打順最適化問題の高速化手法

大澤 清[†] 合田 憲人[†]

本稿ではグリッド環境のような計算資源の性能が不均一な環境に対して適応的ジョブ割当て手法を用いた打順最適化問題の高速化手法について述べる．打順の最適化問題はその打順により得られる期待得点を目的関数として定義され，その期待得点は D'Esopo and Lefkowitz 進塁モデルに基づく確率計算により算出される．この計算をすべての打順について行うため，Ninf-G を用いてグリッド環境上で並列に実行することで計算の高速化を図った．また高速化手法として計算パラメータの複数打順間での共通利用，各打者の守備能力を考慮した計算対象打順の選別を行った．さらに負荷分散方式として計算対象打順を適応的に計算ノードに割り当てる方法と計算開始時の試行計算結果に基づき静的に割り当てる方法を実装し比較を行ったところ，グリッド環境においては適応的に割り当てる方法が有効であることが確認された．また適応的に割り当てる場合の最適な計算粒度すなわち打順の分割数をモデル化により決定し，計算資源の有効利用を図った．以上の高速化手法を適用し 4 拠点の PC クラスタを利用して 134,991,360 通りの打順の組合せについて期待得点を計算し最適な打順を探索した結果，1 台の計算ノードで理論的に 265 日程度を要する処理が 2,522 秒で実行可能となった．

Speed-up Techniques to Find an Optimal Batting Order

Kiyoshi Osawa[†] and Kento Aida[†]

In this paper, we propose speed-up techniques to find an optimal batting order in a baseball team by adaptive job assignment, which is suitable for heterogeneous computing resources like the Grid. The objective function of the optimization problem to find an optimal batting order is to calculate expected runs, and the expected runs are computed using the D'Esopo and Lefkowitz runner advancement model. The proposed technique parallelizes computation for performances of all batting orders using Ninf-G, so that the computation time is reduced on the Grid. Furthermore, the proposed techniques improve performance by sharing parameters among multiple batting orders and by choosing feasible batting orders, which fill all fielding positions. We implemented two load balancing algorithms. The first algorithm adaptively assigns computation of batting orders to computing nodes, and the second algorithm statically assigns according to measured computational power at run time. From the comparison with these algorithms, we found the effectiveness of the adaptive algorithm on the Grid. In order to improve the performance of the load balancing, we estimated the optimal computational granularity, or the number of batting orders, by using the model of the computational complexity. The experimental results show that the proposed techniques compute the optimal batting order among 134,991,360 batting orders in 2,522 seconds on the Grid testbed over 4 sites, while it theoretically takes 265 days on a single computer.

1. はじめに

野球チームの最適な打順を求める手法としてマルコフ連鎖を用いた手法[2] が知られている．この手法では，打者の集合から構成されるすべての打順について D'Esopo and Lefkowitz 進塁モデルを適用してその期待得点を算出する．しかし，少なくとも $9! = 362{,}880$ 通りの打順について期待得点を算出し，その中で最大の期待得点をあげる打順を最適な打順とする組合せ最適化問題[8] を解くことになるため，その計算量が単体の PC にとっては比較的大きいという問題がある．たとえば文献 2) では，一部の攻撃力の低い打者を打順内で固定して打順の組合せ数を減らして全体の計算時間を短縮し，その中で最大の期待得点を与える打順を最適な打順と見なす手法がとられている．しかしチームの勝敗を予測するうえで，その手法により得られる打順の期待得点と真に最適な打順の期待得点との誤差による影響は，試合数が増えるにつれ大きくなる恐れがある．

本稿ではすべての打順の組合せについて期待得点を算出して真に最適な打順を決定するために，以下の高

† 東京工業大学
Tokyo Institute of Technology

速化手法を提案するとともに，提案手法をグリッド上に実装し性能評価を行った結果について報告する．

- 打順の循環に着目した計算パラメータの共通利用
- 守備位置の充足可能性判定による打順の選別

提案手法では，複数の異なる打順の期待得点を算出する際に共通に利用可能な計算パラメータを求め，これらを共通に利用することにより計算時間の短縮を図る．また，打順を構成する打者の守備位置に関する充足可能性判定を行うことにより，無用な計算を省く．

また，提案手法のグリッド上の実装では，負荷分散を効率良く行うことが重要となる．そのため本稿では以下の負荷分散方式を実装し，その性能を比較した．

- 適応的ジョブ割当てによる負荷分散
- 試行計算による性能見積りに基づく負荷分散[4),10)]

提案した高速化手法を適用した結果，計算パラメータの共通利用により 8.81 倍の速度向上が図られ，充足可能性判定によるオーバヘッドを抑えつつ計算対象打順の選別が可能となった．さらに計算ノード性能が不均一なグリッド環境においては適応的ジョブ割当てによる負荷分散方式を用いることで，試行計算による性能見積りに基づく負荷分散手法を用いた場合よりも計算資源の有効利用が図られることが示された．また，適応的ジョブ割当てでは計算ノードに割り当てるジョブを適切に分割することが重要であるが，ジョブの分割手法と実行時間に関する解析を行い，理論的な実行時間と実際の実行時間の傾向はよく一致することが確認された．さらに 4 拠点に分散配置されたグリッド環境において提案手法を適応的ジョブ割当てを用いて適用した結果，高速化手法を適用せずに単一計算ノードで実行した場合に論理的には約 265 日を要する 134,991,360 通りの打順の期待得点計算が 2,522 秒で実行可能となることが確認された．

以下，2 章では本稿が対象とする問題である打順最適化問題について述べ，3 章では提案手法である期待得点算出の高速化手法について述べる．4 章では提案手法をグリッド上に実装する際の負荷分散方式を示し，5 章でその性能評価を行い，6 章でまとめを行う．

2. 最適打順決定手法

2.1 D'Esopo and Lefkowitz 進塁モデル

D'Esopo and Lefkowitz 進塁モデルでは野球の攻撃をアウト数と走者状況ごとで分類し，25 通りの状態を定義している（アウト数 3 種類 (無死，一死，二死) × 走者状況 8 種類 (無走者，一塁，二塁，三塁，一二塁，一三塁，二三塁，満塁) + 三死における攻撃終了状態）．これより各行が “ある打者が打席に入った時点の状態” を表し，各列が “ある打者が打撃を行った結果の状態” を表す 25 × 25 状態遷移行列 P が定義される．さらに各打者の打撃による走者の進塁状況とその確率を以下のように定義する．

- 凡打（Out）：走者は進塁せず，アウト数を 1 つ増やす（確率：p_O）．
- 四球（Walk）：押し出される走者のみ 1 つ進塁（p_W）．
- 単打（Single）：一塁走者は二塁へ，その他の塁の走者は生還（p_S）．
- 二塁打（Double）：一塁走者は三塁へ，その他の塁の走者は生還（p_D）．
- 三塁打（Triple）：すべての塁の走者が生還（p_T）．
- 本塁打（Homerun）：すべての塁の走者と打者が生還（p_H）．

