並列プログラムのN逐次実行方式とMGTPへの適用

長谷川，隆三
九州大学大学院システム情報工学研究科知能システム学専攻

藤田，博
九州大学大学院システム情報科学研究科知能システム学専攻

中田，健浩
九州大学大学院システム情報科学研究科知能システム専攻 ： 修士課程

力，規晃
徳山工業高等専門学校

https://doi.org/10.15017/1523859

出版情報：九州大学大学院システム情報科学紀要. 2 (2), pp.241-246, 1997-09-26. 九州大学大学院システム情報科学研究科
バージョン：
権利関係：

九州大学大学院
システム情報科学研究科報告
第2巻 第2号 平成9年9月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.2, No.2, September 1997

並列プログラムのN逐次実行方式とMGTPへの適用

長谷川 隆三* ・ 藤 田 博* ・ 中田 健浩** ・ 力 規 晃***

An N-Sequential Execution Method for Parallel Programs and Its Application to MGTP

Ryuzo HASEGAWA, Hiroshi FUJITA, Takehiro NAKATA and Noriaki CHIKARA

(Received June 23, 1997)

Abstract: An efficient parallel execution method is presented for parallel programs which create many tasks at runtime on parallel machines with N number of processing elements (PEs) available at any time. In this method, the number of tasks activated at the same time is limited to at most N, and the exceeding number of created tasks are suspended until they are allowed to be activated, thus being processed sequentially. A PE can request a task to another PE only when the set of executable tasks of the requesting PE is empty. A PE can transfer a task to another PE only when the latter requests it and the set of executable tasks of the requested PE is not empty. This effectively suppresses the communication overhead induced by the task transfers between PEs. Also, it allows tasks which should be better executed sequentially to stay on a single PE, while keeping the execution rate of tasks at every PE very high. We implemented the method for the parallel theorem proving system MGTP written in the concurrent programming language KL1 running on the parallel inference machine PIM/m (128PE) as well as the shared-memory parallel machine Cray SuperServer6400 (20PE). The results show remarkable reduction of communication overhead achieving significant speedup.

Keywords: Parallel processing, Load distribution, Concurrent logic programming, Theorem proving

1. はじめに

プログラム(並行プログラムとは限らない)の並列処理，あるいは分散処理における最大の課題は，効率(台数効果)を最大化すべく，如何に適正な負荷分散を行うかということである．特に，細粒度負荷分散を行う場合は，処理要素(PE)間の通信オーバヘッドを最小限に抑えることが大切である．

本論文では，複数のタスクが動的に生成されるようなプログラムに対して，効率のよい並列処理方式を提案する．この方式は，並列実行されるタスクの起動数を高々利用可能なPEの数Nに制限し，Nを超えて生成されたタスクについては逐次化して実行する．負荷分散に関しては，自PE内で実行可能なタスク集合が空であるときに限り他PEにタスク分与要求を行う．未処理タスクをもつPEは，他のPEからのタスク分与要求を受けたときのみタスク分与を行う．

従来，循環割付けや確率的割付けなどのような，タスクが生成されるとただちにPEへ分配する方式が用いられていた．これらに比べ，本方式ではPE間のタスク移動に伴うオーバヘッドが最小限に抑えられる．また，逐次実行することが望ましいタスク群に対してPE間の移動を避けつつ，各PEの稼働率を高く保つことができる．

具体的には，動的に生成されるタスクの親子関係が木構造を形成するような木型プログラムのクラスを対象にする．与えられた木型プログラムを並列化するため，そのタスク生成部にわずかな修正を施すとともに，負荷分散化機構，すなわちPEごとにタスクを管理するプロセスとPE間通信を制御するマスタープロセスを加える．

