JAERI-Tech
95-051

高速モンテカルロ装置上における MORSEコードのベクトル並列処理

1995年11月

長谷川幸弘*・樋口健二

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。
入手の問合わせは，日本原子力研究所技術情報部情報資料課(〒319-11 茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター（〒319-11 茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.
Inquiries about availability of the reports should be addressed to Information Division, Department of Technical Information, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken 319-11, Japan.

編集兼発行　日本原子力研究所
印　　刷　㈱高野高速印刷

高速モンテカルロ装置上におけるＭＯＲＳＥコードのベクトル並列処理

日本原子力研究所計算科学技術推進センター
長谷川幸弘＊・樋口　健二

（1995年10月18日受理）

多群粒子輸送モンテカルロ・コードＭＯＲＳＥを高速モンテカルロ装置上で高速処理した手法及び結果について述べる。高速モンテカルロ装置Ｍｏｎｔｅ－４は，従来のベクトル計算機において困難であった粒子輸送モンテカルロ・コードの高速処理を実現するために開発された専用計算機である。Ｍｏｎｔｅ－４は，モンテカルロ・パイプラインと呼ばれる特殊なハードウェアを搭載した４台のベクトル演算装置を持つ。ＭＯＲＳＥコードのベクトル化手法，並列化手法，及び高速モンテカルロ装置上における性能評価結果について述べる。

日本原子力研究所：〒113　東京都文京区本駒込２－28－８
＊　㈶高度情報科学技術研究機構

The Vector and Parallel Processing of MORSE Code on Monte Carlo Machine

Yukihiro HASEGAWA* and Kenji HIGUCHI

Center for Promotion of Computational Science and Engineering
Japan Atomic Energy Research Institute
2-28-8 Honkomagome, Bunkyo-ku, Tokyo

(Received October 18, 1995)

Multi-group Monte Carlo Code for particle transport, MORSE is modified for high performance computing on Monte Carlo Machine Monte-4. The method and the results are described. Monte-4 was specially developed to realize high performance computing of Monte Carlo codes for particle transport, which have been difficult to obtain high performance in vector processing on conventional vector processors. Monte-4 has four vector processor units with the special hardware called Monte Carlo pipelines. The vectorization and parallelization of MORSE code and the performance evaluation on Monte-4 are described.

Keywords : Monte Carlo, MORSE, Vectorization, Parallelization, Vector Processing, Parallel Processing, Particle Transport Problem

* Research Organization for information Science & Technology

目　　次

Contents

1．はじめに

中性子や光子の輸送問題のための数値解法には，Sn法，直接積分法，有限要素法等などの決定論的手法と，モンテカルロ法のような非決定論的手法（＝統計的手法）がある。モンテカルロ法は決定論的手法と比較し，複雑な形状において厳密な解を得ることのできる反面，精度のよい計算結果を得るために多大の計算時間を必要とする。即ち，モンテカルロ計算の精度は，サンプリングする粒子の数N の平方根に反比例するため，例えば精度を１／１０にしようとすれば１００倍の計算時間を必要とする。

小さな粒子数により，良い精度を得るために，多くの粒子輸送モンテカルロ・コード（以後，モンテカルロ・コード）では種々の分散低減法が使用されている。一方，コンピュータ・アーキテクチャの面からのアプローチとして，ベクトル計算機によるモンテカルロ・コードの高速化の研究も行われてきた。しかし，日本原子力研究所（以下，原研）情報システムセンター（平成７年度より計算科学技術推進センターと改称，本稿では情報システムセンター）において，KENO-IV[1]，MCNP[2]，MORSE[3]，VIM[4] 等の汎用モンテカルロ・コードをベクトル化し，従来のベクトル計算機上において処理した経験によれば，従来のベクトル計算機上の汎用モンテカルロ・コードの高速化は困難である。その理由として，

ｉ）GOTO文やIF文が多用されているためにベクトル化率が低い，

ii）ベクトル長が短く，間接番地データを用いてベクトル計算を行うため加速率が低い，

という点が挙げられる。直観的には，従来のベクトル計算機は，局所的に計算コストが集中し，かつ大量のデータに対し数値演算を行うような連続体モデルの計算に対しては高い性能を得ることができるが，モンテカルロ・コードのように，計算コストがプログラムの広い範囲にわたって分散し，かつ，小量のデータに対し，論理演算及び数値演算を行うような粒子モデル計算に対しては高い性能を得ることができない。

情報システムセンターでは，従来のベクトル計算機上で極めて困難であったモンテカルロ・コードの高速化を実現するために，高速モンテカルロ装置（以下，Monte-4 ）を開発した。[5-8] Monte-4 は以下の特徴を持つ専用計算機である。

ｉ）４つのベクトル・プロセッサを持つ共有メモリ型ベクトル・並列コンピュータである。

ii）リストベクトル処理を高速化するために，ロード/ ストア・パイプラインによるデータ転送機能を強化している。

iii）モンテカルロ計算で多用され，ベクトル処理不可能な処理，ベクトル処理しても性能の得られない処理を高速化するためのモンテカルロ・パイプラインを装備している。

Monte-4 は「既存のコードのスカラー処理に対して１０倍」の実効性能を設計基準として開発された。実際，既にKENO-IV ，MCNPコードが並列化され，両コードについて約１０倍の性能を得ている。[8]

今回は多群モンテカルロコードMORSE[9] の並列化を行った。既に情報システムセンターにおいてVP100 用にベクトル化済みのコード[3] をベースに，Monte-4 上でのチューニング及び並列化を行った。本報告書では，多群モンテカルロコードMORSE の並列化，高速化チューニング及

び実効性能について述べる。

2. MORSEコード

2.1 コードの概要

MORSE コードはOak Ridge National Laboratory で開発された粒子輸送モンテカルロ・コードである。MORSE コードの特徴は次のようなものが挙げられる。

ⅰ）中性子及び光子を取り扱うことができる。

ⅱ）遮蔽及び臨界問題を取り扱うことができる。

ⅲ）エネルギーをいくつかの群に分けて近似する多群法を採用している。

ⅳ）CG(Combinatorial Geometory) 法と呼ばれる体系の記述法を採用している。即ち，ユーザは直方体，円柱などの基本幾何形状（Body）を組み合せることによってZONEと呼ばれる3次元領域を定義し，複雑な体系を記述することができる。

また，モンテカルロ・コードにおいては粒子の飛程と2次曲面の論理演算によって表現された3次元領域との交差判定に多くの処理時間が費やされるため，MORSE コードでは高速化のために次のような工夫がされている。

ⅰ）体系内をランダムウォークする粒子が，現在属しているZONEの境界を越える場合，次に進入するZONEの検索処理が行われる。このZONE検索処理は，ZONEの数が多いほど多大の処理時間を必要とする。ところが，粒子が現在属しているZONEと次に入るZONEは必ず隣接しているという点を考慮すると，ZONE同士の隣接関係を記述したデータ・テーブルを用いて，検索対象となるZONEを限定することができる。この隣接ZONEテーブルを用いたZONE検索処理を取り入れることによって処理時間を短縮している。

ⅱ）ZONEはいくつかのBodyの論理演算（論理和，論理積）によって表現されているため，同じBodyを用いて表現されているZONEが存在する。そのため，異なるZONEであっても，同じ飛程と同じBodyとの交差判定が繰り返して行われることがある。そこで，飛程を識別するインデックスと各Bodyの交差判定の結果を覚えておき，インデックス及びBody番号が同じ場合は既に計算済みの結果を用いることで，無駄な再計算を省略している。

今回の作業の対象となったコードは，各種人体模型が使用可能な実効線量当量評価システムFANTOME のモンテカルロ法による粒子輸送計算部分を構成しているMORSE コードの拡張版（以後，FANTOME 版）である。このコードはMORSE-CGコードに対して次のような拡張がなされたものである。

ⅰ）複雑な人体模型を記述するため，取り扱い幾何形状を追加している。

ⅱ）各臓器，組織のフルエンス及び標準偏差を計算するように統計処理ルーチンが改定されている。

MORSE コードは，最も一般的なMORSE-CGや，計算精度の向上のためにルジャンドル展開法による断面積の他に二重微分型断面積を用いているMORSE-DDなど，いくつかのバージョンが存在する。並列化のベースとしたコードはMORSE-DDをベクトル化したものである。FANTOME 版とMO

び実効性能について述べる。

2. MORSEコード

2.1 コードの概要

MORSE コードはOak Ridge National Laboratory で開発された粒子輸送モンテカルロ・コードである。MORSE コードの特徴は次のようなものが挙げられる。

ⅰ）中性子及び光子を取り扱うことができる。

ⅱ）遮蔽及び臨界問題を取り扱うことができる。

ⅲ）エネルギーをいくつかの群に分けて近似する多群法を採用している。

ⅳ）CG(Combinatorial Geometory) 法と呼ばれる体系の記述法を採用している。即ち，ユーザは直方体，円柱などの基本幾何形状（Body）を組み合せることによってZONEと呼ばれる3次元領域を定義し，複雑な体系を記述することができる。

また，モンテカルロ・コードにおいては粒子の飛程と2次曲面の論理演算によって表現された3次元領域との交差判定に多くの処理時間が費やされるため，MORSE コードでは高速化のために次のような工夫がされている。

ⅰ）体系内をランダムウォークする粒子が，現在属しているZONEの境界を越える場合，次に進入するZONEの検索処理が行われる。このZONE検索処理は，ZONEの数が多いほど多大の処理時間を必要とする。ところが，粒子が現在属しているZONEと次に入るZONEは必ず隣接しているという点を考慮すると，ZONE同士の隣接関係を記述したデータ・テーブルを用いて，検索対象となるZONEを限定することができる。この隣接ZONEテーブルを用いたZONE検索処理を取り入れることによって処理時間を短縮している。

ⅱ）ZONEはいくつかのBodyの論理演算（論理和，論理積）によって表現されているため，同じBodyを用いて表現されているZONEが存在する。そのため，異なるZONEであっても，同じ飛程と同じBodyとの交差判定が繰り返して行われることがある。そこで，飛程を識別するインデックスと各Bodyの交差判定の結果を覚えておき，インデックス及びBody番号が同じ場合は既に計算済みの結果を用いることで，無駄な再計算を省略している。

今回の作業の対象となったコードは，各種人体模型が使用可能な実効線量当量評価システムFANTOME のモンテカルロ法による粒子輸送計算部分を構成しているMORSE コードの拡張版（以後，FANTOME 版）である。このコードはMORSE-CGコードに対して次のような拡張がなされたものである。

ⅰ）複雑な人体模型を記述するため，取り扱い幾何形状を追加している。

ⅱ）各臓器，組織のフルエンス及び標準偏差を計算するように統計処理ルーチンが改定されている。

MORSE コードは，最も一般的なMORSE-CGや，計算精度の向上のためにルジャンドル展開法による断面積の他に二重微分型断面積を用いているMORSE-DDなど，いくつかのバージョンが存在する。並列化のベースとしたコードはMORSE-DDをベクトル化したものである。FANTOME 版とMO

RSE-DDは断面積の処理や統計処理の部分は大きく異なっているが，計算コストの集中するランダムウォーク処理部分に関してはほとんど同じである。したがって，ベクトル版をFANTOME 版用に改良する（断面積の処理，統計処理ルーチンをFANTOME 版と入れ換え，不要なルーチンは削除する）作業は比較的容易であった。以後，MORSE コードのオリジナル・スカラ版，ベクトル版，並列版はFANTOME 版を指すものとして述べる。

2.2 入力データ

テストラン及び性能評価には次の３つの問題を使用した。問題２及び問題３はMORSE-DD用の入力データであり，実効性能評価をするときに使用した。また，高速化のためのチューニングは問題１を使用して行った。

(1)問題１

Fig.2.1 のような人体模型（男性及び女性の性器を併せ持つ成人，Body数１４７，領域数６３）に真正面から光子平行ビームが入射したときの各臓器，組織のフルエンス及び標準偏差を計算する。エネルギー９０KeV(１４群）の光子を１，０００，０００個追跡する。

(2)問題２

Fig.2.2 のようなＦＮＳ（Fusion Neutron Source ）回転ターゲットにおける中性子源の特性解析を行う。ＦＮＳ回転ターゲットの付近に１２個のpoint detectorを置き，そこでの中性子束を計算する。中性子を５００，０００個追跡する。

(3)問題３

Fig.2.3 のような酸化リチウムの６０cm平板体系における中性子の分布と中性子による反応率を計算する。中性子を５００，０００個追跡する。

2.3 コードの挙動解析

高速モンテカルロ装置上のコード解析ツールANALYZER-P[10] を用いて，オリジナル・コードの各問題に対する動的挙動を調査した結果をTable.2.1 (a) ～(c) に示す。すべての問題においてサブルーチンGG,G1 に計算コストが集中していることがわかる。ただし，問題１及び問題３は粒子追跡処理部分の" NXTCOL-GOMST-G1 -GG " ，問題２は統計処理部分の"RELCOL-EUCLID-G1-GG" の経路で呼び出されるGG,G1 のコストが高くなっている。計算コストの高い主要なサブルーチン及び計算内容は次の通りである。

NXTCOL: 粒子の次の衝突点を求める。
GOMST : 粒子の飛程に沿った次の境界までの距離を求める。
G1 : 粒子の飛程が交差するZone番号及びBody番号を求める。
GG : 粒子の飛程がBodyと交差する座標を求める。
RELCOL: 中性子束を評価する。
EUCLID: 粒子の現在位置と各検出器間の距離を求める。