これより行列 P は 8×8 の部分行列 A，B と 8×1 のベクトル F より

$$P = \begin{pmatrix} A & B & 0 & 0 \\ 0 & A & B & 0 \\ 0 & 0 & A & F \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1)$$

と定義される．打者が打席に入った時点の状態を表す P の各行は第 1 行から第 8 行が無死，第 9 行から第 16 行が一死，第 17 行から第 24 行が二死，第 25 行が三死の攻撃終了状態に対応する．打撃を行った結果を状態を表す各列も同様となる．このとき行列 A，B とベクトル F は

$$A = \begin{pmatrix} p_H & p_S+p_W & p_D & p_T & 0 & 0 & 0 & 0 \\ p_H & 0 & 0 & p_T & p_S+p_W & 0 & p_D & 0 \\ p_H & p_S & p_D & p_T & p_W & 0 & 0 & 0 \\ p_H & p_S & p_D & p_T & 0 & p_W & 0 & 0 \\ p_H & 0 & 0 & p_T & p_S & 0 & p_D & p_W \\ p_H & 0 & 0 & p_T & p_S & 0 & p_D & p_W \\ p_H & p_S & p_D & p_T & 0 & 0 & 0 & p_W \\ p_H & 0 & 0 & p_T & p_S & 0 & p_D & p_W \end{pmatrix}$$

$$B = p_O I$$

$$F = (p_O, \ldots, p_O)^T$$

である．I は単位行列を，^{T} はベクトルの転置をそれぞれ表す．

A，B の各行，各列はそれぞれ上端，左端から “無走者，一塁，二塁，三塁，一二塁，一三塁，二三塁，満塁” の状態に対応する．A は打者の打撃結果が安打または四球である場合の遷移確率を表し，このときアウト数は増えることなく走者は生還または進塁する．たとえ

ばある打者が無死無走者の状態で打席に入った場合，打撃結果により無死一塁の状態に遷移する確率は第 1 行第 2 列の要素に対応し，D'Esopo and Lefkowitz 進塁モデルにおいてその状態遷移を実現する打撃結果は単打もしくは四球であるためその値は $p_S + p_W$ となる．式 (1) ではアウト数によらず打者の各打撃結果の確率は一定と仮定しているため，一死，二死における状態遷移確率も A により表される．B は無死，一死における打者の打撃結果が凡打の場合の遷移確率を表し，このときアウト数は 1 つ増え，走者は進塁しない．ベクトル F は凡打により二死から攻撃終了状態である三死に遷移する確率を表す．実際の野球における併殺打や凡打の間に走者が進塁する打撃については，今回は簡単のために考慮していない．

N 人の打者の集合を $S = \{b_1, b_2, \ldots, b_k, \ldots, b_N\}$（$k$ は打者インデックス）とすると，打者ごとに過去の成績から求められる各確率 p_O，p_W，p_S，p_D，p_T，p_H を用いて状態遷移行列 P_k を定義し，それらを打順に従って掛け合わせることで攻撃の状態遷移がシミュレートできる．

2.2 1 イニング中にあげる期待得点算出手法

1 イニング中にあげる期待得点を状態遷移行列 P_k を用いて算出する方法を以下に示す．P_k を $P_k^{(0)}$，$P_k^{(1)}$，$P_k^{(2)}$，$P_k^{(3)}$，$P_k^{(4)}$ に分解する．$P_k^{(r)}$（$r = 0$，1，2，3，4）は P_k の各確率のうち，打者 k の打撃により得点が r 点入る要素を抜き出したもので，

$$P_k = P_k^{(0)} + P_k^{(1)} + P_k^{(2)} + P_k^{(3)} + P_k^{(4)}$$

が成り立つ．1 イニング中にあげる得点を行，アウト数および走者状況で定まる 25 通りの状態を列とする $(R_{\max} + 1) \times 25$ 行列 U を得点と状態の確率分布を表す行列とする．ここで $R_{\max}$ は 1 イニングであげうる最大得点とする．攻撃開始時すなわち 0 人の打撃終了時においては無得点，無走者であることから U の初期値 U_0 は第 1 行第 1 列のみが 1 であり，残りの成分はすべて 0 の行列となる．イニング開始から n 人の打撃終了時における U を U_n で表すと，これは以下の式で定められる．

$$U_n|_i = \sum_{r=0}^{4} U_{(n-1)}|_{(i-r)} P_k^{(r)} \quad (i = 1, 2, \ldots, R_{\max} + 1) \quad (2)$$

$U_n|_i$ は U_n の第 i 行を表し，k は打者インデックスである．k は n とともにインクリメントされるが，周期 9（= 打者数）でループする．このようにして U_n を定めると，$n \to \infty$ のとき U_n の第 25 列（右端）の要素の総和は限りなく 1 に近付く．このとき $u_{i,j}$ を行列 U の第 i 行，第 j 列の成分とすると 1 イニング中にあげる期待得点 ER_{Inn}（Expected Runs）は

$$ER_{Inn} = \sum_{i=1}^{R_{\max}+1} (i-1) \times u_{i,25}$$

で表される．打者の集合 S より打順を構成し，各打者について状態遷移行列 P_k を定義して上記の計算を行うことで 1 イニング中にあげる期待得点 ER_{Inn} が算出できる．

2.3 1 試合中にあげる期待得点算出手法

あるイニングの先頭打者が i 番打者で，その次のイニングの先頭打者が j 番打者である確率を t_{ij}，その場合の期待得点を e_{ij} とする．これらの値は式 (2) の乗算ごとに U_n 最右端の列の要素から算出することができる．このとき m 回の攻撃が n 番打者で始まる確率 $p_{m,n}$ とその場合の初回からの合計得点 $a_{m,n}$ は

$$p_{m,n} = \begin{cases} 1 \quad (m = 1,\, n = 1) \\ 0 \quad (m = 1,\, n = 2, 3, \ldots, 9) \\ \displaystyle\sum_{k=1}^{9} p_{m-1,k} t_{kn} \\ (m = 2, 3, \ldots, 10,\, n = 1, 2, \ldots, 9) \end{cases} \quad (3)$$

$$a_{m,n} = \begin{cases} 0 \quad (m = 1,\, n = 1, 2, \ldots, 9) \\ \displaystyle\sum_{k=1}^{9} \frac{p_{m-1,k}}{p_{m,n}} t_{kn} (a_{m-1,k} + e_{kn}) \\ (m = 2, 3, \ldots, 10,\, n = 1, 2, \ldots, 9) \end{cases} \quad (4)$$