本方式を並列定理証明系MGTP[1]に適用した．MGTPは非ホーン節を扱い，場合分けによりOR並列推論を行なう典型的な木型プログラムである．タスク管理プロセスやPE間通信のためのマスタープロセスを並列論理型言語KL1[2]により記述した．これを並列推論計算機PIM/m(分散メモリ型)，ならびに共有メモリ型汎用並列計算機の上で動作させ，いずれの場合も，通信量比率が低くかつ稼働率が高い並列分散化が達成されることを確認した．

2. 木型プログラム

本論文で扱うプログラムのクラスを定めるため，以下のような定義を行う．

タスクを生成しないタスクを終端タスクという．非終端(親)タスクTは，(子)タスク$< T_1, T_2, \ldots >$を順に生成する．すでにタスク$< T_1, \ldots, T_{i-1} >$までを生成し，

平成9年6月23日受付
* 知能システム学専攻
** 知能システム学専攻修士課程
*** 徳山工業高等専門学校

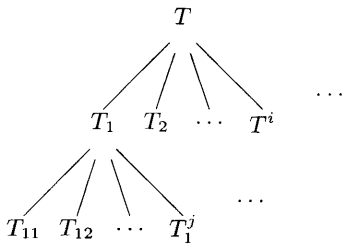


Fig.1　Task tree

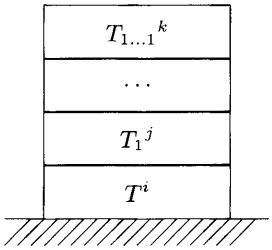


Fig.2　Task stack

$< T_i, \ldots >$の生成を待つ非終端タスクをT^iで表す．非終端(根)タスクTは，その子タスクT_iがまた非終端タスクのとき，子孫タスクが順次生成されることにより**Fig.1**に示すような木構造を形成する．これをTのタスク木という．このような非終端タスクを生成するプログラムを木型プログラムという．

本論文では，各非終端タスクが生成する子タスクの数が有限(有限のiでT^iが終端タスクとなる) であるような木型プログラム†1のみを考える．

2.1　木型プログラムの逐次実行

木型プログラムを逐次実行する場合には，**Fig.2**のようなスタックを用いる．

非終端タスクTから子タスクT_1が生成されたとき，T^2をスタックに積んでおき，T_1の実行を進める．T_1が非終端タスクの場合，その子タスクT_{11}が生成され，$T_1{}^2$がスタックに積まれる．こうして，タスク実行がタスク木の高さ方向に進められるに従ってスタックの段数が重ねられる．いずれ，T^2がスタックから取り出されて実行されたとき，T^2が非終端タスクなら子タスクT_2が生成され，T^3をスタックに積んでT_2の実行が進められる．

3.　木型プログラムの並列化

非終端タスクTの子タスク$< T_1, \ldots, T_n >$が並列実行可能の場合，各々を別々のPEに分配して起動してよい．

†1　本方式の適用に関する限り，タスク木の高さは必ずしも有限(すべての枝が終端タスクの葉に至る) である必要はない．

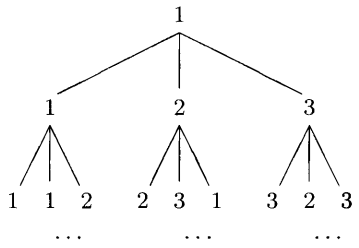


Fig.3　Cyclic allocation method (N=3)

Fig.3に，タスク木において各タスクにPE(N=3)を循環割付けする方式の例を示す．

PE1の上で根タスクTが子タスクT_1とT^2を生成すると，ただちにT^2がPE2に渡される．PE2上でT^2が子タスクT_2, T^3を生成すると，ただちにT^3がPE3に渡される．以下，同様にして，タスク$T_{12}, T_{13}, T_{22}, T_{23}, T_{32}, T_{33}$ にはそれぞれPEが1,2,3,1,2,3という順に循環的に割付けられる†2．ここで，最初の子(長兄)タスクは親タスクが実行されていたPEで引続き実行されることに注意．タスク木の1本の枝上のタスク系列はなるべく同一のPEで連続して処理した方がよい．枝の途中でPEが変わると枝に沿って伝達される情報の転送が必要となり，通信オーバヘッドを増大させるからである．