Table.2.1(a) Summary list of behavior analysis of original MORSE code in scalar processing (probrem.1)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GG	SUB		56096627	249.754(47.2)	226.090(42.8)	54.1	12.3	0.00	0.0	0.000(0)	56.533(11)
G1	SUB		2757610	321.009(60.7)	114.203(21.6)	13.8	0.8	0.00	0.0	0.000(0)	45.824(9)
NXTCOL	SUB		1628208	405.730(76.7)	29.222(5.5)	11.3	0.5	0.00	0.0	0.000(0)	23.224(4)
LOOKZ	SUB		1000000	61.337(11.6)	18.388(3.5)	22.3	1.5	0.00	0.0	0.000(0)	4.513(1)
GOMST	SUB		2757610	337.869(63.9)	16.860(3.2)	29.2	1.3	0.00	0.0	0.000(0)	8.791(2)
MORSE	SUB		1	528.630(100.0)	15.650(3.0)	8.5	0.1	0.00	0.0	0.000(0)	11.943(2)
CUBIC	SUB		1324695	13.057(2.5)	13.057(2.5)	42.1	14.5	0.00	0.0	0.000(0)	2.717(1)
MSOUR	SUB		100	80.668(15.3)	10.928(2.1)	12.2	0.0	0.00	0.0	0.000(0)	7.511(1)
BANKR	SUB		3757812	16.973(3.2)	10.688(2.0)	14.4	0.0	0.00	0.0	0.000(0)	3.633(1)
QRTIC	SUB		1324695	23.664(4.5)	10.607(2.0)	28.0	9.1	0.00	0.0	0.000(0)	3.185(1)
NSIGTA	SUB		2757610	13.471(2.5)	9.698(1.8)	17.1	0.3	0.00	0.0	0.000(0)	5.906(1)
FLTRNF	FUNC		10967589	9.655(1.8)	9.655(1.8)	63.4	15.9	0.00	0.0	0.000(0)	0.331(0)
COLISN	SUB		648613	11.401(2.2)	8.048(1.5)	26.2	5.2	0.00	0.0	0.000(0)	3.866(1)
TRACE	SUB		2757610	6.135(1.2)	6.135(1.2)	37.4	8.2	0.00	0.0	0.000(0)	2.569(0)
GTMED	SUB		4054836	5.129(1.0)	5.129(1.0)	27.5	0.0	0.00	0.0	0.000(0)	0.651(0)
GETNT	SUB		3000001	5.044(1.0)	5.044(1.0)	55.6	0.0	0.00	0.0	0.000(0)	0.577(0)
STORNT	ENT		1000000								
SETNT	ENT		1								
GETETA	SUB		1628208	13.956(2.6)	4.518(0.9)	34.2	6.8	0.00	0.0	0.000(0)	0.779(0)
TESTW	SUB		1648613	3.983(0.8)	3.970(0.8)	29.1	0.8	0.00	0.0	0.000(0)	1.332(0)
EXPRNF	FUNC		1628208	9.438(1.8)	2.993(0.6)	34.9	2.3	0.00	0.0	0.000(0)	0.553(0)
AZIRN	SUB		1648613	3.765(0.7)	2.598(0.5)	59.5	13.1	0.00	0.0	0.000(0)	0.010(0)
SOURC2	SUB		1000000	5.292(1.0)	2.406(0.5)	37.9	10.8	0.00	0.0	0.000(0)	0.064(0)
READSG	SUB		72	1.128(0.2)	1.128(0.2)	98.6	0.0	0.00	0.0	0.000(0)	0.008(0)
OUTPT	SUB		201	2.332(0.4)	0.869(0.2)	51.8	19.6	0.00	0.0	0.000(0)	0.050(0)
JOMIN	SUB		1	0.214(0.0)	0.135(0.0)	38.2	0.3	0.02	46.5	0.000(0)	0.019(0)
NRUN	SUB		1	0.134(0.0)	0.134(0.0)	32.7	0.2	0.11	18.7	0.000(0)	0.025(0)
FIND	SUB		791	0.190(0.0)	0.119(0.0)	17.2	3.0	0.00	0.0	0.000(0)	0.038(0)
JNPUT	SUB		1	1.426(0.3)	0.084(0.0)	61.6	2.4	0.00	0.0	0.000(0)	0.013(0)
GENI	SUB		1	0.074(0.0)	0.074(0.0)	45.4	0.2	0.00	0.0	0.000(0)	0.007(0)
Q	FUNC		93751	0.073(0.0)	0.073(0.0)	81.3	9.1	0.00	0.0	0.000(0)	0.001(0)
OUTPT2	SUB		1	0.057(0.0)	0.057(0.0)	34.8	0.1	0.29	15.7	0.000(0)	0.010(0)
MAIN_	MAIN		1	528.658(100.0)	0.015(0.0)	99.9	0.0	0.00	0.0	0.000(0)	0.000(0)
INPUT2	SUB		1	1.443(0.3)	0.015(0.0)	88.3	0.0	0.00	0.0	0.000(0)	0.000(0)
NBATCH	SUB		100	0.014(0.0)	0.014(0.0)	55.6	15.0	0.00	0.0	0.000(0)	0.008(0)
DTLIST	SUB		1	0.012(0.0)	0.012(0.0)	34.4	0.0	0.26	46.5	0.000(0)	0.003(0)
ANGLES	SUB		400	0.203(0.0)	0.008(0.0)	20.6	1.3	0.00	0.0	0.000(0)	0.004(0)
INPUT1	SUB		2	0.222(0.0)	0.008(0.0)	47.6	0.1	0.00	0.0	0.000(0)	0.001(0)
STORE	SUB		72	0.007(0.0)	0.007(0.0)	85.6	0.0	0.00	0.0	0.000(0)	0.001(0)
GTVLIN	SUB		1	0.005(0.0)	0.005(0.0)	40.3	0.5	0.41	15.7	0.000(0)	0.001(0)
LEGEND	SUB		400	0.004(0.0)	0.004(0.0)	60.9	19.5	0.00	0.0	0.000(0)	0.001(0)

Table.2.1(b) Summary list of behavior analysis of behavior analysis of original MORSE code in scalar processing (probrem.2)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GG	SUB		34649741	113.525(21.9)	113.525(21.9)	41.0	7.0	0.00	0.0	0.000(0)	31.864(6)
G1	SUB		4758792	208.430(40.1)	95.999(18.5)	17.2	1.1	0.00	0.0	0.000(0)	46.557(9)
NSIGTA	SUB		47896824	125.515(24.2)	85.973(16.5)	31.8	0.6	0.00	0.0	0.000(0)	11.292(2)
RELCOL	SUB		12878	298.516(57.5)	72.830(14.0)	35.5	5.5	0.00	0.0	0.000(0)	41.851(8)
FLUXST	SUB		10516683	40.884(7.9)	40.884(7.9)	35.8	1.3	0.00	0.0	0.000(0)	8.583(2)
GTMED	SUB		48077116	39.668(7.6)	39.668(7.6)	41.0	0.0	0.00	0.0	0.000(0)	0.638(0)
PTHETA	SUB		154536	24.659(4.7)	24.562(4.7)	59.4	8.5	0.00	0.0	0.000(0)	1.959(0)
EUCLID	SUB		1651866	220.007(42.4)	19.128(3.7)	29.6	1.5	0.00	0.0	0.000(0)	10.057(2)
SDATA	SUB		50000	188.096(36.2)	12.440(2.4)	32.4	4.7	0.00	0.0	0.000(0)	6.153(1)
NXTCOL	SUB		62873	23.727(4.6)	2.460(0.5)	14.6	0.5	0.00	0.0	0.000(0)	1.934(0)
FNSSOC	SUB		50001	2.512(0.5)	2.158(0.4)	45.2	8.5	0.00	0.0	0.000(0)	0.488(0)
GOMST	SUB		358648	18.984(3.7)	1.472(0.3)	40.1	1.9	0.00	0.0	0.000(0)	0.609(0)
BANKR	SUB		408850	488.036(93.9)	1.246(0.2)	15.1	0.0	0.00	0.0	0.000(0)	0.610(0)
MACRO4	SUB		6	1.162(0.2)	1.162(0.2)	52.0	2.2	0.23	53.6	0.000(0)	0.086(0)
ROTAT2	SUB		600000	0.833(0.2)	0.833(0.2)	41.6	10.8	0.00	0.0	0.000(0)	0.000(0)
FLTRNF	FUNC		952486	0.799(0.2)	0.799(0.2)	66.5	16.7	0.00	0.0	0.000(0)	0.048(0)
LOOKZ	SUB		50000	1.768(0.3)	0.674(0.1)	19.8	0.8	0.00	0.0	0.000(0)	0.333(0)
MSOUR	SUB		100	193.704(37.3)	0.645(0.1)	12.2	0.2	0.00	0.0	0.000(0)	0.539(0)
MORSE	SUB		1	519.443(100.0)	0.468(0.1)	10.9	0.2	0.00	0.0	0.000(0)	0.337(0)
RESTOR	SUB		8	0.467(0.1)	0.331(0.1)	78.2	0.0	5.66	57.6	0.000(0)	0.035(0)
GETNT	SUB		150001	0.297(0.1)	0.297(0.1)	47.1	0.0	0.00	0.0	0.000(0)	0.076(0)
STORNT	ENT		50000								
SETNT	ENT		1								
SOURCE	SUB		50000	2.848(0.5)	0.281(0.1)	23.7	0.4	0.00	0.0	0.000(0)	0.142(0)
COLISN	SUB		12878	0.318(0.1)	0.244(0.0)	28.3	6.1	0.00	0.0	0.000(0)	0.125(0)
GETETA	SUB		62873	0.504(0.1)	0.220(0.0)	27.2	5.4	0.00	0.0	0.000(0)	0.084(0)
MACRO2	SUB		12	0.136(0.0)	0.136(0.0)	97.9	31.2	0.00	0.0	0.000(0)	0.023(0)
REARAG	SUB		864	0.136(0.0)	0.136(0.0)	84.7	5.0	1.66	63.8	0.000(0)	0.014(0)
EXPRNF	FUNC		62873	0.284(0.1)	0.116(0.0)	35.0	2.4	0.00	0.0	0.000(0)	0.013(0)
NRUN	SUB		1	0.114(0.0)	0.112(0.0)	36.3	0.3	0.04	23.2	0.000(0)	0.016(0)
OUTPT	SUB		201	0.191(0.0)	0.093(0.0)	45.0	9.2	0.00	0.0	0.000(0)	0.011(0)
APMATX	SUB		1	0.080(0.0)	0.080(0.0)	106.2	0.0	34.66	63.0	0.000(0)	0.010(0)
MACRO1	SUB		12	1.364(0.3)	0.063(0.0)	114.2	0.0	0.00	0.0	0.000(0)	0.002(0)
ROTAT1	SUB		100000	0.058(0.0)	0.058(0.0)	99.3	25.8	0.00	0.0	0.000(0)	0.000(0)
JNPUT	SUB		1	1.501(0.3)	0.048(0.0)	83.7	0.8	0.00	0.0	0.000(0)	0.001(0)
NBATCH	SUB		100	0.047(0.0)	0.047(0.0)	88.5	14.1	0.00	0.0	0.000(0)	0.003(0)
GENI	SUB		1	0.036(0.0)	0.036(0.0)	37.0	0.3	0.00	0.0	0.000(0)	0.006(0)
MAIN_	MAIN		1	519.478(100.0)	0.035(0.0)	100.0	0.0	0.00	0.0	0.000(0)	0.000(0)
JOMIN	SUB		1	0.066(0.0)	0.030(0.0)	36.5	0.3	0.05	46.5	0.000(0)	0.005(0)
SCORIN	SUB		1	0.029(0.0)	0.029(0.0)	35.1	0.2	0.00	0.0	0.000(0)	0.005(0)
AZIRN	SUB		12878	0.040(0.0)	0.026(0.0)	46.5	10.2	0.00	0.0	0.000(0)	0.002(0)
STBTCH	SUB		100	0.017(0.0)	0.017(0.0)	92.0	0.0	0.00	0.0	0.000(0)	0.000(0)