で表される．これらより 1 試合の期待得点 ER_G は

$$ER_G = \sum_{k=1}^{9} p_{10,k} a_{10,k} \quad (5)$$

として算出される．

3. 期待得点算出の高速化手法

本稿では，期待得点算出について以下に示す高速化手法を提案する．高速化手法を含めた最適打順決定手法のアルゴリズムを**図 1** に示す．

3.1 計算パラメータの共通利用による高速化

野球では 9 人の打者が循環して打席に立つことを考慮すると，集合 S 内の打者を b_k で表した場合に，たとえば打順 $[b_1 b_2 \ldots b_9]$ について求めた t_{ij} と e_{ij} は循環シフトした打順 $[b_2 b_3 \ldots b_9 b_1]$ の期待得点を求める際にも利用することができる．具体的には式 (3) で $p_{m,n}$（$m \geq 2$）を $p_{1,1} = 1$，$p_{1,n} = 0$（$n = 2, 3, \ldots, 9$）すなわち初回の攻撃はつねに 1 番目の打者 b_1 から始まるとして求めていたが，$p_{1,2} = 1$，

各選手の $P_k^{(r)}$ を定義
for 打者の集合 S から選手 9 人を抽出（${}_NC_9$ 通り）**begin**
  if 選手 9 人が守備位置を充足するか判定 **then** /* 高速化適用時のみ */
    for 9! 通りの打順を構成 **begin** /* 高速化適用時：8! 通り */
      do begin
        打順に従って U_n を U_{n-1} と $P_k^{(r)}$ より定義（式 (2) 参照）
        U_n 最右端列より t_{ij} と e_{ij} を計算（2.3 節参照）
        if U_n 最右端列要素和 ≈ 1 **break;**
      end do
      t_{ij}，e_{ij} より $p_{m,n}$，$a_{m,n}$ を計算（式 (3)，式 (4) 参照）
      期待得点 ER_G を $p_{m,n}$，$a_{m,n}$ より計算（式 (5) 参照）
      $p_{1,n}$ を変化させ循環シフトした 8 通りの
        打順について ER_G を計算 /* 高速化適用時のみ */
    end for
  end if /* 高速化適用時のみ */
end for
ER_G を最大にする打順を最適打順と定義
（最外 **for** ループの内部は独立のため，並列に実行可能）

図 1 最適打順決定アルゴリズム
Fig. 1 The algorithm to find an optimal batting order.

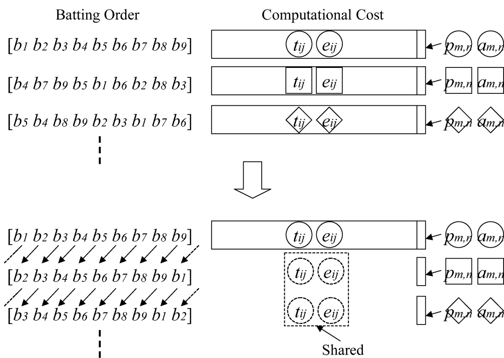

図 2 計算パラメータ t_{ij}，e_{ij} の共通利用による高速化
Fig. 2 Speed-up by sharing common parameters t_{ij} and e_{ij}.

$p_{1,n} = 0$（$n = 1, 3, \ldots, 9$）すなわち初回の攻撃はつねに 2 番目の打者 b_2 から始まるとしてすでに求めた t_{ij} と e_{ij} を用いて $p_{m,n}$（$m \geq 2$）を求めることができる（**図 2** 下部）．同様に打順 $[b_3 b_4 \ldots b_9 b_1 b_2]$ 等の期待得点を求める際にも t_{ij} と e_{ij} を利用することができる．$p_{m,n}$ と $a_{m,n}$ の計算コストは t_{ij} と e_{ij} のそれに比べて非常に小さく，t_{ij} と e_{ij} を共通して利用できる打順は 9 通りであるため，これらの打順について連続して期待得点の算出を行うことで，すべてのパラメータを求める場合（図 2 上部）と比較して 9 倍近くの速度向上が期待される．

3.2 守備位置の充足可能性判定による打順の選別

前節までに述べた期待得点の算出方法を単純に実装した場合，打者の集合 S の要素数すなわち打者数 N が打順を構成しうる最小の値（$= 9$）の場合においても $9! = 362{,}880$ 通りのすべての打順の組合せについて期待得点を算出する必要があった．3.1 節で示したパラメータの共通利用による高速化手法を用いるとその計算時間はおよそ 1/9 に削減されるが，打者数を増やしてより大規模な問題を扱う場合，$N = 10$ 人の場合は ${}_{10}P_9 = 3{,}628{,}800$ 通り，11 人の場合は 19,958,400 通り，12 人の場合は 79,833,600 通りの打順について計算することになり，計算量が大きくなることは避けられない．しかし実際の野球において打者は指名打者を除き守備を行う必要があり，各打者について技術的に守れる守備位置と守れない守備位置が存在するため，S から 9 人を選出した際，守備が破綻しないようにする必要がある．S から抽出されたある打者の組合せですべての守備位置が充足可能かどうかを判定するため，各打者について守れる守備位置を示すパラメータを定める．期待得点を算出する前にその組合せに含まれる打者ですべての守備位置が充足可能かどうかを判定し，可能な場合のみ期待得点を算出することで，計算量の増加を抑えることが可能になる．以下にその判定を行うための基本的な考えについて述べる．

3.2.1 守備可能なポジションを示すパラメータ

各打者について各守備位置に就くことが可能かどうかを表すパラメータ列を用意し，守備可能の場合は 1，そうでない場合は 0 を与える．守備力（守備範囲，肩

Position

Batting Order	P	C	1B	2B	3B	SS	LF	CF	RF
b_3	0	1	1	0	0	0	0	0	1
b_5	0	0	0	0	1	0	0	0	0
b_9	0	0	0	0	0	0	1	0	1
b_1	1	0	0	0	0	0	0	0	0
b_4	0	0	0	1	1	1	0	0	0
b_{10}	0	0	0	0	0	0	1	0	0
b_2	0	1	0	0	0	0	0	0	0
b_8	0	0	0	0	0	0	1	1	1
b_6	0	0	0	0	0	1	0	0	0

図 3 ある打順における打者が守備可能なポジション
Fig. 3 The feasible fielding position table.

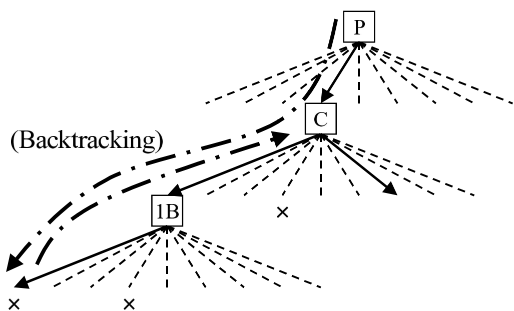

図 4 守備位置の充足可能性判定
Fig. 4 The method to determine satisfiability for fielding positions.