一般に各非終端タスクの生成する子タスクの数は一定でないので，子タスクの移動先PE番号を各PEにおいて局所的に決定することはできない．単一の大域的カウンタを用いた割当ての機構が必要である．

3.1　N 逐次方式

循環割付け方式では，2番目以降の子タスクについては一般に(循環による一致を除いて) 常にほかのPEへの移動が発生する．従って，タスク移動総数はタスク木のノード数に比例する．これは，PE間通信コストが高い(時間がかかる)場合には逆の台数効果(実行速度低下)をもたらす．

同時利用可能なPEの数がNに限られるとき，実際に並列走行可能なタスクは高々N個なのだから，大域的にNを超える数のタスクが生成されている場合には，タスク移動を行うのは無駄である．循環割付けは，「タスクがPEを取りに行く」スタイルに基づいている．PE数が有限である限り「(暇な)PEがタスクを取りに行く」スタイルに基づくべきであるが，その実装はシステムに強く依存し，前者に比べて困難である．

我々は，後者のスタイルをシステム独立にプログラミング技法の範囲内で近似的に実現するために，N逐次方式を考案した．**Fig.4**に，N逐次方式によるPE(N=3)の割付け例を示す．

†2　T_i^4等は終端タスクであり，表示されていない．

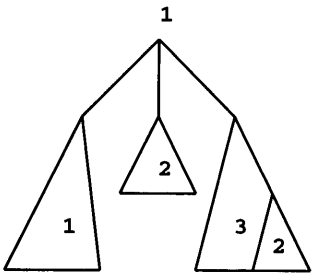


Fig.4 PE allocation with N-sequential method (N=3)

PE1の上で非終端タスクTが子タスクT_1とT^2を生成し，T_1の子孫タスクはPE1で逐次実行される．その間，PE2からのタスク分与要求があり，T^2がこれに渡される．PE2上では子タスクT_2が生成され，その子孫タスクが逐次実行される．その間，PE3からのタスク分与要求があり，T^3がこれに渡される．PE2がT_2の子孫タスクの逐次実行を完了したとき，PE3(PE1でもよい)にタスク分与要求を行うことができる．

各PEは2.1節に示したスタックを用いてタスクの逐次実行を行う．ある終端タスクを実行終了すると，スタックが空でない場合にはスタック頂上から未処理タスクを取り出して実行を再開する．ほかのPEからタスク分与の要求があり，かつ自PE内のスタックが空でない場合は，要求元のPEへのタスク移動が可能である．このとき，分与されるタスクはスタックの底から取り出すことにする．すなわち，タスク木の(葉ではなく)根に近い部分で部分木を分割する．これにより，大粒度の部分木が分与されることになり，分与を受けたPEが再び分与要求を行うまでの時間が延びて，PE間通信量を抑える効果が得られる．こうして，理想的な場合(釣合のとれたタスク木)，N逐次方式はタスク木をN個の均等なサイズの部分タスク木に分割し，それぞれを逐次実行するという方式になっている．