Table.2.1(c) Summary list of original MORSE code in scalar processing (probrem.3)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GG	SUB		5600256	31.134(22.4)	31.134(22.4)	37.9	9.4	0.00	0.0	0.000(0)	9.477(7)
G1	SUB		1586575	51.468(37.0)	20.336(14.6)	20.4	1.1	0.00	0.0	0.000(0)	10.262(7)
NXTCOL	SUB		768424	93.587(67.3)	12.842(9.2)	13.2	0.4	0.00	0.0	0.000(0)	9.641(7)
COLISN	SUB		699426	16.105(11.6)	11.789(8.5)	25.3	5.2	0.00	0.0	0.000(0)	6.381(5)
MORSE	SUB		1	139.003(100.0)	10.324(7.4)	8.2	0.1	0.00	0.0	0.000(0)	8.224(6)
BANKR	SUB		1636777	21.769(15.7)	7.731(5.6)	10.0	0.0	0.00	0.0	0.000(0)	5.089(4)
GOMST	SUB		1586575	58.613(42.2)	7.146(5.1)	38.8	1.8	0.00	0.0	0.000(0)	2.776(2)
FLUXST	SUB		1586575	6.430(4.6)	6.430(4.6)	31.5	0.7	0.00	0.0	0.000(0)	2.426(2)
FLTRNF	FUNC		5652443	4.838(3.5)	4.838(3.5)	65.2	16.4	0.00	0.0	0.000(0)	0.221(0)
NSIGTA	SUB		1517577	5.478(3.9)	4.532(3.3)	19.1	0.3	0.00	0.0	0.000(0)	2.488(2)
TLE5	SUB		699426	6.718(4.8)	3.616(2.6)	19.7	3.7	0.00	0.0	0.000(0)	2.349(2)
TLE7	SUB		887149	6.831(4.9)	3.502(2.5)	25.8	4.8	0.00	0.0	0.000(0)	1.799(1)
GTMED	SUB		2916429	2.739(2.0)	2.739(2.0)	36.0	0.0	0.00	0.0	0.000(0)	0.323(0)
TESTW	SUB		812212	2.838(2.0)	2.633(1.9)	23.0	0.7	0.00	0.0	0.000(0)	1.695(1)
GETETA	SUB		768424	6.173(4.4)	2.328(1.7)	31.4	6.3	0.00	0.0	0.000(0)	0.582(0)
AZIRN	SUB		699426	2.362(1.7)	1.518(1.1)	43.2	9.5	0.00	0.0	0.000(0)	0.154(0)
EXPRNF	FUNC		768424	3.845(2.8)	1.370(1.0)	36.0	2.4	0.00	0.0	0.000(0)	0.149(0)
SOURCE	SUB		50000	1.011(0.7)	0.824(0.6)	40.1	5.5	0.00	0.0	0.000(0)	0.179(0)
GETNT	SUB		275573	0.603(0.4)	0.603(0.4)	42.7	0.0	0.00	0.0	0.000(0)	0.197(0)
STORNT	ENT		112786								
SETNT	ENT		1								
MACRO4	SUB		2	0.493(0.4)	0.493(0.4)	51.9	2.0	0.25	62.7	0.000(0)	0.038(0)
RESTOR	SUB		8	0.608(0.4)	0.471(0.3)	73.9	0.0	4.36	57.0	0.000(0)	0.060(0)
OUTPT2	SUB		1	0.278(0.2)	0.278(0.2)	35.9	0.3	0.26	15.7	0.000(0)	0.041(0)
MSOUR	SUB		100	1.516(1.1)	0.270(0.2)	29.4	0.4	0.00	0.0	0.000(0)	0.135(0)
NBATCH	SUB		100	0.259(0.2)	0.259(0.2)	88.1	13.8	0.00	0.0	0.000(0)	0.017(0)
MACRO2	SUB		9	0.138(0.1)	0.138(0.1)	96.8	30.8	0.00	0.0	0.000(0)	0.026(0)
REARAG	SUB		1000	0.136(0.1)	0.136(0.1)	85.9	4.8	2.18	64.0	0.000(0)	0.014(0)
NRUN	SUB		1	0.139(0.1)	0.134(0.1)	41.6	0.1	0.02	43.8	0.000(0)	0.008(0)
STBTCH	SUB		100	0.091(0.1)	0.091(0.1)	94.5	0.0	0.00	0.0	0.000(0)	0.000(0)
OUTPT	SUB		201	0.439(0.3)	0.091(0.1)	45.8	9.5	0.00	0.0	0.000(0)	0.009(0)
LOOKZ	SUB		50000	0.064(0.0)	0.062(0.0)	45.1	2.4	0.00	0.0	0.000(0)	0.002(0)
GENI	SUB		1	0.051(0.0)	0.051(0.0)	36.7	0.1	0.00	0.0	0.000(0)	0.008(0)
JOMIN	SUB		1	0.099(0.1)	0.044(0.0)	36.1	0.1	0.04	46.5	0.000(0)	0.007(0)
INPUT1	SUB		1	0.143(0.1)	0.038(0.0)	48.8	0.2	0.06	46.5	0.000(0)	0.004(0)
JNPUT	SUB		1	0.737(0.5)	0.036(0.0)	84.5	0.3	0.00	0.0	0.000(0)	0.001(0)
MAIN_	MAIN		1	139.039(100.0)	0.035(0.0)	100.0	0.0	0.00	0.0	0.000(0)	0.000(0)
MACRO1	SUB		9	0.663(0.5)	0.030(0.0)	107.6	0.0	0.00	0.0	0.000(0)	0.002(0)
SCORIN	SUB		1	0.026(0.0)	0.026(0.0)	34.0	0.2	0.00	0.0	0.000(0)	0.005(0)
APMATX	SUB		1	0.026(0.0)	0.026(0.0)	116.4	0.0	44.00	62.4	0.000(0)	0.004(0)
INPUT2	SUB		1	1.403(1.0)	0.023(0.0)	88.4	0.0	0.00	0.0	0.000(0)	0.000(0)
RNDOUT	SUB		111	0.016(0.0)	0.016(0.0)	39.5	0.5	0.00	0.0	0.000(0)	0.003(0)

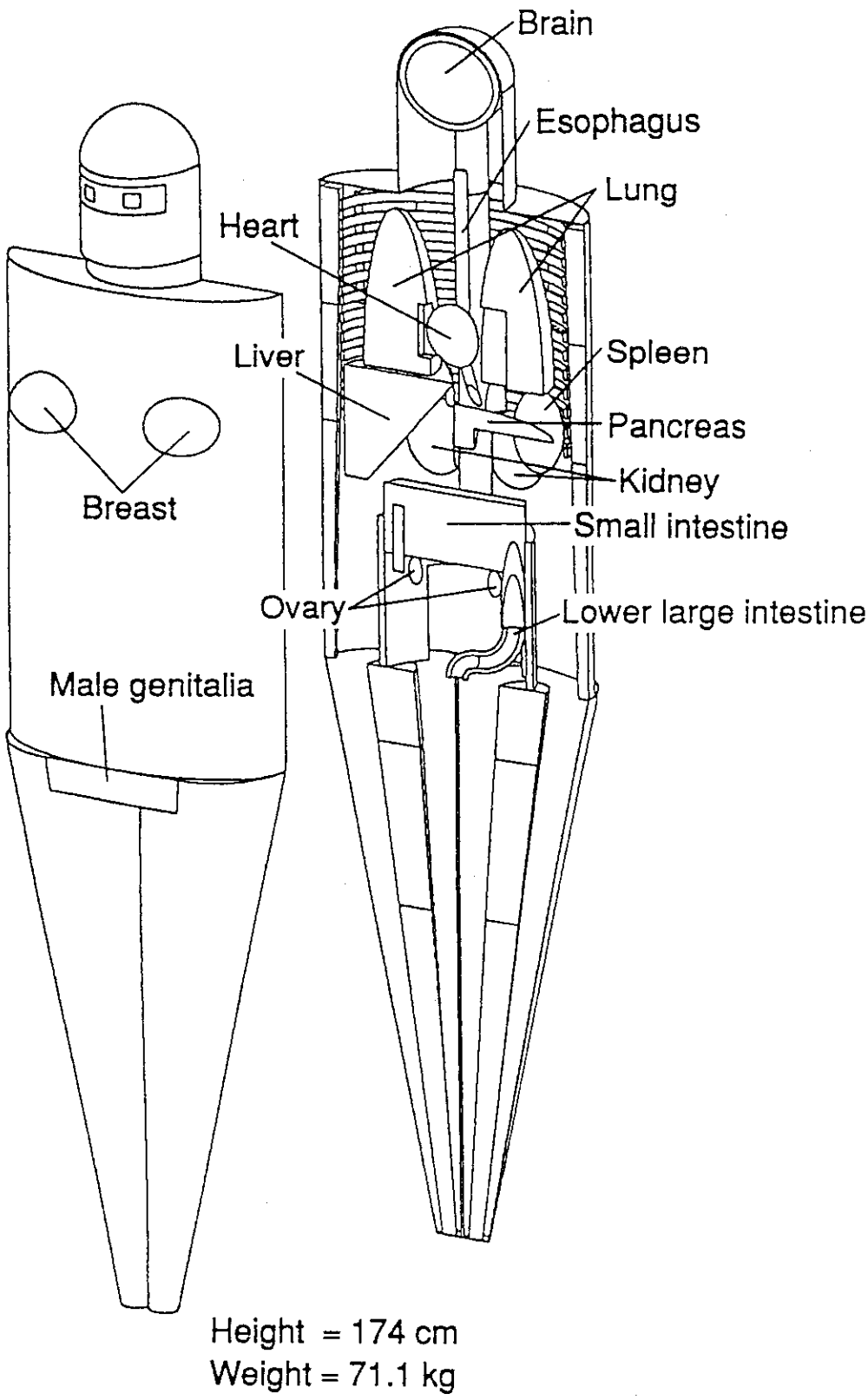


Fig.2.1 Human phantom

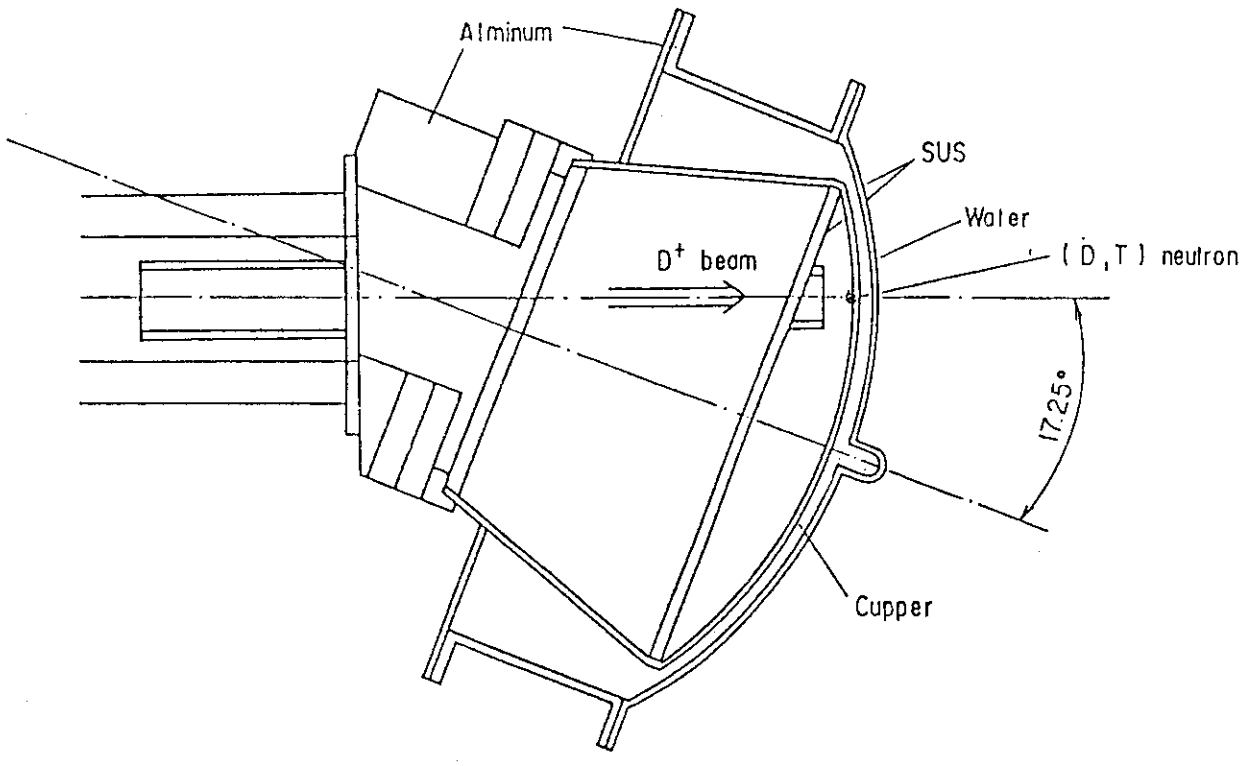


Fig.2.2 Cross sectional view of rotating target

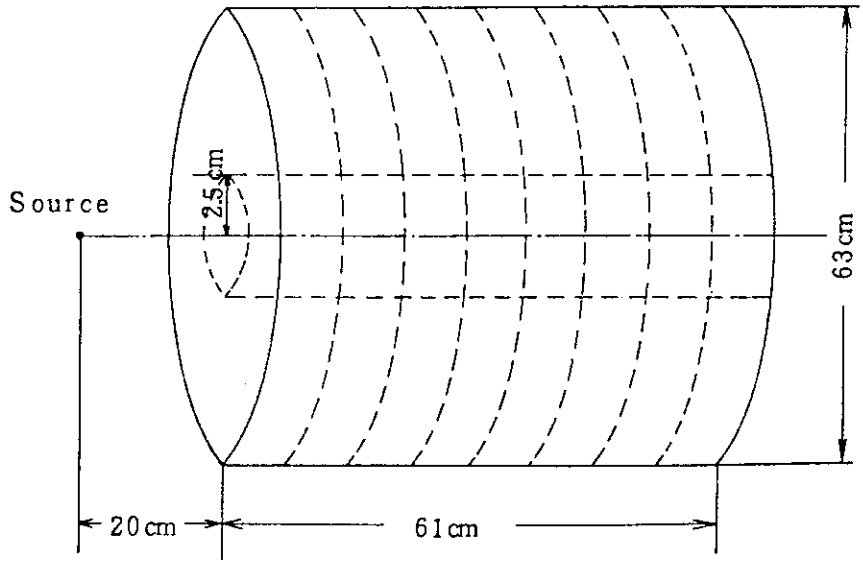


Fig.2.3 Li_2O cylindrical slab

3．並列化手法

3.1 概要

並列処理の目的は，１つのコードをいくつかの部分に分割して，複数のプロセッサ上で同時に（並列に）処理することにより，コードの処理時間（コードの処理が開始してから終了するまでの経過時間）を短縮させることにある。

モンテカルロ・コードでは粒子間の相互作用を考慮しないので，個々の粒子の挙動は互いに独立である。したがって，複数のプロセッサに個々の粒子を振り分けて（粒子分割），プロセッサ毎に粒子の誕生から消滅までのシミュレーションを並列に行うことが可能である。並列処理にはマクロタスク機能またはマイクロタスク機能を用いる２つの方法がある。前者がサブルーチンのような大きな単位の並列性に着目して並列化するのに対し，後者はDOループや文の集まりのような比較的小さな並列性に着目して並列化する。モンテカルロ・コードは，計算コストの集中した大きなDOループが存在しないため，マイクロタスク機能を用いて並列化しても大きな効果を得ることは難しく，マクロタスク機能を用いて並列化するのが望ましいと考えられる。即ち，今回の作業では各プロセッサにいくつかの粒子を割り当てるマクロタスク機能を用いて並列処理を行った。さらに，各プロセッサに割り当てられた粒子群はベクトル処理される。