の強さ，捕球の確実性）については考慮しない．ある打順に沿ってパラメータ列を並べた表を**図 3** に示す．ただし表の各行は打者を，各列は守備位置を表している．

3.2.2 守備位置の充足可能性判定手法

図 3 の表から守備位置の充足可能性判定を行う手法を以下に示す．図 3 の表において投手を表す左端の列を上から走査し，守備可能を表す「1」が i 行目に現れた場合は**図 4** において「P」で表されている投手のノードの左から i 番目の枝（図 3 の表の i 番打者に対応）を実線の矢印で表す．図 4，**図 5** の各ノードは守備位置を，各ノードから伸びる 9 本の枝のうち左から i 番目の枝が実線の矢印の場合は i 番打者がその守備位置に就けることを表し，点線の場合は守れないことを表している．実線の矢印は次の（図 4 中で下方の）守備位置に就ける打者の探索に利用可能であるが，図 4 では投手のノードの左から 4 番目の枝から「C」で表されている捕手のノードに移ったため，同ノードから 4 番目の枝を再び利用しないように，すなわち四番打者を探索の候補から外すために「×」のラベルを付与する．捕手のノードにおいても同様の処理を行い，

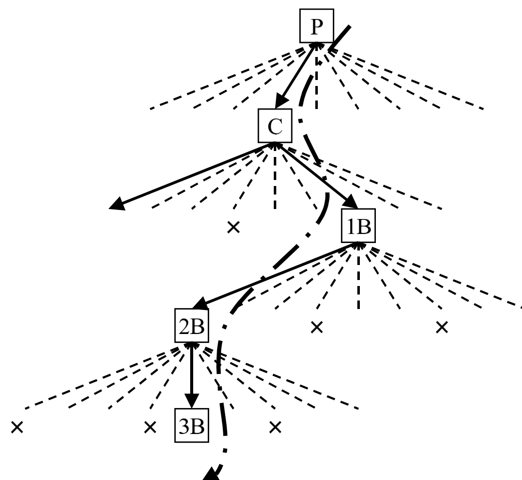

図 5 守備位置の充足可能性判定（バックトラッキング後）
Fig. 5 The method to determine satisfiability for fielding positions (after backtracking).

これを繰り返す．実線の矢印で表され，その矢印の先に「×」のラベルが付与されていない枝を “利用可能” な枝と表すことにすると，図 4 では「1B」で示される一塁手のノードにおいて利用可能な枝が存在しない（実線の矢印の先に「×」のラベルが付与されている）ため，N-Queens 問題等で用いるバックトラッキングの手法[6] を用いて捕手のノードまで戻り，再び探索を繰り返す（図 5）．「RF」で示される右翼手のノードにおいて利用可能な枝が存在する場合，その打順に含まれる打者で全守備位置が充足可能である．逆にバックトラッキングを繰り返し，最上位にある投手のノードにおいて利用可能な枝が存在しない場合は，充足不可能と判定される．

守備位置を考慮しつつ最適な打順を決定する従来手法[7] では，あらかじめ人間が守備位置の充足性を判定したうえで期待得点の計算を行っていたため，多くの組合せについて最適な打順を決定するのは困難であった．本研究では以上の手法を用いることで守備位置の充足性をプログラムにより判定し，かつ打者数の増加による組合せ数の増加を抑えることが可能となった．

4. グリッド環境上における負荷分散方式

本稿が対象とする問題では個々の打順についての計算は独立に実行できるため，これらを複数の計算機に割り当てることにより計算時間の短縮が期待できる．計算プラットフォームが単一拠点内の均一な計算ノードから構成される PC クラスタの場合，打順ごとの期待得点計算時間に大きな差はないため，組合せ数を計算ノード数で均等に分割して割り当てれば負荷は均衡

```
void AdaptiveAssign()
  for i := 0 to np - 1 do begin                    np は計算ノード数
    jid := GetJobID();                             ジョブ ID を取得
    CallAsync(handle[i],                           handle[i] は計算ノード i に対応するハンドラ
      ComputeER(jid, &er[i], &order[i][0]));       ER は期待得点（Expected Run）
  end for
  while jid <> END do begin
    WaitAny(&ret_handle);                          ジョブの終了を待つ
    nid := Handle2NodeID(ret_handle);              ハンドラから計算ノード ID を取得
    if er[nid] > er_best then                      期待得点が最大の場合，更新
      er_best := er[nid];
      order_best[] := order[nid][];
    end if
    jid := GetJobID();                             ジョブ ID を取得（打順の各分割の ID）
    if jid <> NONE then                            割り当てるジョブが存在する場合
      CallAsync(ret_handle,
        ComputeER(jid, &er[nid], &order[nid][0]));
    end if                                         jid == NONE の場合，未終了のジョブは
  end while                                          存在するが割り当てるジョブは存在しない
  WaitAll();                                       ただし，故障計算ノードについては除く
end
integer GetJobID()
  if すべてのジョブが終了 then GetJobID := END;
  if すべての未終了ジョブが最大多重度で実行中 then GetJobID := NONE;
  GetJobID := 未終了で多重度が最小のジョブ ID;
end
```

図 6 適応的ジョブ割当てアルゴリズム
Fig. 6 The adaptive algorithm to assign jobs.

する．しかし規模の大きな問題に対してグリッド環境のような複数拠点の PC クラスタを用いて計算を行う状況では計算ノードの性能が拠点間で異なるため，負荷の均衡を図る必要がある．さらに計算ノードの数が増すにつれて，一部の計算ノードがその負荷により他の計算ノードに比べて処理速度を大きく低下させるという状況も起こりうる．そのような状況において計算時間の増加を抑えるような負荷分散を行う必要がある．またグリッドのような環境である計算ノードが故障した場合に全体の処理が中断されないようにするために，計算の多重化を行う必要がある．

4.1 多重化と適応的ジョブ割当てによる負荷分散

計算の多重化と適応的ジョブ割当てによる負荷分散方式を用いた打順最適化問題の実装方法を以下に示す．最初に全守備位置を充足する全打順を一定数に分割し，各分割中の打順の期待得点を計算する処理を 1 つのジョブとする．各ジョブには ID が与えられ，ID が小さいジョブから各計算ノードに割り当てられ，ある計算ノードのジョブが終了したらその計算ノードに対し再びジョブが割り当てられる．再び割り当てるジョブは

(1) いまだに実行されていないジョブ

(2) 実行中のジョブで最も多重度の低いジョブ

の優先順位で選択される．ただし同一の優先順位となるジョブが複数存在する場合は，ジョブ ID の小さなジョブから選択される．また，最終的な段階で全計算ノードが 1 つのジョブに集中する状況を避けるために最大多重度を指定する．具体的には実行中のジョブがすべて最大多重度で実行されている状況下で，ある計算ノードがジョブを終了した場合はその計算ノードに対して新たにジョブを割り当てないようにする．基本的なアルゴリズムを**図 6** に示す．

この割当て方法により，計算資源の性能が不均一な環境では高速な計算資源により多くのジョブが割り当てられることで負荷分散が図られる．また，ジョブを多重に割り当てることにより，あるジョブを実行中の計算ノードが故障した場合でも他の計算ノードに割り当てられた同じジョブの計算結果を得られるため，故障した計算ノード上で実行されていたジョブの計算結果を失うことを防ぐことができる[9]．この場合，故障が発生した計算ノードを切り離し，以後の計算で故障計算ノードを使用しないようにする必要があるが，これらの処理は Ninf-G を用いてプログラミング可能で