4. MGTPのOR並列実行

4.1 モデル生成型定理証明系MGTP

MGTPは，$A_1,\cdots,A_n \rightarrow B_1;\cdots;B_m (n,m \geq 0)$ のような含意形式で表される節の集合に対し，モデル(節集合を充足するアトム集合)を，空集合を初期値として構成的に求める手法[1]である．構成途中のアトム集合Mをモデル候補と呼ぶ．節集合が非ホーン節を含むとき，後件$B_1;\ldots;B_m (m>1)$に従って場合分けによるモデル候補拡張が行なわれる．すなわち，モデル候補Mに関するMGTPタスクT_Mは，並列実行可能な子タスク集合$T_{M\cup\{B_1\}},\ldots,T_{M\cup\{B_m\}}$を生成する非終端タスクとなる．一般に，各モデル候補を担当するタスク$T_{M\cup\{B_i\}}$がさらに場合分けモデル候補拡張により非終端タスクとなるので，T_Mは木型タスクである．MGTPプログラムは，1個

```
mgtp(D0, M, Cls, Ans, ReqTS) :-
  empty(D0, E),
  (E = yes → Ans = sat, ReqTS = [pop_top]
  ; otherwise; true → pickup(Delta, D0, D1),
    subsTest(M, Delta, T),
    (T = yes → mgtp(D1, M, Cls, Ans, ReqTS)
    ; T = no, Delta = [_|_] →
      caseSplit(Delta, D1, M, Cls, Ans, ReqTS)
    ; otherwise; true →
      cjm(Cls, Delta, M, F),
      (F = [false|_] → Ans = unsat, ReqTS = [pop_top]
      ; otherwise; true →
        cjm(Cls, Delta, M, New),
        addM(M, Delta, M1),
        addNew(New, D1, D2),
        mgtp(D2, M1, Cls, Ans, ReqTS)))).
```

Fig.5 MGTP main program *mgtp*/5

の初期モデル候補$M_0=\phi$に関するMGTPタスクT_{M_0}を含む木型プログラムである．MGTPプログラムのタスク木は，与えられた節集合が充足不能の場合には，すべての枝がモデル候補棄却を実行する終端タスクに対応する葉で終るような有限の木となる．従って，MGTPプログラムに対して本並列化方式が適用できる．

4.2 N逐次方式によるOR並列実行

MGTPプログラムの本体である*mgtp*/5述語を**Fig.5**に示す．第5引数の*ReqTS*は，タスク管理プロセスへメッセージを送るためのストリームである．

推論過程の詳細に関する部分の説明は省略する．ここでは，MGTP(終端)タスクが終了する局面($Ans=sat$と$Ans=unsat$のとき)のみに着目すればよい．このとき，*ReqTS*上に*pop_top*メッセージを発行する．タスク管理プロセスが*pop_top*メッセージを受けたとき，タスクが存在すれば自PE(*pop_top*メッセージ発行元)でこれを起動する．

Fig.6に示す*caseSplit*/6述語は，MGTP非終端タスクを生成する局面を担う．入力節の後件が2個以上のリテラルからなる選言であるとき，$Delta=[H,H2|T]$であり，先頭の子タスクHは自PEで引続き実行し，残りの兄弟タスク$[H2|T]$をスタックに積むべく*ReqTS*上に*push*メッセージを発行する．

4.3 タスク管理プロセス

タスク管理プロセスは，**Fig.8**に示すように未処理MGTPタスクをスタック上で管理するようなプロセスとして，**Fig. 7** に示す *tsManager*/8述語で実現される．第5引数の *ReqTask* は，タスク管理プロセスへのタスク分与要求メッセージのためのストリーム，第6引数の *GiveTask* は，同じくタスク分与提案メッセージの

```
caseSplit(Delta, D, M, Cls, Ans, ReqTS) :-
  (Delta = [H] → case1(H, D, M, Cls, Ans, ReqTS)
  ; Delta = [H, H2|T] → joinAns(Ans1, Ans2, Ans),
    ReqTS = [push(caseSplit([H2|T],
              D, M, Cls, Ans2, _))|ReqTS1],
    case1(H, D, M, Cls, Ans1, ReqTS1)).
case1(H, D, M, Cls, Ans, ReqTS) :-
  addNew([H], D, D1), mgtp(D1, M, Cls, Ans, ReqTS).
```