Fig.3.1.(a) ～(c) にモンテカルロ・コードのオリジナル・スカラ版，ベクトル版，並列版の簡単な流れ図を示す。モンテカルロ・コードでは粒子が誕生してから消滅するまでを世代と呼んでいる。オリジナル・スカラ版は，複数の粒子の１世代のランダムウォーク・シミュレーションを行う粒子ループ（Particle Loop ）と，世代を繰り返し行う世代ループ（Batch Loop）の二重ループで構成されている。粒子ループはランダムウォーク・シミュレーションを１つの粒子ずつ逐次的に処理する。一方，ベクトル版はイベント・バンク方式と呼ばれる手法を用いて，１世代当りの粒子をまとめて処理するようにアルゴリズムが変更されている。まず，ランダムウォーク中の粒子に対するイベントを定義する。イベントとは粒子が体系内をランダムウォークして行く過程で必要になるZONEの検索処理，衝突後の状態の決定処理等を指す。同一のイベントが発生した粒子をイベントバンク（粒子バンクとも呼ぶが，本報告書では発生粒子を溜める粒子バンクと区別するためにイベントバンクと呼ぶことにする）と呼ばれる粒子溜めに集めると，同一のイベントバンク中の粒子に対しては同一の処理を施せば良いため，粒子に関してベクトル処理が可能となる。並列版は１世代当りの粒子を４つに分割し，各プロセッサに振り分け，それぞれ独立にシミュレーションを行う。その時，各タスクはベクトル処理を行う。

並列処理の効果を妨げる要因がまったくないと仮定した場合，Monte-4 における理想的な速度向上率は４倍となる。しかし，実際には，次に挙げるような粒度，負荷分散，メモリ競合によって性能効率が低下するため，理想的な速度向上率（台数効果）を得ることは困難である。

・粒度

粒度とは並列処理される部分の大きさである。高い速度向上率を得るためには，並列処理によるオーバヘッドが並列処理される部分に対して相対的に小さくなるように，粒度を十分に大きくする必要がある。並列処理によるオーバヘッドには，ⅰ）タスク（逐次実行される一連の処理）を生成，消去のための時間，ⅱ）データの初期化，各プロセッサで処理された結果を1つにまとめるための時間，ⅲ）排他制御（逐次実行が必要な処理について，同時に1つだけのタスクが実行する制御をする）や同期制御（あるタスクの処理が終了するのを待ち，次のタスク群の処理を行う制御をする）のための時間などがある。Fig.3.2（a）に示すように小さい粒度の並列処理が何度も行われると，オーバヘッドが大きくなってしまい，並列処理の効果が小さくなってしまう。

・負荷分散

負荷分散とは各プロセッサに割り当てられる仕事量の分布状態であり，各プロセッサで均等になるのが望ましい。Fig.3.2(b)に示すように各プロセッサの仕事量にばらつきがある場合，並列処理による効率が非常に悪くなる。1つのプロセッサのみ動作している場合，他のプロセッサはアイドル状態となるためである。4つのプロセッサが同時に動作を開始し，同時に終了すれば理想的な負荷分散となる。

・メモリ競合

共有メモリはすべてのプロセッサから同じようにアクセスすることができるが，メモリからベクトルレジスタ（スカラ処理の場合はキャッシュメモリ）へのロード（データの読み出し）命令やベクトルレジスタからメモリへのストア（データの書き込み）命令があった場合，すべてのプロセッサがこれらの処理を同時に行うことはできない場合がある。この場合，1つのプロセッサがメモリとロードやストアを行っている時は，他のプロセッサはその終了を待たなくてはならない。そのため，ロード／ストア命令の多いプログラム（メモリ負荷の高いコード）は，メモリ競合によって大きくその処理性能を阻害される。Fig.3.2(c)に示すように，メモリ競合が起こると各タスクの処理時間が増加するため，コード全体の処理時間も増加する。

3.2 マクロタスク機能を用いた並列化手法

本節ではMORSE コードのマクロタスク機能を用いた具体的な並列化の手法について述べる。

(1)並列処理部分のルーチン化

コードをマクロタスク機能を用いて並列化する場合に，まず行わなければならないことは，コードの並列性を見いだしてその部分を並列処理部分として抜き出し，ルーチン化することである。抜き出されたルーチンが並列処理されるタスクとなる。MORSE コードのランダムウォーク制御ルーチンMORSE は①入力データの読み込み，②断面積データの作成などのランダムウォークの前処理，③ランダムウォーク部分及び④ランダムウォークの結果得られたデータを統計処理する部分で構成されている。このうち，計算コストが大きく並列計算可能なランダムウォーク部分を並列処理部分として抜き出してサブルーチンBATCH とした。

(2)グローバル・コモンとローカル・コモン

並列処理機能を利用する場合，コードで使用するデータ（変数及び配列）は割り付ける方法によって共通データ，タスク間共通データ，ローカルデータの３種類に分類できる。共通データは，コードの開始から終了まで有効なデータであり，コードのローディング時にスタティック領域（静的な領域）に割り付けられ，すべてのタスクから参照できる。タスク間共通データは，１つのマクロタスクの開始から終了までの間有効なデータであり，マクロタスクの開始時にタスク毎のスタック領域（動的な領域）に割り付けられ，マクロタスクの終了時に解放される。ローカルデータは，１つのサブルーチンの処理が開始してから終了するまでの間有効なデータであり，サブルーチンの処理が開始した時にスタック領域に割り当てられ，サブルーチンの処理の終了時に解放される。"GLOBAL COMMON" で宣言されたデータは共通データとなり，"LOCAL COMMON"で宣言されたデータはタスク間共通データとなり，それ以外のデータがローカルデータとなる。並列処理機能を利用しなければ，すべてのデータは共通データとなる。

モンテカルロ・コードの場合，コード内の各データはその内容によって次のように分類できる。

ⅰ）共通データ
- 断面積データ，形状データ等各タスクで更新されず，参照のみされるデータ。
- 統計データ等各タスクで更新され，タスクが終了しても必要なデータ。
- 円周率等の参照のみされる定数データ。

ⅱ）タスク間共通データ
- 粒子の位置，方向，領域番号等の粒子属性を示すデータ。

ⅲ）ローカルデータ
- 一時変数，一時配列等の各サブルーチンでのみ使用し，そのサブルーチンの処理が終了すると必要なくなるデータ。

コードを並列化する場合，上記のデータの内容による分類を考慮して，各データに対して次のような作業を行う。

ⅰ）COMMON文で宣言された（GLOBAL COMMON 文で宣言しているのと同じ）データは共通データとなる。したがって，データの内容を考慮してタスク間共通データとすべきものは，LOCAL COMMON文の宣言に変更する。共通データとすべきものとタスク間共通データとすべきものが１つのコモンの中に混在している場合は，それぞれを分離して宣言する。

ⅱ）PARAMETER 文，DATA文，SAVE文で宣言されたデータは共通データとなる。PARAMETER 文で宣言されているデータは，コード内では参照しかされないので，共通データとして問題ない。しかし，DATA文，SAVE文で宣言されているが，各タスク内で更新される可能性のあるデータは，タスク間共通データとして宣言する。つまり，DATA文，SAVE文で宣言されたデータを共通データとするか否かは，そのデータの内容や定義・参照関係等を考慮する必要がある。

ⅲ）ローカルデータを共通データまたはタスク間共通データに昇格させる。

これらの作業にはANALYZER-Pの静的解析によって得られるコモンブロック相互参照リスト[10]や英字名相互参照リスト[10]の情報を参考にするとよい。また，ⅲ）の作業はコードが

FORTRAN77 の規約通りコーディングされていれば，本来必要のない作業である。しかし，実際にはiii）の作業が必要になるような規約違反ともいえるような方法を用いてコードが書かれていることが多い。例えば，データが暗黙のうちに0に初期化されていることを期待して，データの初期化を省略していたり，ローカルデータの値が暗黙のうちに保存されていることを期待して，SAVE文の宣言を省略している場合がある。このようなデータが存在した場合，並列処理における正常動作は期待できない。MORSE コードのサブルーチンGETNT を例に具体的に説明する。Fig.3.3 に示すようにGETNT はGETNT,STORNT,SETNTの3つの入口を持っている。MORSE コードでは1バッチにおいて追跡する粒子をバッチの最初にまとめて発生し，「粒子バンク」と呼ばれる領域に保存しておき，必要に応じてそこから粒子を取り出して処理するようになっている。GETNT は「粒子バンク」への保存，STORNTは「粒子バンク」からの取り出し，SETNT は「粒子バンク」の領域確保を行う。コードの最初にSETNT が呼び出され，ブランクコモン内の粒子バンクとして使用する領域の先頭アドレスNNO が定義される，GETNT,STORNTでは保存または取り出しを行うアドレスを計算するためNNO を参照している。ところがNNO はSAVE文やCOMMON文で宣言されていない。つまり，ローカルデータNNO が暗黙のうちに保存されていることを期待している。このまま並列処理すると，NNO はSETNT,GETNT,STORNTの実行時にスタック領域に確保されるため，その値は不定となる。つまり，SETNT で定義したNNO の値を参照することはできないので，コードの正常動作は期待できない。

並列処理によって起こるエラーは，タスク間共通データとすべきものを共通データとしたり，共通データとすべきものがローカルデータであったりというようなデータの誤用が原因となることが多い。並列化においては，各データを共通データ，タスク間共通データ，ローカルデータのどれに割り当てるかを決定することが最も重要な作業となる。

(3)タスクの生成と消去

コードを並列処理するには，主タスクから並列処理部分を子タスクとして生成する必要がある。マクロタスクの場合，主タスクはコードの処理開始時にシステムにより生成され，子タスクは並列処理関数PTFORK[11] により生成される。子タスクはPTFORKで呼び出した処理の正確なコピーである。主タスクとN-1 個の子タスクでN 並列処理を実現する。また，子タスクの消去は並列処理関数PTJOIN[11] によって行われ，待ち合わせ制御がされる。子タスクが終了していないのに主タスクの次処理（例えば，各タスクで行われた計算結果をまとめる処理）を行うと，正常な計算結果を期待できないので，待ち合わせ制御が必要になる。主タスクの処理は，N-1 個のジョイン命令が実行され，すべての子タスクが消去されたのを確認してから再開される。本コードにおいては，Fig.3.4 に示すように，コントロール・ルーチンMORSE からPTFORKでBATCH を呼び出して，3つの子タスクを生成する。その結果，4並列処理が実現する。

(4)排他制御

並列処理の問題の1つに，共通データの更新がある。複数のタスクが共通データを同時に変えようとするところに問題がある。並列処理は，共通データの更新は同時に1つだけのタスクが実行するような制御（排他制御）を必要とする。なぜなら共有メモリ型並列計算機では，す

べてのタスクの処理がランダムにスケジューリングされるため，共通データはどのような順番で変更されるかわからない。したがって，共通データの最終的な値は変更される順番によって異なることがある。簡単な例としてFig.3.5(a)のコードを考える。このコードを実行すると，データのロードのタイミングの違いによって，いくつかの計算結果があり得る。4つのタスクがデータS(=0) をロードし，計算後にデータS(=1.000000E+8) をストアすると，

S = 1.000000E+08

となる。また，3つのタスクがデータS(=0) をロードし，計算後にデータS(=1.000000+08) をストアしてから，残りのタスクがデータS(=1.000000+08) をロードして計算が行われると，

S = 2.000000E+08

となる。そこで，排他制御関数[1)] PLLOCKとPLUNLOCK[1)] を用いる。Fig.3.5(b)に示すように，PLLOCKとPLUNLOCKで囲まれた計算(S=S+1) は排他的に処理され，常に1タスクのみが共通データS を更新するので，正しい計算結果S(=4.000000E+08)を得ることができる。ただし，排他制御は各プロセッサに待ちを生じることになり，コードの高速処理を阻害する要因にもなる。したがって，高速化という観点からは排他制御はできるだけない方が望ましい。Fig.3.5(b)のコードは，Fig.3.5(c)のように排他制御を用いずに正常な値を得られるように書き換えることができる。本作業でも，以後述べる統計処理，粒子の発生，乱数などに排他制御が必要であったが，最終的には排他制御をまったく用いないコードに書き換えた。この作業は，まず排他制御を用いた正常動作をするものを作成し，1つずつ排他制御をなくしていくという手順で行った。

(5)統計処理

モンテカルロ・コードにおける中性子束等の統計的評価は，各粒子の寄与を各領域毎，エネルギー群毎に累積することによって行われる。各タスクの粒子が同じ領域，同じエネルギー群を持つ場合を想定して，(4)で述べたような排他制御を行う。Fig3.6に示すようなブランクコモンのsplitting counter やscattering counterはこれにあたる。さらにcollision,boundary-cross,escape 等の履歴や死粒子の履歴，ウェイト等の足し込みが並列処理部分で行われ，排他制御が必要である。これらを排他制御を用いずに処理するために，Table.3.1 のような並列処理用の作業用配列を定義する。並列処理部分ではこの作業用配列に対して足し込みを行い，タスクがすべて終了したらそれを1つにまとめて出力等の処理を行う。

(6)粒子の発生方法

MORSE コードでは，まずある世代についてシミュレーションする粒子をまとめて発生させ，「粒子バンク」に保存しておき，必要に応じて取り出した粒子を追跡する。スプリット粒子のような二次粒子もこの領域に一度保存される。したがって，並列処理をする場合，この粒子の保存と取り出しの処理に対して排他制御が必要になる。MORSE コードではブランクコモンの一部を「粒子バンク」としている。「粒子バンク」を4等分して，各タスクにそれぞれを割り当てた。さらに，粒子の発生をタスク生成後に各タスクにおいて行うようにした。その結果，各タスクの「粒子バンク」の制御は独立に行うことが可能になった。

(7)並列計算用乱数（低並列の場合）

モンテカルロ・コードでは粒子の飛程，発生位置等を決定したり，Russian Rouletteを行う時などに乱数を使用する。現在モンテカルロ・コードで広く使用されている合同法の場合，乱数は、漸化式によって逐次的に生成される。このため、並列計算においては、排他制御を行いながら乱数の発生，あるいは予め発生した乱数を配列から読み出す処理を行うことが考えられる。しかし，この「排他制御による乱数列の共用」を行った場合，次のような問題がある。

