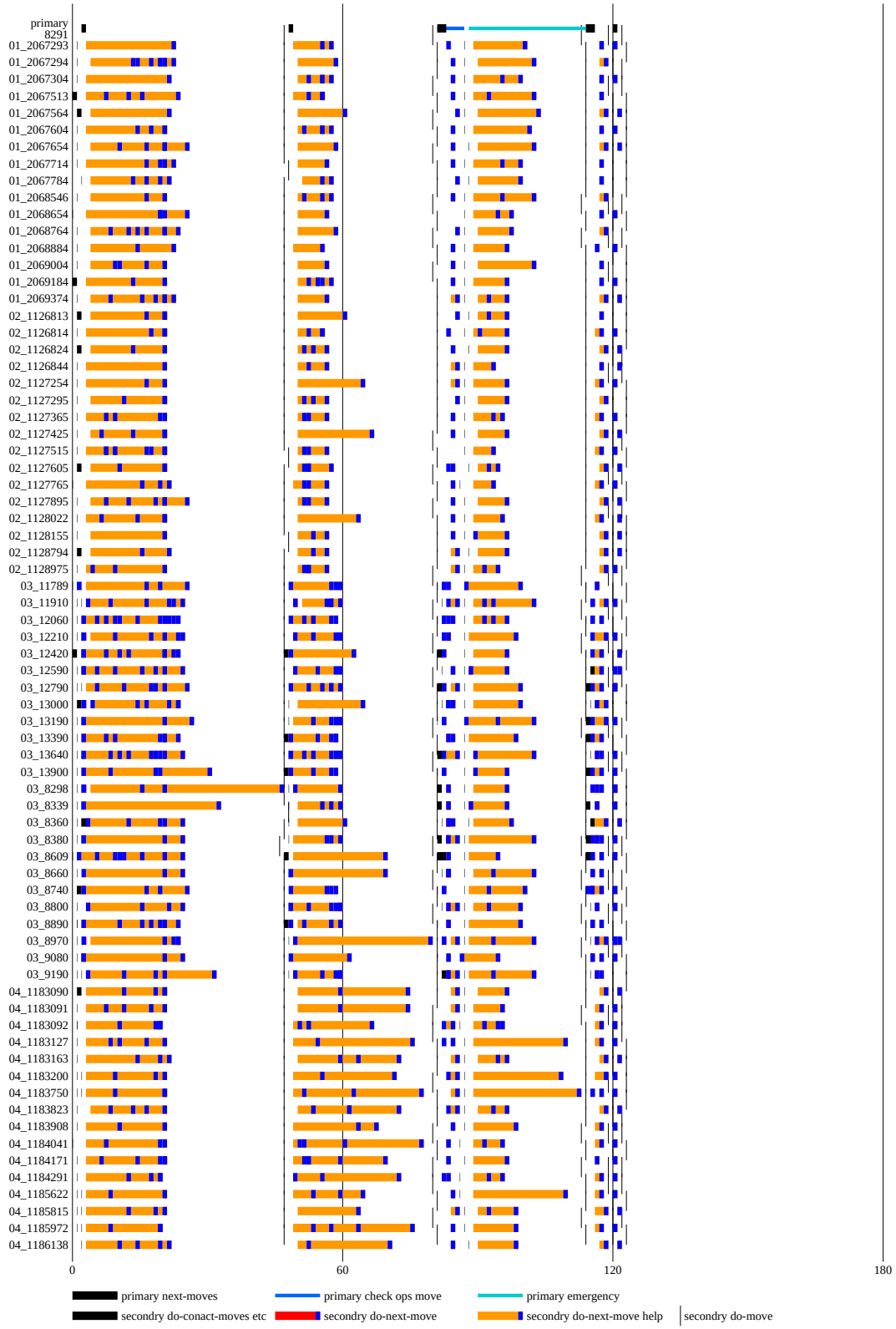

図 11　セカンダリ 72 台 ホスト 4 台