Fig.6　Task creation part *caseSplit*/6

ためのストリームである．

タスク分与を提案したメッセージに対し，マスタープロセスから確認メッセージ($info(Ack, \ldots)$)を受信したとき，*pop_bottom*により，スタックTSの底からタスクの取り出しを試みる．TSが空の場合は，$Ack = no$と返答する．タスクが取れた場合，$Ack = yes$と返答するとともに，*give_task*により実際に要求元$ReqPE$へのタスク移動を実行する．*pop_bottom*後のスタックが空でない場合，新たなタスク分与提案(*give_task*)メッセージを$GiveTask$ストリーム上に発行する．

MGTPプログラムから*pop_top*メッセージを受信したとき(コードは省略)，スタックTSが空ならば，$ReqTask$ストリーム上に*get_task*メッセージを発行する．タスクが取れる場合には，スタック頂上から一つタスクを取り出し，自PEでこれを起動する．

MGTPプログラムからタスクの*push*メッセージを受信したとき(コードは省略)，スタックTSにそのタスクを積む．$Flag = ok$の場合，$GiveTask$ストリーム上に*give_task*メッセージを発行しておく．

$Flag$は，*give_task*メッセージの発行のタイミングを適切に制御するためのものである．すなわち，他PEへ移

```
tsManager(no, info(Ack, ReqG, ReqPE, MyReqTS),
    CurPE, ReqTS, ReqTask, GiveTask, CurG, TS) :-
  ReqPE ≠ −1 |
  pop_bottom(Task, TS, NTS),
  (TS = [ ] → Ack = [no_task], Flag = ok,
    ReqTask = ReqTask2, GiveTask = GiveTask2
  ; otherwise; true → Ack = [have_task],
    (NTS = [_|_] →
      GiveTask = [update(CurG, CurPE, NInfo)
          |GiveTask0], Flag = no
    ; otherwise; true →
      GiveTask = GiveTask0, Flag = ok),
    give_task(Task, ReqG, ReqPE, CurPE, MyReqTS,
      ReqTask, ReqTask2, GiveTask0, GiveTask2)
      @node(ReqPE)),
  tsManager(Flag, NInfo, CurPE,
    ReqTS, ReqTask2, GiveTask2, CurG, NTS).
```

Fig.7　Task management process *tsManager*/8

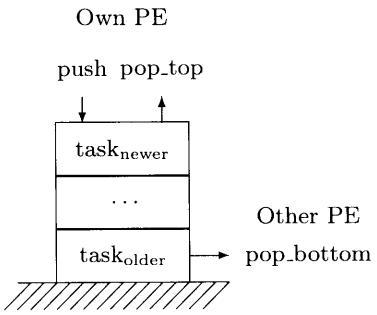


Fig.8　Task management with a task stack

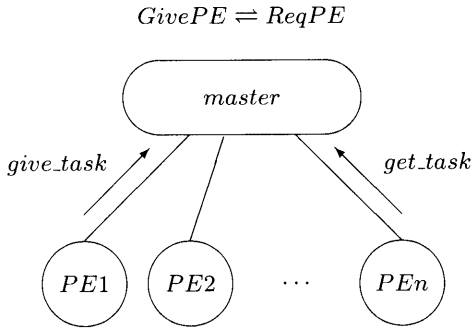


Fig.9　Master-slave architecture

動可能なタスクが存在していても，以前発行された*give_task*メッセージに対する実際のタスク移動の成否が確認されない間は，新たな*give_task*メッセージの発行を抑えるようにしている．

4.4　マスタープロセス

あるPEからのタスク分与要求と，別のPEからの未処理タスクの分与提案を受け，双方の間でのタスク受渡しの仲介を行わせるために，**Fig.9**に示すようなマスタープロセスを用意する．これは，**Fig.10**の*master*/4述語で実現される．第3，4引数の*Vec*, *NewVec*は各PEにおけるタスク処理状況，すなわちタスクが尽きた状態(初期状態を含む)から，新たなタスクを得て稼働状態に遷移した回数Gen(世代番号) を記録しておくベクタである．