ｉ）排他制御により、各タスクの粒度が小さくなる。この結果，負荷の不均衡が生じ、並列計算の性能が低下する。

ii）非常に小さな時間単位に着目した場合，あるプログラムを複数台のプロセッサを用いて並列処理した時の各プロセッサの状態（例えば，計算の進み具合、乱数列を参照するタイミング等）は，コードを書き換えた後実行する場合場合に応じて異なる。このため，一つの乱数列を複数台のプロセッサが共用した場合，計算結果は常に一致するとは限らない，というよりも，計算を行うごとに結果が異なる時のほうが多い。即ち，同じ入力データに対し、同じ計算結果を得ることができる（計算結果が完全に一致する）という計算の再現性が失われる。計算結果が異なるため，並列化や高速化の作業が正しく行われたかどうかを判断するのが難しくなる。例えば，あるDOループをベクトル化したとする。ある書換えを行った後コードを実行すると，計算結果が書換え以前と異なっている。この場合，ベクトル化の作業でコーディングミスがあった，ベクトル処理による影響（最適化等），使われた乱数の順番が前と違うなど複数の原因を考慮しなければならない。再度コードを実行すると，また計算結果が異なるかもしれない。このように，乱数列の違いによる計算結果の差異は作業を効率良く進めていく上で大きな障害となる。

そこで，本コードでは「排他制御による乱数列の共用」をせずに，各タスクで独立な乱数を生成している。即ち，Fig.3.7 に示すように、連続する最初の４つのシードを各タスクに割当て，各タスクで重なりがないように４飛びの乱数を生成する。生成法については次のとおり。

いま，一様乱数生成式として，次のような合同乗算法を考える。[12]

$$X_n = MOD(a \cdot X_{n-1} + b, M) \quad （\cdot は，乗算を表す）$$

即ち，

$$X_1 = MOD(a \cdot X_0 + b, M)$$
$$X_2 = MOD(a \cdot X_1 + b, M)$$
$$X_3 = MOD(a \cdot X_2 + b, M)$$
$$X_4 = MOD(a \cdot X_3 + b, M)$$
$$\vdots$$
$$X_k = MOD(a \cdot X_{k-1} + b, M)$$

である。本コードでは，４並列乱数生成式として次式を使用する。

$$X_n = MOD(a^4 \cdot X_{n-4} + a^3 \cdot b + a^2 \cdot b + a \cdot b + b, M)$$

ここで，$A=a^4$,$B=a^3\cdot b+a^2\cdot b+a\cdot b+b$ とおくと，

$$X_n = MOD(A\cdot X_{n-4}+B,M)$$

である。即ち，定数A,B が決まれば，今までと同様の方法で4並列乱数を生成することができる。また，2並列，3並列乱数も同様の手法で生成することができる。

定数A,B の計算手法について述べる。本コードにおいては，a=32771,b=1234567891,$M=2^{31}$ とする[12] 生成式を用いた。この場合，それぞれの並列乱数を生成する定数A,B は次のようになる。

並列数 = 1 : A = 32771 B = 1234567891
並列数 = 2 : A = 1073938441 B = 666995532
並列数 = 3 : A = 1074626587 B = 109094071
並列数 = 4 : A = 3539025 B = 796094712

ただし、この値は代数計算で求めたものであり，計算機上で通常の乗算及び加算を用いて計算していくと，桁溢れあるいは桁落ちのために正しい結果を求めることができない。即ち、整数を用いた場合、10進数で7桁以上の整数は桁溢れが起き、倍精度実数を用いた場合でも，有効桁数は10進数で16桁（2進数で56桁）であり[13]，それを越えると桁落ちが起きて正しく計算されない。例えば3並列乱数を生成する定数A,B は

$A = a^3 = 35194036650011$

$B = a^2\cdot b + a\cdot b + b = 1325849916169197931$

であるから，Bの計算に関して桁落ちが起きる。また，

$MOD(A,M) + MOD(B,M) = MOD(A+B,M)$

$MOD(A,M) \cdot MOD(B,M) = MOD(A\cdot B,M)$

である点を考慮し，1演算ごとにMOD をとりながら計算した場合でも，次のとおり正しい結果を求めることができない。

（計算式）	（計算機による計算結果）	（代数計算による計算結果）
a1 = dmod(a•a,M)	1073938441	1073938441
A = dmod(a1•a,M)	1074626587	1074626587
b1 = dmod(a1+a,M)	1073971212	1073971212
b2 = dmod(b1+1,M)	1073971213	1073971213
B = dmod(b2•b,M)	109093888	109094071

計算機によって求めた結果と代数計算によって求めた結果を比較すると，B の計算の時に両者の結果が違っていることがわかる。これは"b2•b" の値が次のように10進数の場合の有効桁数16桁を越えているため，正しい値を得ることができないことが理由である。

$b2\cdot b = 1073971213\cdot 1234567891 = 1325890375428121783$

そこで，本コードでは，桁落ちを防ぐために次の手法を用いた。いま，X1，X2という二つの整数の演算（乗算を行った後、2^{31} の剰余をとる）を考える。ここで，X1，X2は、倍精度実数として表現されているが，本来，整数である。整数であるから，X1，X2を

$X1 = a1\cdot 2^{16} + b1 \quad (a1<2^{15})$

$X2 = a2\cdot 2^{16} + b2 \quad (a2<2^{15})$

とすると

$$X1 \cdot X2 = (a1 \cdot 2^{16} + b1)(a2 \cdot 2^{16} + b2)$$
$$= a1 \cdot a2 \cdot 2^{32} + (a1 \cdot b2 + a2 \cdot b1) \cdot 2^{16} + b1 \cdot b2$$

となる。ここで,

$$M > K \text{ の時, } MOD(A \cdot 2^K, 2^M) = MOD(A, 2^{M-K}) \cdot 2^K$$

であるから，a1・a2の項を消去し，

$$MOD(X1 \cdot X2, 2^{31})$$
$$= MOD(a1 \cdot a2 \cdot 2^{32} + (a1 \cdot b2 + a2 \cdot b1) \cdot 2^{16} + b1 \cdot b2, 2^{31})$$
$$= MOD(MOD(a1 \cdot b2 + a2 \cdot b1, 2^{15}) \cdot 2^{16} + b1 \cdot b2, 2^{31})$$

とし，桁落ちを起こさずに二つの数の演算を行うことができる。

以上の点を考慮し，本コードにおいては，並列計算用乱数を次のように生成している。即ち，定数A，b，及びX_n を計算するためのX_{n-1} を

$$A = ra0 \cdot 2^{16} + ra1$$
$$B = rb0 \cdot 2^{16} + rb1$$

$X_{n-1} = rx0 \cdot 2^{16} + rx1$とおき,

$$X_n = MOD(A \cdot X_{n-1} + B, 2^{31})$$
$$= MOD(MOD(ra0 \cdot rx1 + rx0 \cdot ra1, 2^{15}) \cdot 2^{16} + ra1 \cdot rx1 + B + rb0 \cdot 2^{16} + rb1, 2^{31})$$

とする。この式を用いて乱数を生成したところ，正しく4飛びの乱数が生成されていることが確認できた。

乱数を排他制御を用いずに各プロセッサで独立に計算することで，各タスクの粒度は大きくなった。また，各タスクで使われる乱数列が常に同じになったことで，同じ計算結果を常に得られるようになった。その結果，コードのデバックも容易になった。

Table.3.1(a) Array description used for parallelization

並列処理用ワーク配列	オリジナル配列	内　　容
NPSCLP(13,MAXCPU)	NPSCL(13)	Counters of events for each batch
NDEADP(5,MAXCPU)	NDEAD(5)	Number of deaths of each type
DEADWP(5,MAXCPU)	DEADWT(5)	Weight equivalent to NDEAD

NPSCL(13)

1) SOURCE GENERATED
2) SPLITTINGS OCCURING
3) FISSIONS OCCURING
4) GAMMA RAYS GENERATED
5) REAL COLLISIONS
6) ALBEDO SCATTERINGS
7) BOUNDARY CROSSINGS
8) ESACPES
9) ENERGY CUT-OFF
10) TIME CUT-OFF
11) RUSSIAN ROULETTE KILLS
12) RUSSIAN ROULETTE SURVIVORS
13) GAMMA RAYS NOT GENERATED BECAUSE BANK WAS FULL

NDEAD(5),DEADWT(5)

1) RUSSIAN ROULLETTE KILL
2) PARTICLE ESCAPED THE SYSTEM
3) PARTICLE REACHED ENERGY CUT-OFF
4) PARTICLE REACHED AGE LIMIT
5) NOT USE

Table.3.1(b) Array description used for parallelization

並列処理用ワーク配列	オリジナル配列	内　　容
NSCTP(IG,NREG,MAXCPU)	NSCT(IG,NREG)	Number of real scatterings
WSCTP(IG,NREG,MAXCPU)	WSCT(IG,NREG)	Weight equivalent to NSCT
NALBP(IG,NREG,MAXCPU)	NALB(IG,NREG)	Number of albedo scatterings
WALBP(IG,NREG,MAXCPU)	WALB(IG,NREG)	Weight equivalent to NALB
NFIZP(IG,NREG,MAXCPU)	NFIZ(IG,NREG)	Number of fissions
WFIZP(IG,NREG,MAXCPU)	WFIZ(IG,NREG)	Weight equivalent to NFIZ
NGAMP(IG,NREG,MAXCPU)	NGAM(IG,NREG)	Number of secondary productions
WGAMP(IG,NREG,MAXCPU)	WGAM(IG,NREG)	Weight equivalent to NGAM
NSCAP(IMED,MAXCPU)	NSCA(IMED)	Scattering counter
NSPLP(IG,NREG,MAXCPU)	NSPL(IG,NREG)	Splitting counter
WSPLP(IG,NREG,MAXCPU)	WSPL(IG,NREG)	Weight equivalent to NSPL
NOSPP(IG,NREG,MAXCPU)	NOSP(IG,NREG)	Counter for full bank when Splitting was requested
WNOSP(IG,NREG,MAXCPU)	WNOS(IG,NREG)	Weight equivalent to NOSP
NRKLP(IG,NREG,MAXCPU)	RRKL(IG,NREG)	Russian roulette death counter
WRKLP(IG,NREG,MAXCPU)	WRKL(IG,NREG)	Weight equivalent to RRKL
NRSUP(IG,NREG,MAXCPU)	RRSU(IG,NREG)	Russian roulette survival counter
WRSUP(IG,NREG,MAXCPU)	WRSU(IG,NREG)	Weight equivalent to NSPL

注）オリジナル配列はブランクコモンの中にとられていて，コード内ではNSCT,WSCT---等の配列を用いずに，ブランクコモンのアドレスを直接指定して定義・参照が行われている。