あるほか，Condor [3] 等のミドルウェアを用いて実現可能である．

4.1.1 打順の最適分割数

4.1 節で示した負荷分散手法において全打順を一定数に分割する際，その分割数，すなわち 1 つのジョブの大きさが計算時間に影響を与えることになる．分割数が小さい場合ジョブ呼び出しの回数は少なくなるが，1 つのジョブの計算時間が長くなるため，分割数が大きい場合に比べ全体の計算の最終的な局面で多重化により生成されるジョブの数が増加し [5]，その冗長なジョブの計算時間が全体の計算時間に影響を与える．逆に分割数が大きい場合，1 つのジョブの計算時間は短くなるが，ジョブ呼び出しの回数が増えることになる．提案手法の実装では，実行時間に関係するこれらの要因をモデル化し最適な打順の分割数を求めている．

モデル化を行うため，パラメータを以下のように定める．

T：全打順数（ord）
D：打順の分割数
c_i：クラスタ i の計算ノードの処理能力（ord/sec）
n_i：クラスタ i の計算ノード数
t_{oh}：ジョブ呼び出しに要する時間（sec）

同一クラスタ内の計算ノードの性能は均一であるとすると，このときクラスタ i 中のある計算ノードが 1 つのジョブに要する処理時間 s_i（sec）は

$$s_i = \frac{T/D}{c_i} + t_{oh} \qquad (6)$$

で表され，クラスタ i 全体の処理能力 C_i（ord/sec）は

$$C_i = \frac{T/D}{s_i} \times n_i \qquad (7)$$

で表される．全クラスタの処理能力は $\sum C_i$（ord/sec）で表され，理想的な処理時間は $T/\sum C_i$（sec）となる（**図 7**）．

しかし実際の処理時間は理想的な処理時間に加え，計算開始時にすべてのジョブ呼び出しに要する時間の影響と計算終了直前に呼び出された冗長なジョブの計算時間の影響を受ける．

計算開始時に N 番目のジョブを N 台目の計算ノードで呼び出す時刻は開始から $t_{oh}(N-1)$ 後となり，各計算ノードに理想的にジョブが渡されたとすると，ジョブ呼び出しに要する時間がジョブの計算時間に比べて非常に小さい状況における処理の終了時刻は各計算ノードの開始時刻の平均値 $t_{oh}(N-1)/2$ を理想的な処理時間に加えた時刻となる（**図 8**）．ただし，本稿で用いた実験環境では数千秒の全体の処理時間に対しこのジョブ呼び出しに要する時間は十数秒であり，その影響は小さい．これに対して計算終了直前に計算の多重化により呼び出された冗長なジョブの計算時間の影響は最悪の場合で全計算ノードのうち最も計算能力の低い計算ノードの 1 つのジョブの計算時間（$\max_i s_i$）で表され（**図 9**），分割数が小さく 1 つのジョブの計算時間が長い状況では無視できなくなる．

これらの影響を考慮した実際の処理時間は上記のパ

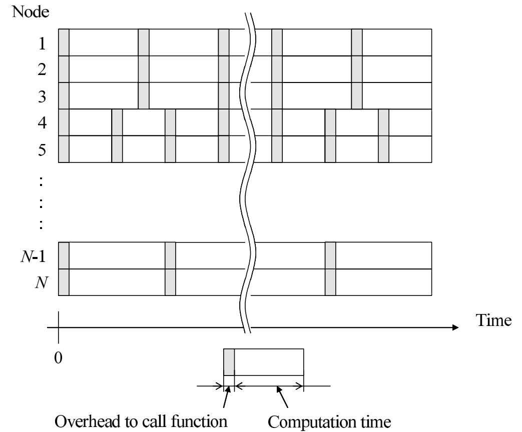

図 7 理想的なジョブ割当てが行われた例
Fig. 7 An example of optimal job assignment.

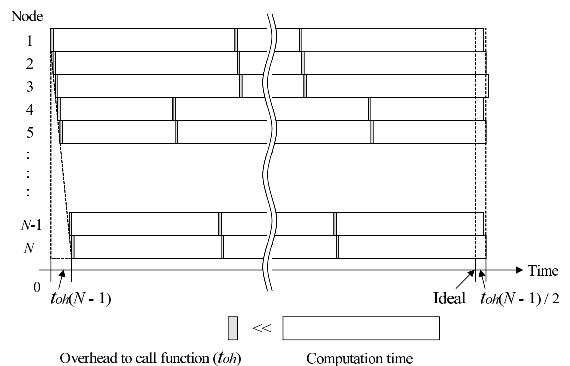

図 8 計算開始時のジョブ呼び出し時間を考慮した実行時間
Fig. 8 Execution time including initial job assignment time.

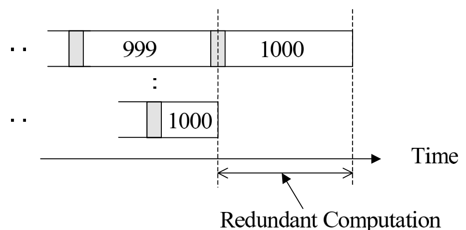

図 9 計算終了直前に呼び出された冗長なジョブによる影響
Fig. 9 Effect of the last job called redundantly.

ラメータを用いると

$$T/\sum C_i + t_{oh}(\sum n_i - 1)/2 + \max_i s_i \ \text{(sec)} \quad (8)$$

で表される．式 (8) は分割数が小さい場合の冗長なジョブに要する時間の影響を強く受けるため，分割数が大きくなるにつれて処理時間は短くなる．しかし分割数が非常に大きくなり 1 つのジョブに要する時間が短くなると，1 つのジョブの処理時間よりもすべての計算ノードに対し逐次的にジョブ呼び出しを行う時間が長くなり，計算ノードにおいてジョブ終了から次のジョブ呼び出しを行うまでのアイドル時間が発生する．この状況を考慮した全体の処理時間は

$$\max\{T/\sum C_i + t_{oh}(\sum n_i - 1)/2 + \max_i s_i, \; t_{oh}D\} \ \text{(sec)} \quad (9)$$

で表される．

4.2 試行計算による性能見積りに基づく負荷分散

4.1 節で示した適応的負荷分散手法と比較を行うために，試行計算による性能見積りに基づく負荷分散方式[4),10)] を実装した．この負荷分散方式では，初めに試行として少量の計算を各計算ノード上で実行することにより計算ノードの性能を評価し，その結果より全打順の計算において各計算ノードへ割り当てる打順の数を決定する．具体的には以下の手順で負荷分散を行った．