*get_task*と*give_task*の両メッセージが届いているとき，$GivePE$から$ReqPE$へのタスク移動の可能性を調べる．$ReqTask$ストリームと$GiveTask$ストリームが別個であることから，一つのPEから発行された*get_task*メッセージが，同じPEからそれ以前に発行された*give_task*メッセージより早くマスタープロセスで受信されるという，“追い越し”が生じる可能性がある．*get_task*メッセージによれば，そのPE上のタスクはすでに尽きているのだから，遅れて受信された*give_task*メッセージのタスク分与

```
master([get_task(ReqTry, ReqG, ReqPE, NReqTS)
        |ReqTask],
    [give_task(GiveG, GivePE, Info)|GiveTask],
    Vec, NewVec) :-
  vector_element(Vec, GivePE, CurGG) |
  (CurGG > GiveG → Info = info(_, _, −1, _),
    master([get_task(ReqTry, ReqG, ReqPE, NReqTS)
      |ReqTask], GiveTask, Vec, NewVec, Stop)
  ; ReqPE = GivePE → Info = info(_, _, −1, _),
    set_vector_element(Vec, 1,
      info(Cnt), info(~(Cnt + 1)), Vec1),
    master([get_task(ReqTry, ReqG, ReqPE, NReqTS)
      |ReqTask], GiveTask, Vec1, NewVec)
  ; ReqTry = first → NRG := ReqG + 1,
    set_vector_element(Vec, ReqPE, _, ng(NRG), Vec1),
    Info = info(Ack, NRG, ReqPE, NReqTS),
    master1(Ack, NRG, ReqPE, NReqTS, ReqTask,
      GiveTask, Vec1, NewVec)
  ; otherwise; true →
    Info = info(Ack, ReqG, ReqPE, NReqTS),
    master1(Ack, ReqG, ReqPE, NReqTS, ReqTask,
      GiveTask, Vec, NewVec)).
```

Fig.10 Master process *master*/4

提案は反古にしなければならない．

そのため，*Vec*中に記録している*GivePE*の世代番号*CurGG*と，*GivePE*が*give_task*発行時に有していた世代番号*GiveG*とが比較検査される．*CurGG*>*GiveG*の場合は，*GivePE*の*get_task*が*give_task*を追い越して受信されたことを表しているので*give_task*メッセージの方は棄却される．*GivePE*と*ReqPE*が一致する場合も同様である．このとき，*get_task*メッセージの方は，再度，新たな*give_task*メッセージとの対を試みるため，*ReqTask*ストリームの先頭に戻している．

以上の条件以外の場合，*GivePE*から*ReqPE*へのタスク移動が一応実行可能である．そこで，*give_task*メッセージ中の変数*Info*に，*info*(*Ack*,...)を書き込み，*GivePE*側に現時点で実際に移動可能なタスクがあるか否かを確認する．*Ack*が*GivePE*から返答されるはずであるから，*master*1プロセスでこれを待ち受ける．*master*1プロセスは，*GivePE*からの*Ack*に応じて，*Vec*の適切な設定と同時に*master*の再帰呼び出しを行なう．

以上，一個のタスク移動に係わる二つのPEとマスタープロセス間でのメッセージのやりとりを**Fig.11**に示す．

5. 実　　験

N逐次方式による並列化効果を循環割付け方式と比較評価するため，定理証明系のベンチマーク集(TPTPライブラリ[3])等から非ホーン問題を選び，分散メモリ型の並列推論機PIM/m-128PE上でMGTPのOR並列実行の実験を行った．