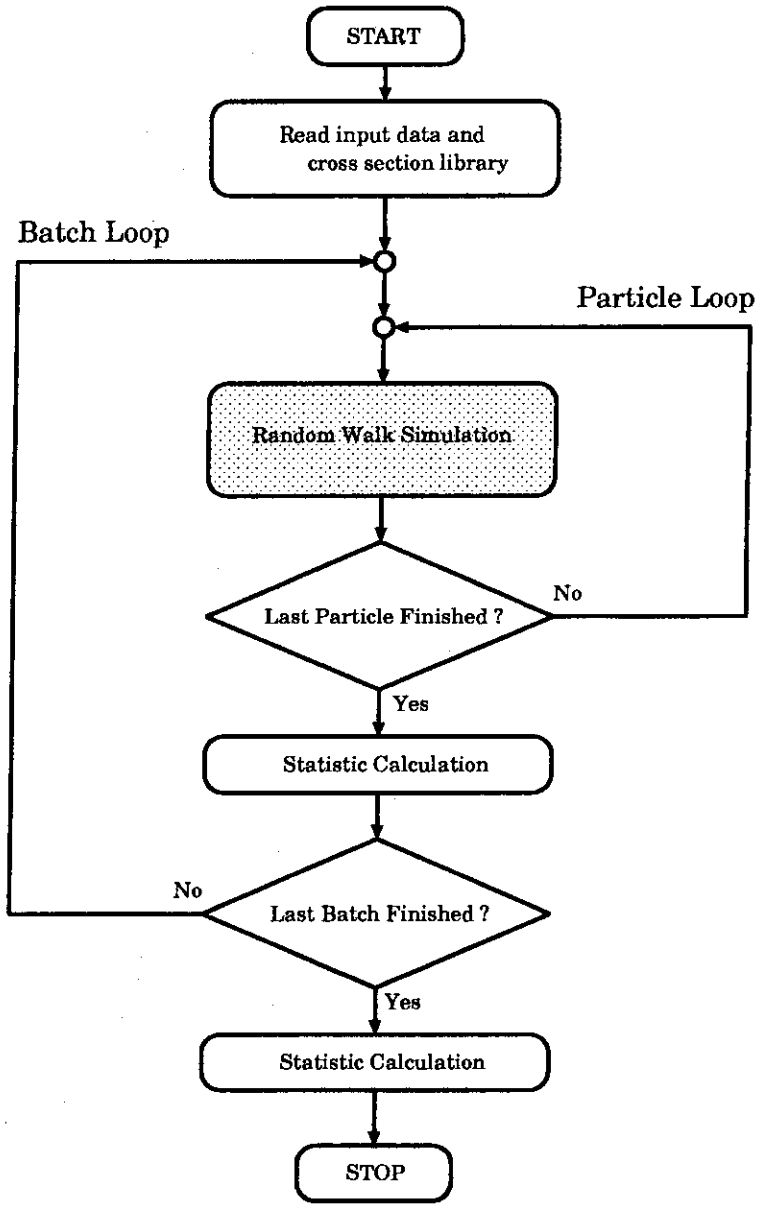


Fig.3.1(a) Flow diagram of original MORSE code

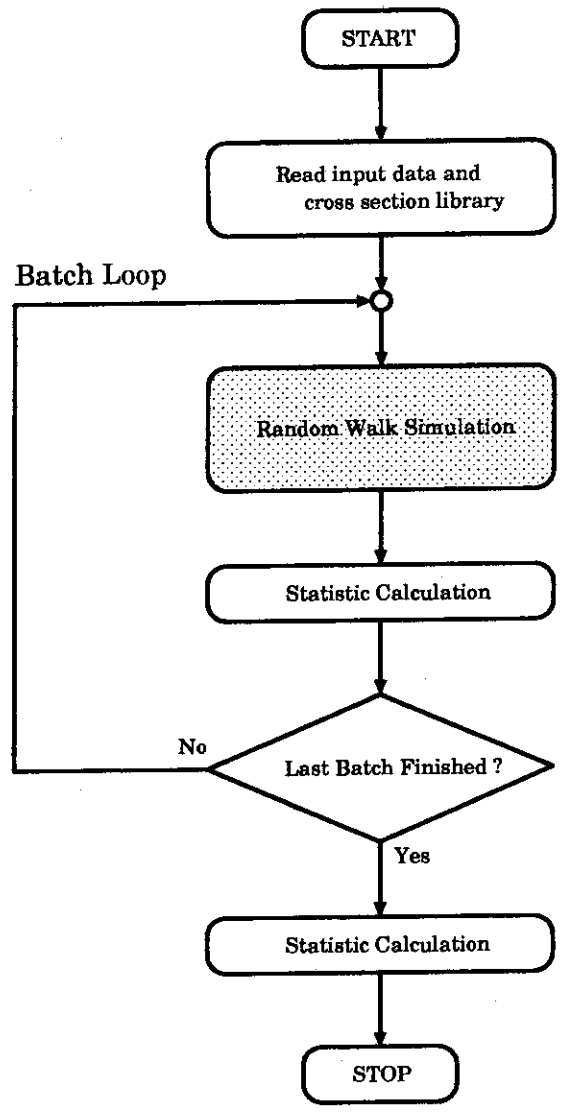


Fig.3.1(b) Flow diagram of vectorized MORSE code

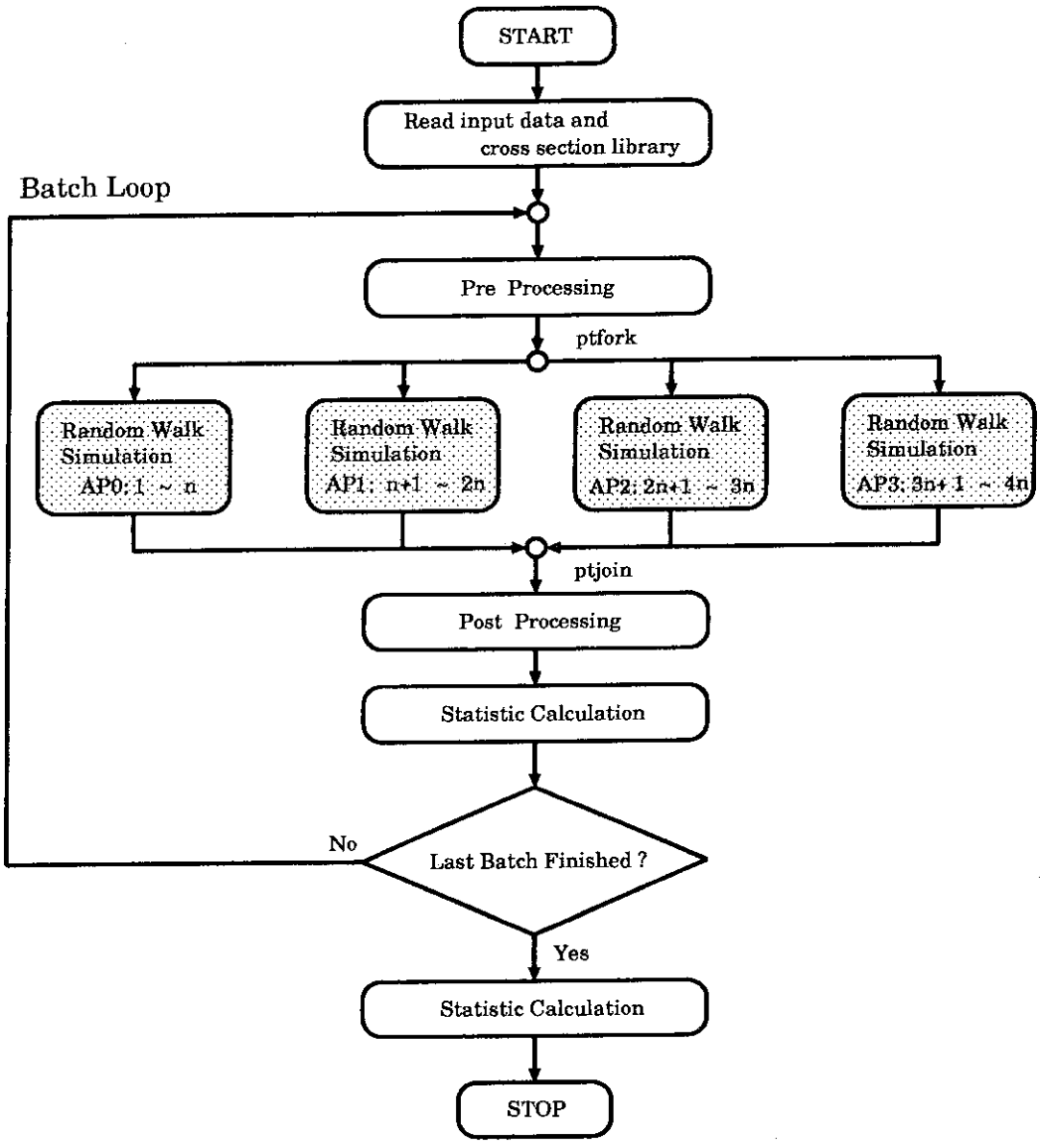
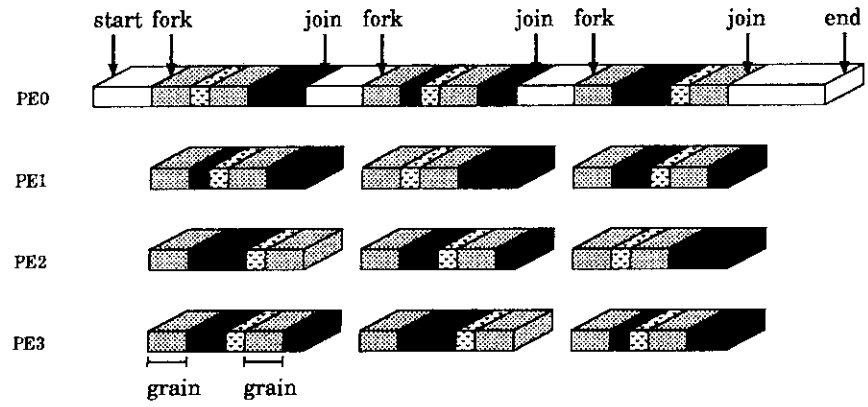


Fig.3.1(c) Flow diagram of parallelized MORSE code

Fine -Grain



Coarse-Grain

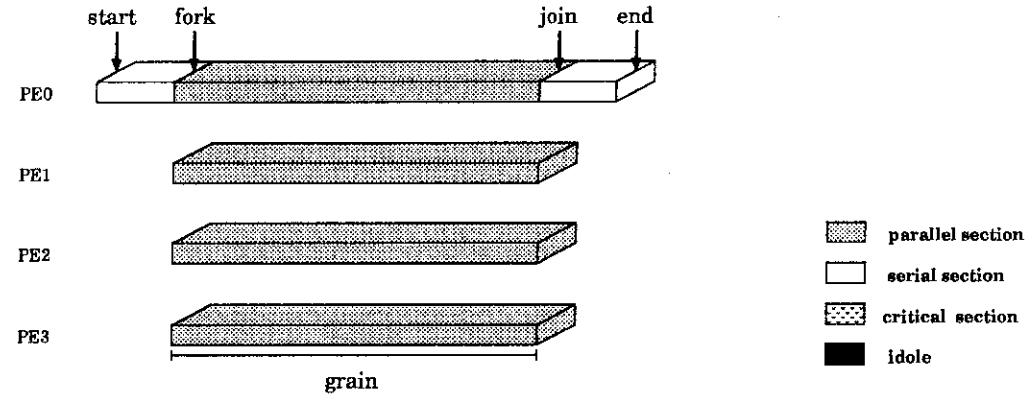


Fig.3.2(a) Scheduling of the fine-grain and coarse-grain programs

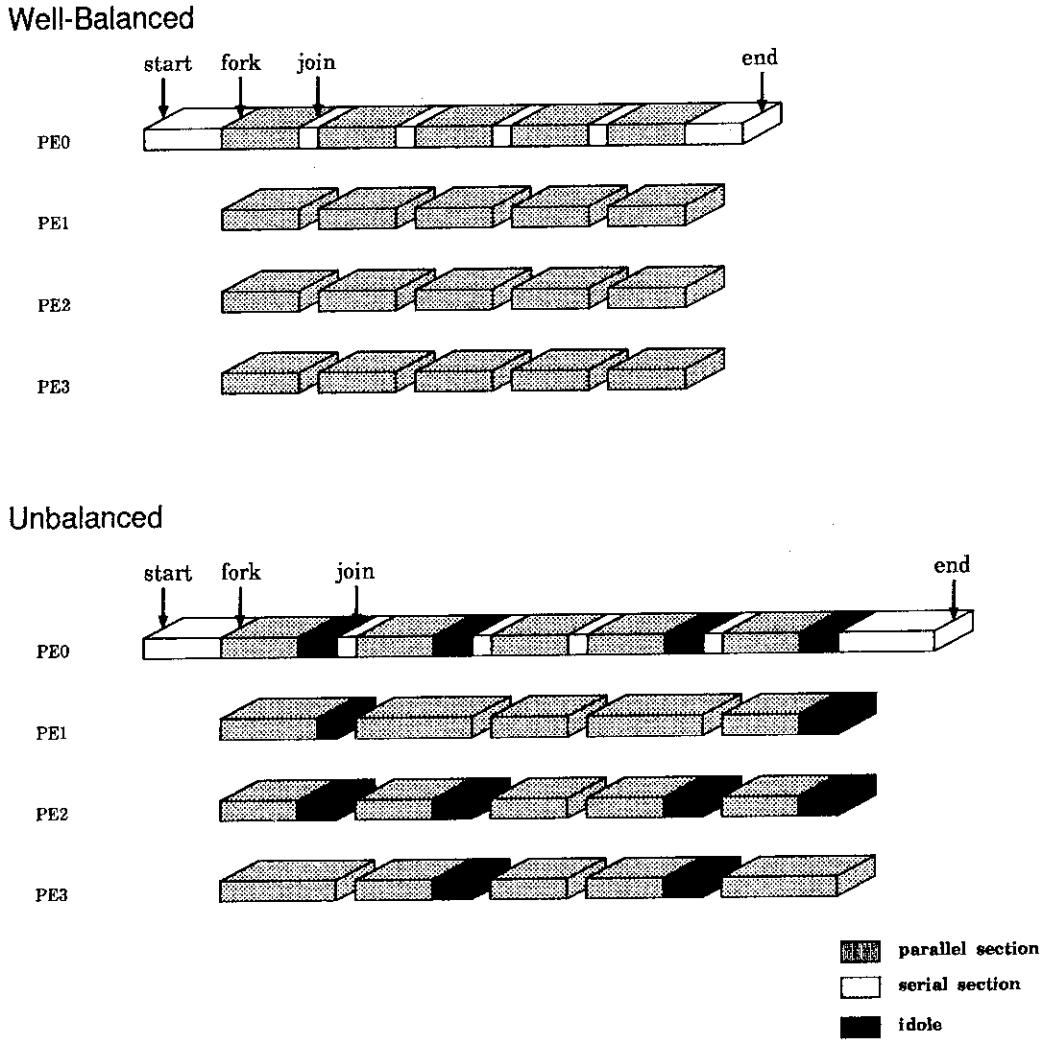


Fig.3.2(b) Load balance in parallel processing

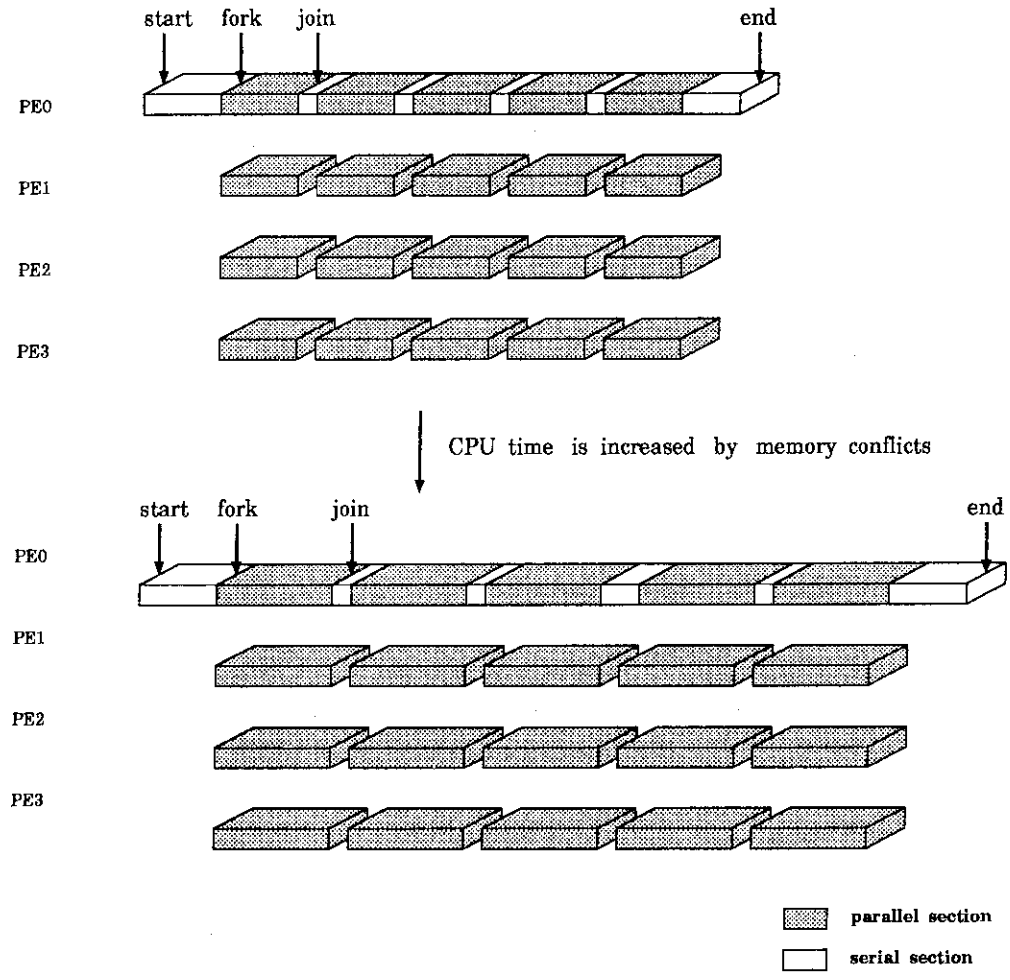


Fig.3.2(c) Memory conflicts in parallel processing

```
      K4 = 2*(K-1)
      IG = I2B(K4)
      U  = BC(K)
      K  = K + 1
      V  = BC(K)
      K  = K + 1
      W  = BC(K)
      K  = K + 1
            ⟆
      NREG = I2B(K4)
      RETURN
      ENTRY STORNT(N)
      K = NNO + 12*N - 10
      K4 = 2*(K-1)
      I2B(K4) = IG
      BC(K) = U
      K = K + 1
            ⟆
      I2B(K4) = NREG
      RETURN
      ENTRY SETNT(NLAST,NMOST)
      NNO = NLAST
      NLAST = NLAST + 12*NMOST
      RETURN
      END
```