最初に N 人の打者の集合 S から構成しうる $_NP_9$ 通りの打順のうち，守備位置を充足する打順の数 $sat(N)$ を求める．ただし，守備位置充足性は打者の順列ではなく組合せによって判定を行うことが可能であり，$_NC_9$ 通りの打者の組合せについて判定を行い，守備位置を充足する打者の組合せの数に 9!を掛けた値が $sat(N)$ となる．

実際の期待得点算出の前に，各計算ノードにおいて $f \times sat(N)/$(全計算ノード数) 通り（$f < 1$）の打順について期待得点を算出し，その計算に要した時間 $t(k,l)$（拠点 k の計算ノード l における計算時間）を 1 台の計算ノードに集約する．

各計算ノードの計算時間を集約した計算ノードにおいて，式 (10) により各計算ノードに割り当てる打順の数 $ord(k)$ を決定する．

$$C_A = \sum_{k,l} \frac{1}{t(k,l)}$$
$$ord(k) = sat(N) \times \frac{1/\bar{t}(k)}{C_A} \quad (10)$$

ここで，$\bar{t}(k)$ は拠点 k の全計算ノードの平均計算時間であり，拠点 k の各計算ノードに $ord(k)$ の打順が割り当てられることになる．

この負荷分散方式では各計算ノードに対して試行と本番の 2 つのジョブに関する情報を送信すればよく，適応的ジョブ割当てによる負荷分散方式に比べて通信量を大幅に削減できる．ただし，クラスタ内に処理能力の低い計算ノードが存在した場合，そのノードによる計算の停滞により全体の処理時間が増大する恐れがある．またグリッドのような不特定多数の利用者が存在する環境においては，各利用者のジョブの実行によりクラスタの負荷がつねに変動しているため，試行計算による性能見積りが正確ではない状況も考えられる．本稿では 4.1 節で示した適応的ジョブ割当てによる負荷分散方式と試行計算による負荷分散方式をグリッド環境上で実装して比較し，適応的ジョブ割当てによる方式の有効性を検証する．

5. 性能評価

性能評価には産業技術総合研究所グリッド研究センター，東京工業大学小野研究室，東京大学田浦研究室，同志社大学知的システムデザイン研究室の所有する各 PC クラスタで構成されるグリッドチャレンジ 2006[1)] のために用意された計算機環境を使用した．産総研 F32 クラスタは 100 台の計算ノード，小野研 DIS クラスタは 40 台，田浦研 Tau クラスタは 100 台，同志社大 Xenia クラスタは 40 台を使用し，合計 280 台の計算ノードからなるグリッド環境を構成した．同環境上に D'Esopo and Lefkowitz 進塁モデルを実装し，GridRPC ミドルウェアである Ninf-G2.3.0[11)] を用いて並列化を行った．計算のカーネル部分となる行列ベクトル積計算には性能の自動チューニングが施された数値計算ライブラリの ATLAS[12)] を用いた．

5.1 計算パラメータの共通利用による高速化の効果

計算パラメータの共通利用による高速化の効果を示すために，F32 クラスタにおいて共通利用を行った場合と行わなかった場合の実行時間を比較した．集合 S に含まれる打者数は 10 人とし，共通利用による効果のみを測定するために構成しうるすべての打順において守備位置を充足するような打者を用いた．負荷分散手法としてはいずれの場合も適応的ジョブ割当てによる方法を採用し，分割数は 3,000 とした．40 台の計算ノードを使用して測定した結果を**表 1** に示す．循環シフトにより一致する 9 通りの打順について t_{ij} と e_{ij} を共通して利用できるため，ほぼ 9 倍の速度向上が得られている．9 倍にならないのは共通利用できない $p_{m,n}$ と $a_{m,n}$ の計算コストによるものと考えられる．

5.2 守備位置充足可能性判定による高速化の効果

守備位置充足可能性判定による高速化の効果を示す

表 1　計算パラメータの共通利用による高速化の効果
Table 1　The effect of sharing parameters.

共通利用なし	共通利用あり	速度向上比
3,258（sec）	370（sec）	8.81

表 2　守備位置充足可能性判定による高速化の効果
Table 2　The effect of choosing feasible batting orders.

充足可能性判定	打順数	実行時間 (sec)	処理速度 (ord/sec)
あり	2,540,160	396	6,415
なし	3,628,800	535	6,783
速度比（あり/なし）		0.95	

ために，F32 クラスタの計算ノード 30 台を用いて検証を行った．集合 S に含まれる打者として，2005 年の千葉ロッテマリーンズの打者から 10 人を選出した．集合 S から構成される打順の総数は ${}_{10}P_9 = 3{,}628{,}800$ 通りであり，このうち全守備位置を充足する打順の総数は 2,540,160 通りである．充足判定を行った場合と行わなかった場合の実行時間と，1 秒あたりに処理する打順数を**表 2** に示す．なお負荷分散手法としてはいずれの場合も適応的ジョブ割当てによる方法を採用し，分割数は 3,000 とした．

表 2 より計算対象として選別された打順の数にほぼ比例して実行時間が短縮されていることが分かる．4.2 節で述べたように N 人の打者からなる ${}_NP_9$ 通りの打順に対して充足可能性判定処理は ${}_NC_9$ $(= {}_NP_9/9!)$ 通りの打順のみに対して行うため，判定を行った場合と行わなかった場合の処理速度比は 1 に近い値 $(= 0.95 = 6{,}415/6{,}783)$ となり，この処理に要するオーバヘッドは小さいことが示されている．

5.3　適応的ジョブ割当てにおける打順の最適分割数の決定

式 (9) で示した適応的ジョブ割当てにおける分割数と実行時間の関係を実際の環境を用いて検証した．4 拠点 280 台の計算ノードからなるグリッド環境を利用し，集合 S に含まれる打者として，2006 年 3 月に開催された野球の国際大会（WBC）に出場した 14 選手を選出した．各選手の 2005 年度打撃成績，守備可能位置を表すパラメータを**図 10** に示す．ただし，米大リーグに所属する Ichiro の打撃成績は日本で主力選手として出場した 7 年間の平均打率（.359）と 2005 年度の米大リーグにおける打率（.303）との比（1.18）を参考とし，安打数，四球数を 1.2 倍して算出している．

ジョブ呼び出しに要する時間の平均値 t_{oh} とクラスタ i の計算ノードの処理能力 c_i を実測により求めた．

$$t_{oh} = 0.096 \text{ (sec)} \tag{11}$$

Name	AVG	HR	P	C	1B	2B	3B	SS	LF	CF	RF
			Position								
Satozaki	.303	10	1	1	0	0	0	0	0	0	0
Matsunaka	.315	46	1	0	1	0	0	0	0	0	0
Ogasawara	.282	37	1	0	1	0	1	0	0	0	0
Arai	.305	43	1	0	1	0	1	0	0	0	0
Imae	.310	8	1	0	0	1	1	0	0	0	0
Nishioka	.268	4	1	0	0	1	0	1	0	0	0
Iwamura	.319	30	1	0	0	0	1	0	0	0	0
Kawasaki	.271	4	1	0	0	0	0	1	0	0	0
Wada	.322	27	1	0	0	0	0	0	1	0	0
Tamura	.304	31	1	0	0	0	0	0	0	1	0
Aoki	.344	3	1	0	0	0	0	0	0	1	0
Fukudome	.328	28	1	0	0	0	0	0	0	1	1
Ichiro	.364	18	1	0	0	0	0	0	0	1	1
Kinjo	.324	12	1	0	0	0	0	0	0	0	1