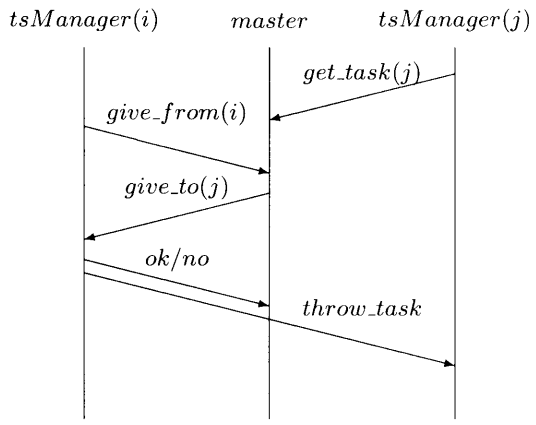


Fig.11 The protocol for task transfer

実行結果を**Table 1**に示す．Cntsはタスク移動総数，Timeは実行時間である．問題はいずれも充足不能な節集合であり，SYN002-1は完全2分木，SYN009-1やTEST2-445は完全3分木，SYN036-4は枝分かれが2～4となる非ホーン節のほか，ホーン節も多く含まれており，枝分かれに比べて高さの高い木を作るような節集合である．

本方式のタスク移動総数は，SYN036-4を除くと循環割付け方式に比べ2桁程度低減化されている．実行時間について3割～1/3倍程度の短縮にしかならないのは，むしろPIM/mの設計が細粒度並列処理のために極めてよく最適化されていることを示唆しているというべきであろう．

しかしながら，**Fig.12**のxamonitor画面[†3](問題TEST2-445 on PIM/m 128PE)でわかるように，通信オーバヘッド(棒グラフで濃い部分)に関する両方式の差は歴然としている．

SYN036-4でN逐次方式による改善が顕著でない理由は，**Fig.13**のxrmonitor画面[†4]に示すように，ホーン節によるモデル候補拡張が著しく長い(分岐不可能な)枝を担当するPEが数個存在するためである．しかし，循環割付け方式(**Fig.13**(a))に比較すると，N逐次方式(**Fig.13**(b))では上記以外のPEは一斉に終了できている

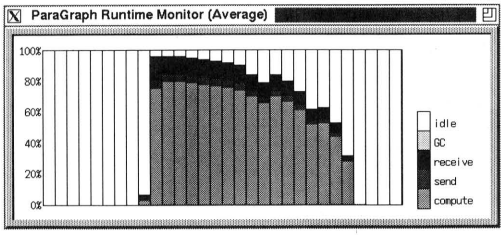
Table 1 Performance comparison between cyclic and N-sequential methods

	Cyclic		N-sequential	
Problem	Cnts	Time	Cnts	Time
SYN002-1.020	43782	28709	987	7949
SYN009-1	19683	8413	317	3033
SYN036-4	1037	23204	713	19489
TEST2-445	26559	38329	1049	27188

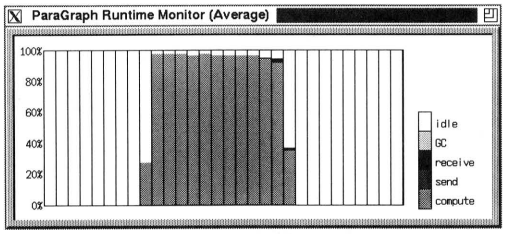
No. of PEs: 128, Time: msec

†3 横軸に時間，縦軸に全PEの処理内容の比率を示す．

†4 横軸に時間，縦軸に各PE毎の稼働率を示す．

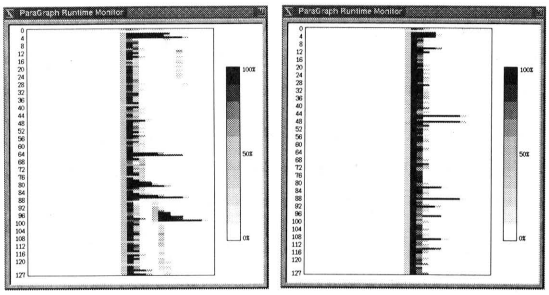


(a) Cyclic allocation method



(b) N-sequential method

Fig.12　Snapshot of "xamonitor" on PIM/m



(a) Cyclic allocation　(b) N-sequential method

Fig.13　Snapshot of "xrmonitor" on PIM/m

ので，良好な負荷分散が図られていることがわかる．

N逐次方式の台数効果(PIM/m 1〜128PE)を，**Fig.14**に示す．すべての問題でほぼ線形の台数効果が得られており，通信オーバヘッドが極めて小さく，実質計算に係わる稼働率が極めて高いという本方式の特徴を如実に示している．