Fig.3.3 FORTRAN statement of subroutine GETNT

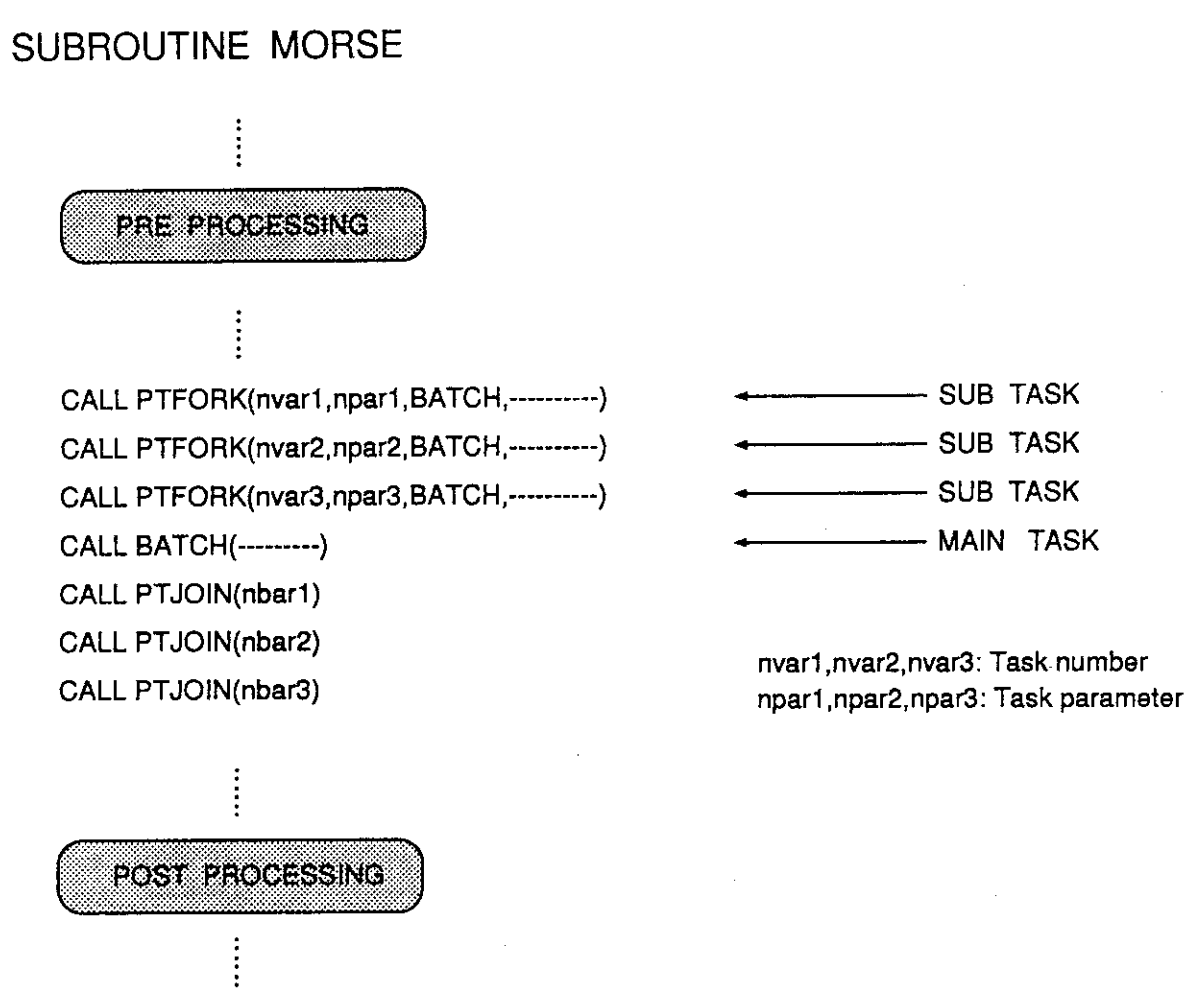


Fig.3.4 Timing model of fork and join in MORSE code

(a)

```
      EXTERNAL SUB
      INTEGER TVAR(4),TPARM(4)
      S = 0
      PTFORK(TVAR(3),TPARM(3),SUB,S)
      PTFORK(TVAR(2),TPARM(2),SUB,S)
      PTFORK(TVAR(1),TPARM(1),SUB,S)
      CALL SUB(S)
      WRITE(6,*) S
      STOP
      END
```

```
      SUBROUTINE SUB(S)
      DO 100 I=1,1000
      DO 100 I=1,1000
        S = S + 1
  100 CONTINUE
      RETURN
      END
```

(b)

```
      EXTERNAL SUB
      INTEGER TVAR(4),TPARM(4)
      CALL PLASGN(TVAR)
      S = 0
      PTFORK(TVAR(3),TPARM(3),SUB,S,LTVAR)
      PTFORK(TVAR(2),TPARM(2),SUB,S,LTVAR)
      PTFORK(TVAR(1),TPARM(1),SUB,S,LTVAR)
      CALL SUB(S,LTVAR)
      WRITE(6,*) S
      STOP
      END
```

```
      SUBROUTINE SUB(S,LTVAR)
      DO 100 J=1,1000
        DO 100 I=1,1000
      CALL PLLOCK(LTVAR)
            S = S + 1
      CALL PLUNLOCK(LTVAR)
  100 CONTINUE
      RETURN
      END
```

(c)

```
      EXTERNAL SUB
      INTEGER TVAR(4),TPARM(4)
      DIMENSION SS(4)
      S = 0
      SS(1) = 0
      SS(2) = 0
      SS(3) = 0
      SS(4) = 0
      PTFORK(TVAR(3),TPARM(3),SUB,SS(4))
      PTFORK(TVAR(2),TPARM(2),SUB,SS(3))
      PTFORK(TVAR(1),TPARM(1),SUB,SS(2))
      CALL SUB(SS(1))
      S = SS(1) + SS(2) + SS(3) + SS(4)
      WRITE(6,*) S
      STOP
      END
```

```
      SUBROUTINE SUB(S)
      DO 100 I=1,1000
      DO 100 I=1,1000
        S = S + 1
  100 CONTINUE
      RETURN
      END
```

Fig.3.5 Examples of exclusive control

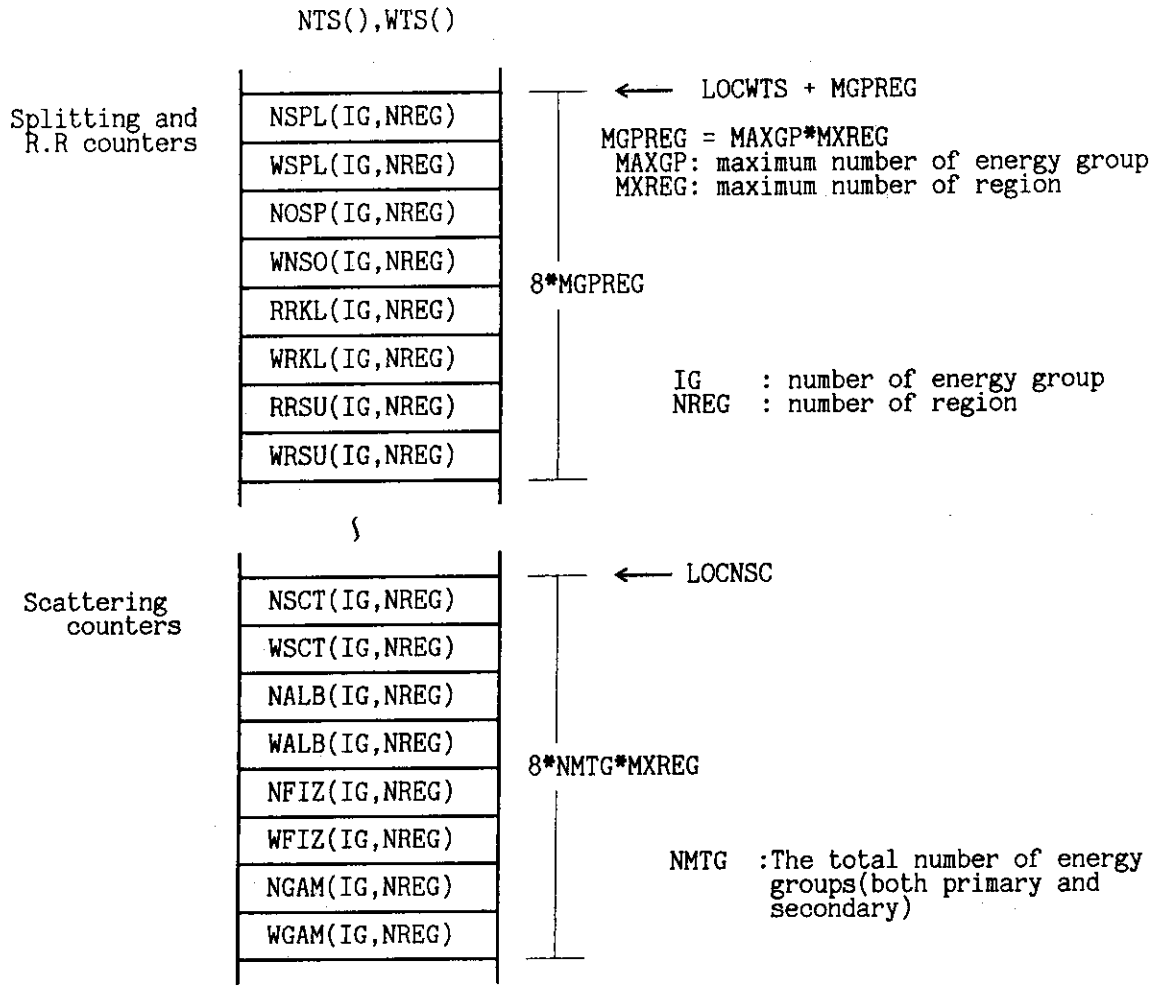
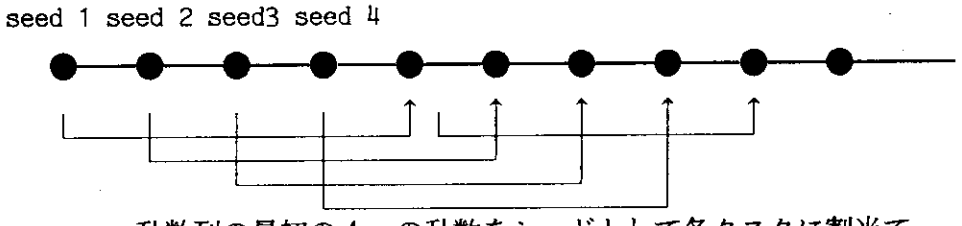


Fig.3.6 Bookkeeping counter of MORSE code



乱数列の最初の４つの乱数をシードとして各タスクに割当て，
４飛びに乱数を生成する。

Fig.3.7 Random number sequence in parallel processing

4．高速化手法

4.1 サブルーチンGGV に対するチューニング

Fig.4.1 にサブルーチンGGV の流れ図を示す。GGV は各Bodyにおいて粒子の存在する位置からBodyの入口と出口までの距離RIN,ROUT及び入口面と出口面LRI,LRO を計算する。GGV は粒子の飛程を識別するインデックスとRIN,ROUT,LRI,LRO等を保存しておき，次にGGV が呼び出されたときは，同じ飛程と同じBodyの交差計算は行わずに，保存されているデータを参照する。即ち，GGV が呼び出されると，まず，各粒子の飛程を識別するインデックスとBody番号を保存されているデータと比較し，、交差計算の必要な粒子と不要な粒子に分類する。次に，粒子が現在属するZoneの幾何形状によって粒子をイベントバンクに分類し，各イベントバンクごとに各粒子のRIN,ROUT,LRI.LROを計算する。拡張版（FANTOME 版）MORSE コードは，任意六面体，球，直円柱，直楕円柱，円錐台，回転楕円体，直方体（回転可），三角柱，直方体（回転不可）に一般楕円体，トーラス，楕円錘台の幾何形状を拡張し，全部で１２の幾何形状が使用可能になっている。