図 10　WBC 日本代表選手 14 人が守備可能なポジション
Fig. 10　The fielding position table for 14 Japanese players.

$$c_i \text{ (ord/sec)} = \begin{cases} 264.25 & \text{(F32)} \\ 115.48 & \text{(DIS)} \\ 192.67 & \text{(Tau)} \\ 193.81 & \text{(Xenia)} \end{cases} \tag{12}$$

クラスタ i の計算ノード数 n_i は

$$n_i = \begin{cases} 100 & \text{(F32)} \\ 40 & \text{(DIS)} \\ 100 & \text{(Tau)} \\ 40 & \text{(Xenia)} \end{cases} \tag{13}$$

となる．全打順数 T は 134,991,360 となり，今回の検証に用いた計算ノードの中で最も高い性能を持つ F32 クラスタの計算ノード 1 台でこれらの打順の期待得点を計算した場合，理論的には 510,847（sec）を要し，これはおよそ 5.9 日に相当する．なお式 (12) の値は計算パラメータの共通利用による高速化手法を適用した値のため，この手法を適用せず，かつ守備位置充足可能性判定も行わない場合の単一計算ノードによる理論的な計算時間は表 1，表 2 から ${}_{14}P_9/264.25 \times 8.81 \times 0.95 \approx 2.3 \times 10^7$（sec）$\approx$ 265 日となる．式 (11)，式 (12)，式 (13) の値を式 (9) に代入した結果およびグリッド環境上での実測値を**図 11** に示す．理論値では分割数 24,853 の場合に実行時間が 2,394.5（sec）と見積もられ，実測値では分割数 15,000 の場合に実行時間が 2,522（sec）で最短となった．理論値ではジョブが各計算ノードに理想的に割り当てられている状況を仮定しているため実測値はこの値を上回っているが，分割数と実行時間との関係はよく示されている．図 11 より分割数 50,000 の場合に 5,131（sec）を要し，最短の場合に比べて 2 倍以上の性能低下が生じていること

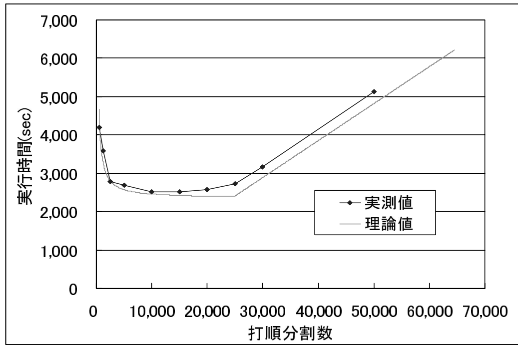

図 11 打順分割数と処理時間の関係
Fig. 11 The relationship between execution time and job partition size.

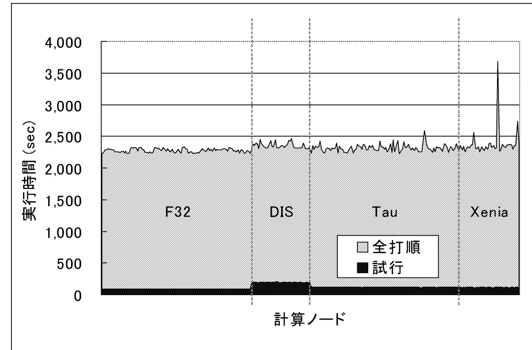

図 12 試行計算による方式の各計算ノード計算時間
Fig. 12 The execution time on computing nodes with static load balancing.

表 3 WBC 日本代表の最適打順
Table 3 The optimal batting order of the WBC Japan team.

打順	打者	打率	本塁打
1	Fukudome	.328	28
2	Iwamura	.319	30
3	Matsunaka	.315	46
4	Wada	.322	27
5	Ichiro	.364	18
6	Tamura	.304	31
7	Satozaki	.303	10
8	Imae	.310	8
9	Nishioka	.268	4
期待得点：6.995			

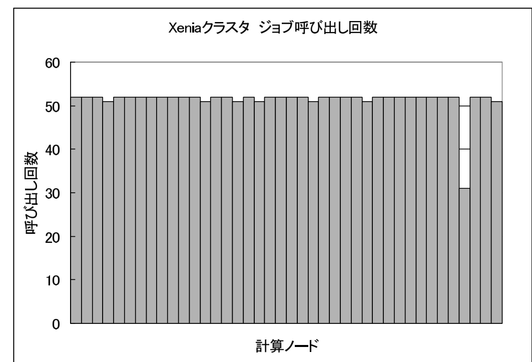

図 13 Xenia クラスタにおけるジョブ呼び出し回数
Fig. 13 The number of calling jobs on Xenia cluster.

表 4 負荷分散方式の性能比較
Table 4 Performance comparison between two load balancing algorithms.

負荷分散方式	実行時間（sec）
適応的割当て	2,522
試行計算	3,925

が分かる．よって打順分割数が実行時間に与える影響は大きく，モデル化によるその最適値の見積りは計算の高速化に有効な手法といえる．

期待得点計算の結果得られた最適打順を**表 3** に示す．

5.4 負荷分散方式の性能比較

適応的ジョブ割当てによる負荷分散方式と 4.2 節で示した試行計算による性能見積りに基づく負荷分散方式の性能比較を行った結果を**表 4** に示す．さらに，試行計算に基づく方式における各計算ノードの試行計算と全計算対象打順の計算に要した時間を**図 12** に示す．ただし，横軸は各計算ノードを表している．

図 12 より，Xenia クラスタ内に他の計算ノードと比べて処理能力の低い計算ノードが 1 台存在していることが示されている．クラスタ内の計算ノードの性能が均質ではなく，また時間経過とともに性能の変化が起こりうるグリッド環境において試行計算に基づく方式を用いるとこのような計算ノードに割り当てられた計算が終了するまでその他の計算ノードはアイドル状態となり，非効率的である．図 12 では Xenia クラスタの 1 台の計算ノードを除きほぼすべての計算ノードで試行計算と全打順に対する計算の合計が 2,500 秒を下回っているが，これらの計算ノードが 1 台の計算ノードの計算終了まで 1,500 秒程度の同期待ちを行うことになる．打順分割数を 15,000 として適応的ジョブ割当てによる負荷分散方式により期待得点の計算を行った際の Xenia クラスタにおけるジョブの呼び出し回数を**図 13** に示す．適応的にジョブを割り当てているため，図 13 では処理能力の低い 1 台の計算ノードに対して少ない数のジョブが割り当てられていることが示されている．このようにグリッド環境では適応的ジョブ割当てによる負荷分散方式を採用することで計算資源の有効利用が図られ，全体の計算時間が短縮されることが確認された．