SYN036-4で32PE以上のとき台数効果が劣化しているのは，前述の理由による．また，SYN009-1で128PEのとき逆に実行時間が延びるのは，問題規模が小さく並列化のオーバヘッドが顕著になるからである．

我々は，さらに共有メモリ型の汎用並列機Cray SuperServer6400(20PE) を用いた実験も試みたが，循環割付け方式は負荷分散化により逆に著しく計算時間が増大し，比較の対象にならない．

一方，N逐次方式は，一般には細粒度並列処理に適さないと考えられていた上記汎用並列計算機上での並列実行においても有効であることが確認された．ただし，KL1ゴールの優先度プラグマをPIM/mと同じに設定した場合，プログラムと問題が同じでも試行ごとに実行時間に

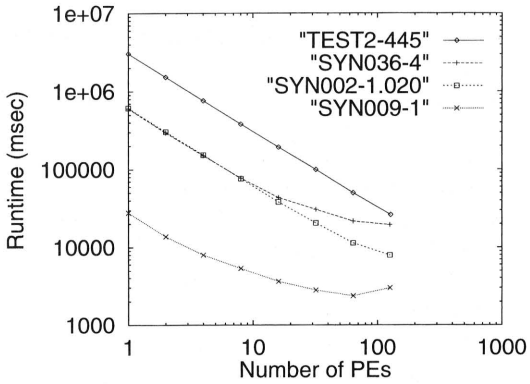


Fig.14　Execution times on PIM/m(1–128PE)

かなり大きなバラツキが認められた．それでも，試行中最良の場合だけを見ればPE数Nの増大に従って単調に計算時間が減少するという台数効果が得られている．優先度プラグマについては，計算機構成に応じて適正な値に調整する必要があろう．

6. おわりに

本論文では，負荷分散を適切な粒度で行い，PE間の通信オーバヘッドを最小限に抑える動的な並列処理方式としてN逐次方式を示した．

ある種のアルゴリズム（MGTPにおけるFUP計算[4]等）の中には，タスク木の根の近くで分割された大きな部分タスク木群は複数のPEによって幅優先的に並列実行可能だが，葉に近い小さな部分木については1PE内での逐次実行による深さ優先的な実行が望ましいというものがある．N逐次方式は，そのような準逐次的アルゴリズムに対して一層その真価を発揮する．

本方式の実現は，もとよりMGTPのOR並列実行を念頭においたため，木型プログラムに限定した形で示したが，より一般のプログラムに対しても適用可能である．特に汎用並列論理型言語klicによって記述されるプログラムに対しては，わずかな修正で容易に適用可能となるので，タスク管理部分と負荷分散制御プロセス部分を標準化し，ライブラリとして整備することを検討中である．

参考文献

1) 長谷川 隆三, 藤田 博: MGTP: 並列論理型言語 KL1 によるモデル生成型定理証明系, 情報処理学会論文誌, Vol. 37, No. 1, pp. 1–12, 1996.
2) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, 1990.
3) Sutcliffe, G., Suttner, C. and Yemenis, T.: The TPTP Problem Library, *Proc. CADE-12*, pp. 252–266, 1994.
4) 松本 誉史, 越村 三幸, 久保山 哲二, 長谷川 隆三: MGTP におけるケース分割の重複削除手法とその評価, 九州大学大学院システム情報科学研究科報告, Vol. 2, No. 1, pp.75–80, 1997.