5.5 性能評価結果のまとめ

各高速化手法による効果を以下に示す．打者 10 人

の集合に対して実験を行ったところ，計算パラメータの共通利用により 8.81 倍，守備位置充足可能性判定により (全打順数)/(守備位置充足打順数)×0.95 倍の速度向上が得られた．よって理論的には WBC 出場 14 選手の集合に対し ${}_{14}P_9\,/\,134{,}991{,}360 \times 0.95 = 5.11$ 倍の速度向上が得られることになる．4 拠点 280 台の計算ノードを使用して 2 種類の負荷分散方式の実装を行ったところ，上記の高速化を施したうえでの最も高速なクラスタの計算ノード 1 台による理論的な実行時間 510,847（sec）に対し，表 4 より適応的ジョブ割当てによる手法では 202 倍（2,522（sec）），試行計算による手法では 130 倍（3,925（sec））の速度向上がそれぞれ得られた．

6. まとめ

本稿では，野球チームの打順最適化問題計算の高速化手法を提案し，提案手法をグリッド上に実装して評価した結果について報告した．高速化手法として，計算パラメータを共通利用することにより 8.81 倍の速度向上が図られ，守備位置の充足可能性判定処理をプログラム中でそのオーバヘッドを抑えつつ実行することで計算対象打順を人手を介さずに選別することが可能となり，選別による削減数に応じて実行時間が短縮された．

さらにグリッドのような不均質な計算機環境における負荷分散方式として適応的ジョブ割当てによる負荷分散と試行計算による性能見積りに基づく負荷分散を比較した．試行計算に基づく方式では多数の計算ノードが処理の遅い計算ノードの計算終了を待つという状態が発生しうるため，その場合は処理の遅い計算ノードの実行時間が全体の実行時間となり，多数の計算ノードが有効利用されていなかった．適応的ジョブ割当てによる方式では全打順の期待得点計算を一定数のジョブに分割することで各計算ノードの性能に応じてジョブが割り当てられ，計算に参加しているすべての計算ノードの有効利用が可能となった．

適応的ジョブ割当てによる方式においてその実行時間に影響を与える打順の分割数，すなわち期待得点計算のジョブ数の最適値をモデル化により見積もった．その結果分割数とモデル化による理論的な実行時間との関係と実際の実行時間との関係はよく一致し，分割数の選択における 1 つの指針を与えた．

ジョブの多重割当てでは，多重に割り当てられたジョブの 1 つが終了した場合，他の計算ノードで重複して実行されているジョブの実行を中断することにより，さらに性能を向上させることが期待できる．これら耐故障性に対する詳細な評価については，今後の課題である．

14 人の打者の集合からなる 134,991,360 通りの打順について単一計算ノードで期待得点を計算すると理論的には最大で 265 日を要するが，4 拠点の PC クラスタを用いて上記の高速化手法を施したうえで計算を行った結果，2,522 秒で最適な打順を決定することが可能となった．

謝辞 性能評価にご協力いただいた産業技術総合研究所グリッド研究センター，東京工業大学小野研究室，東京大学田浦研究室，同志社大学知的システムデザイン研究室に感謝いたします．

参考文献

1) 合田憲人，大澤 清，大角知孝，笠井武史，小野功，實本英之，松岡 聡，斎藤秀雄，遠藤敏夫，横山大作，田浦健次朗，近山 隆，田中良夫，下坂久司，梶原広輝，廣安知之，藤澤克樹：グリッドチャレンジテストベッドの構築と運用—グリチャレテストベッドの作り方，情報処理学会研究報告，Vol.2006, No.87, pp.49–54 (2006).

2) Bukiet, B. and Harold, E.: A Markov Chain Approach to Baseball, *Operations Research*, Vol.45, No.1, pp.14–23 (1997).

3) Condor Team: Condor Project Homepage. www.cs.wisc.edu/condor/

4) Dobber, M., Koole, G. and van der Mei, R.: Dynamic Load Balancing Experiments in a Grid, *Proc. CCGrid 2005*, pp.1063–1070 (2005).

5) Fujimoto, N. and Hagihara, K.: Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid, *Proc. 2003 International Conference on Parallel Processing* (*ICPP'03*), IEEE, pp.391–398 (2003).

6) Golomb, S.W. and Baumert, L.D.: Backtrack Programming, *J. ACM*, Vol.12, No.4, pp.516–524 (1965).

7) 廣津信義，宮地 力：野球チームのラインナップ選定のための数理的一手法—日本代表チームの選定を例として，オペレーションズ・リサーチ，Vol.49, No.6, pp.380–389 (2004).

8) Horst, R., Pardalos, P.M. and Thoai, N.V.: *Introduction to Global Optimization*, Kluwer Academic Pub. (2000).

9) 久保田和人，仲瀬明彦：耐故障/耐高負荷を考慮した並列分枝限定法と基本性能の評価，情報処理学会論文誌：コンピューティングシステム，Vol.45, No.SIG11(ACS 7), pp.171–181 (2004).

10) Osawa, K. and Aida, K.: Speed-up Techniques for Computation of Markov Chain Model to Find an Optimal Batting Order, *Proc. 8th In-*

ternational Conference on High-Performance Computing in Asia-Pacific Region (*HPC Asia 2005*), pp.315–322 (2005).

11) Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S.: Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing, *Journal of Grid Computing*, Vol.1, No.1, pp.41–51 (2003).

12) Whaley, R.C., Petitet, A. and Dongarra, J.J.: Automated Empirical Optimization of Software and the ATLAS Project, *Parallel Computing*, Vol.27, No.1–2, pp.3–35 (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448 (2000). www.netlib.org/lapack/lawns/lawn147.ps

(平成 18 年 4 月 28 日受付)
(平成 18 年 12 月 7 日採録)

大澤 清（学生会員）

1998 年東京工業大学工学部情報工学科卒業．2001 年東京大学大学院理学系研究科情報科学専攻修士課程修了．現在，東京工業大学大学院総合理工学研究科博士課程在学中．大規模並列数値計算に関する研究に従事．日本オペレーションズ・リサーチ学会会員．

合田 憲人（正会員）

昭和 42 年生．平成 2 年早稲田大学理工学部電気工学科卒業．平成 4 年同大学大学院修士課程修了．平成 8 年同大学院博士課程退学．平成 4 年早稲田大学情報科学研究教育センター助手，平成 8 年同特別研究員．平成 9 年東京工業大学大学院情報理工学研究科助手，平成 11 年同大学院総合理工学研究科専任講師，平成 15 年同助教授，現在に至る．博士（工学）．並列・分散計算方式，グリッドコンピューティング，スケジューリング技術の研究に従事．電子情報通信学会，電気学会，ACM，IEEE-CS 各会員．