GGV は主に粒子追跡処理部分において"BATCH-NXTCOL-GOMST-G1V-GGV"という経路で呼ばれる。サブルーチンNXTCOLにおいては，各粒子の衝突点を求めるために，各粒子の軌跡の追跡を行っている。各粒子の軌跡が境界を通る回数はさまざまであるため，計算が進むにつれて，GGV において処理を受ける粒子数は極端に減少する。この様子をFig.4.2 に示す。問題１においては１２の形状がすべて使用されている。そのため，更に粒子が１２のイベントバンクに分類されるため，ベクトル長が短くなってしまう。１２の形状の処理はすべてベクトル化し，後に述べるような最適化等もきめ細かく行ったが，大きな効果は得られなかった。これは最もコストの集中しているGGV が非常に深い所に位置するルーチンであるため，粒子が途中で多くのイベントバンクに分散してしまった状態で処理されているのが原因である。

4.2 サブルーチンG1V に対するチューニング

サブルーチンG1V は粒子が現在属するZone内での飛程距離と次に入るべきZoneの番号を決定する。今回はFACOM VP100 時代のメモリ容量の制限のため使用されなかった「直線と三次元領域との交差判定に関する学習機能」をオリジナル・コードと同様に動作するよう書き換えを行った。

粒子が現在属しているZoneから次に入るZoneを検索する場合，すべてのZoneと交差判定を行い，交差点までの距離が一番小さいものが求めるZoneとなる。このZone検索には多くの処理時間を必要とするが，現在属するZoneに隣接しているZoneがわかっていれば，そのZoneのみを交差判定すればよいのは明らかであり，すべてのZoneを検索の対象にするのに比べて処理時間を短縮することができる。そこで，MORSE コードでは予め隣接Zoneリストを作成しておき，それによる検索を優先して行い，リストから見つからなかった場合にのみ，すべてのZoneを検索する，という方法を用いてZone検索，つまり直線と三次元領域との交差判定に要する処理時間の

短縮を図っている。隣接Zoneリストは，最初は何も情報を持っていないが，全Zone検索の結果を隣接Zoneとして追加する操作を繰り返す（学習する）ことによって，最初はすべてのZoneを検索していても，最終的には隣接Zoneのみを検索するようになり，無駄な計算が行われなくなる。

隣接リストによる検索は，Fig.4.3 に示すMAアレーと呼ばれるブランクコモンに格納されているデータ(c),(d),(e),(f) を参照して行う。粒子が現在属するゾーンのどのボディで交差するか決定したら，まず(c) のゾーンが隣接Zoneとしてチェックされる。そして(c) が求めるZoneでなかったときには，次にチェックする隣接Zoneのアドレスを(d) から読み込み，そのデータによってポインタを移動する。移動したポインタ(e) に格納されている隣接Zone番号をチェックする。求めるZoneでなければ，次にチェックする隣接Zoneのアドレスを(f) から読み込み，ポインタを移動する。このようにポインタを移動しながら隣接Zoneリストの最後まで順番にチェックしていく。また，MAアレーの大きさLTMAはサブルーチンJOMIN で定義され，LTMAの範囲以内で学習は継続される。

ベクトル版及び並列版ではMAアレーの隣接Zoneリストを使用せずに，Fig.4.4 のような配列に隣接Zoneに関する情報を格納し，これを使用してZone検索が行われるように改良した。連続エネルギーコードVIM もMORSE コードのような学習機能を持っており，ベクトル化の際に同じように隣接Zone配列を使用するように書き換えられている[3]。これを参考に書き換えを行った。ただし，VIM が各Zoneに隣接するZoneのリストを作成しているのに対し，MORSE は各Zoneを構成するBody毎にリストを作成した。Zone N1 のBody N2 で交差する場合，隣接Zoneとして NXTBO(N1,N2,1)からNXTBO(N1,N2,NXTNO(N1,N2)) までをチェックし，見つからなかった場合にのみ全Zoneを検索する。その時に見つかったZoneは，隣接ZoneリストとしてNXTBO に格納される。このようにしてオリジナル・コードと同様な動作を実現した。１プロセッサ・１タスク・ベクトル処理の場合，この改良によるコード全体に対する速度向上率は１．２倍，処理時間短縮は１４％である。

また，並列処理する場合，１つのNXTBO,NXTNO で処理するには排他制御が必要となる。その場合，排他制御に関連する範囲はG1V の大部分になる。そのため著しい性能低下を招くので，各タスク用にNXTBO,NXTNO を用意して，各タスクが独立に処理できるようにした。ただしその場合は各タスクで同じ学習が重複して行われることになる。

4.3 インライン展開による高速化

呼び出し回数の多いサブルーチンをインライン展開することによって，呼び出しにかかるオーバヘッド分の処理時間を短縮することができる。また，DOループ内に外部呼び出し関数があるためにベクトル化できない場合，その関数をインライン展開することによって，そのループのベクトル化が可能になることがある。MORSE コードでは次のようなルーチンをインライン展開した。インライン展開はコンパイラによる方法と手作業でソースを書き下す方法を用いて行った。

・乱数ルーチン

・インターフェース・ルーチンVINTERP
・粒子取りだしルーチンGETNV

(1)乱数ルーチン

MORSE コードには次の乱数ルーチンが用意されている。[9]

FLTRNF(0),SFTRNF(0),EXPRNF(0),AZIRN(SIN,COS),POLRN(SIN,COS),GTISO(X,Y,Z),
RNMAXF(T),FISRNF(0),RNDIN(R),RNDOUT(R)

これらのルーチンはもともとはアセンブラ・ルーチンであったが，既にFORTRAN ソースに書き換えられている。オリジナル・スカラ版では一様乱数発生ルーチンFLTRNFを用いて逐次的に乱数を生成していたが，ベクトル処理のためには大量の乱数を一括して生成する必要があり，FLTRNFの代りにRANU22を用いた。RANU22はFACOM VP2600でベクトル化した時に使用した科学技術計算ライブラリSSL-IIのRANU2[12]と同じ乱数を生成するFORTRAN ルーチンである。このRANU22はコンパイラによってインライン展開し，それ以外のルーチンは手作業でインライン展開した。しかし，乱数ルーチンのしめるコスト比率が小さいため，大きな速度向上は得られなかった。

(2)インターフェース・ルーチンVINTERP

ベクトル化されたルーチンでは，これまでスカラ変数であった，粒子に依存した変数が配列化されているため，ベクトル・ルーチンからスカラ・ルーチンを呼び出す場合，スカラ変数とベクトル配列で整合をとる必要がある。そのためFig.4.5 に示すようにスカラ変数にそれに対応する配列から値を入れたり，その逆の処理を行うインターフェース・ルーチンを用いた。ベクトル化されていないルーチンが残っている部分で，このようなインターフェース・ルーチンを用いる。インターフェース・ルーチン自身は処理時間をあまり必要としないが，呼び出しのオーバヘッドは無駄に消費されてしまうので，インライン展開してそれを削減した。

(3)サブルーチンGETNV

「粒子バンク」に保存された粒子は，サブルーチンGETNT によって１粒子づつ取り出され，一度スカラ変数（例えばコモンNUTRON）に格納され，インターフェース・ルーチンVINTERP によってベクトル配列（例えばコモンVNUTRON ）に格納される。スカラ変数への格納は無駄な処理であり，GETNT とVINTERP の処理を同時に行うサブルーチンGETNV を作成した。その結果，各粒子の属性に関するデータは，「 粒子バンク」からベクトル配列に値を直接格納されるため，無駄な処理が省かれて処理が高速化された。

さらにこれをコンパイラによってインライン展開した。その結果，サブルーチンの呼び出しによるオーバヘッドが削減され，インライン展開したDOループがベクトル処理可能になった。

4.4 平均ベクトル長の増大

モンテカルロ・コードをベクトル処理する場合，個々の粒子の振舞いはそれぞれ異なるため，追跡している粒子が幾つかのイベントバンクに分散してしまう。分散した粒子がさらに分散し

てしまうこともあり，各イベントのベクトル長が非常に短くなってしまう。また，モンテカルロ・コードはEnergy-Cutoff , Russian Roulette, Age-Cutoff, Escapeなどによってすべての粒子が消滅するまで処理を続けていくため，計算が進むに連れて処理粒子数が少なくなってしまう。つまり，最初はベクトル長が十分あっても，徐々にあるいは急激にベクトル長が短くなるのは避けられない。そのためベクトル処理による十分な速度向上を得ることが難しくなっている。

(1)ループ長によるコードの選択

粒子の消滅に伴ってベクトル長が短くなっていくと，プログラムを高速処理するためのベクトル処理が，逆に処理を遅くしてしまう状況になる。Monte-4 の場合，ベクトル処理によって効果の得られるループ長は一般に５以上であり，４以下の場合はスカラ処理の方が効率がよい。そこで，短いループ長のベクトル処理による性能低下を軽減するために，ループ長にあわせて処理を行う。通常，DOループをベクトル処理，スカラ処理するかはプログラムのコンパイル時に決定されるが，次のようなコンパイル・オプションを指定することによって，ループ長が短いときはスカラ処理，長いときはベクトル処理を実行時に動的に決定して処理することが可能になる。

```
" -pvctl altcode=loopcnt "
```

Fig.4.6 に示すDOループをスカラ処理，ベクトル処理，オプション指定処理（以下，loopcnt 処理と呼ぶ）した場合の処理時間及び１粒子の平均処理時間をTable.4.1 に示す。N は演算量，T は処理時間，T/N は１粒子の平均処理時間を表す。ループ長４以下のときはベクトル処理がスカラ処理よりも効率が悪いことがわかる。loopcnt 処理によって，ループ長４以下での処理が効率良く行われている。ただし，ループ長５以上でのベクトル処理時間が多少遅くなってしまっている。これは動的に行われるスカラ処理とベクトル処理の選択にかかる時間が加わっているためと推測される。loopcnt 処理によってFig.4.6 のDOループは約８．２％の処理時間が短縮された。コード全体では約７％の処理時間が短縮された。

一方，loopcnt 処理を用いると，DOループ間でスカラ処理とベクトル処理の切り替えが頻繁に起こるようになり，キャッシュ・ミスヒットが増加し，コード全体としては十分な効果を得られないことがある。極端な場合，処理時間が増加してしまうこともあり，loopcnt 処理はキャッシュミス・ヒットの増加によるマイナス要因を考えても効果の得られる場合にのみ用いた方が良い。

(2)粒子補充処理

MORSE コードではヒストリ数を（１バッチ当りの粒子数）×（バッチ数）で表すが，バッチは遮蔽計算においてはあまり意味を持たず，統計誤差の推定が目的になっている。従来のベクトル版は1 バッチ当りの粒子をベクトル処理粒子数（一回にまとめて扱う粒子数）として，次のようなステップで粒子の追跡を行う。

（ｉ）粒子の発生

（ii）粒子バンクから粒子を取り出し

（ⅲ）Russian Roulette

（ⅳ）衝突の判定

（ⅴ）領域，寿命等のチェック

（ⅵ）衝突粒子の処理（エネルギー，方向余弦，重み等の変換）

（ⅰ）において発生させた粒子とスプリット等でできた粒子がすべて消滅するまで，（ⅱ）から（ⅵ），これをサイクルと呼ぶことにする，の処理が繰り返される。

ベクトル長を長くするために，まず1バッチ当りの粒子数を大きくすることが考えられるが，メモリ容量に制限がある，ベクトル処理粒子数を大きくすればするほど処理時間が短くなるわけではない，などの理由から適当な大きさにする必要がある。問題1の場合，ベクトル処理粒子数は１０，０００が適当であった。

１０，０００個の多量の粒子を処理しているのにもかかわらず，サイクルが進むにつれて粒子は消滅し，イベントバンクに空き領域が増えていくため，短いベクトル長で効率の悪い処理が多く行われている。これを改善するために，粒子の消滅によって生じた空き領域に粒子を補充して，次のヒストリの追跡をいち早く開始するようにした。Fig.4.7 に簡単な例を示す。１２個の粒子を(a) 3粒子×4バッチで処理した場合，(b) 補充処理を用いて１２粒子×1バッチで処理した場合の様子である。1サイクル目が終了すると，粒子P_1が消滅してイベントバンクに1つの空きが生じる。2サイクル目は(a) がP_2,P_3 の2粒子を処理しているのに対し，(b) は空きバンクに粒子P_4を補充して，P_4,P_2,P_3の3粒子を処理している。(a) が3粒子が完全に消滅してから次の3粒子の追跡を開始しているのに対し，(b) は3粒子が完全に消滅しなくても，消滅した分だけ粒子を補充して追跡を開始してしまう。この処理によって，各サイクルで処理される粒子数を多く保つことができる。また，総サイクル数も減らすことができる。例では(a) が１１サイクルを必要するのに対して，(b) は6サイクルで終了する。

粒子補充処理のためにいくつかのルーチンを改良した。まず，粒子発生方法を改良した。従来の方法では，バッチの最初に1バッチ当りの粒子をすべて発生させて「粒子バンク」に保存しておくため，1バッチ当りの粒子数を増やすと，「粒子バンク」の領域として必要なメモリ容量が増加する。そこで，粒子の発生処理を1サイクルの中に引き込み，粒子補充処理を行う時に「粒子バンク」の粒子が足りなければ，一定量の粒子を発生させることにした。さらに，粒子発生系のルーチンで用いられるコモンを変更した。粒子発生系処理ルーチンの"MSOUR-VLOOKZ-GGV"と粒子追跡系ルーチンの"NXTCOL-GOMST-G1V-GGV"は，どちらも粒子に依存した同じコモン変数を用いている。粒子補充処理では1バッチ中に複数回MSOUR が呼び出されるため，このままでは追跡途中の粒子の属性を持ったコモン変数が破壊されてしまい，正常に動作しない。これは粒子発生系ルーチンで用いるコモン変数を別に用意することで解決した。また，入力データの1バッチ当りの粒子数NSTRT ，バッチ数NBAT，粒子バンクの大きさNMOST の値を変更した。問題1 では粒子を１，０００，０００ヒストリを１０，０００×１００で処理していたので５００，０００×2に変更した。ここで１，０００，０００×１としなかったのはバッチ数は統計誤差を推定するために，バッチ数は2以上でなければならなかったためである。これに伴ってサブルーチンINPUT2を改良した。NSTRT は"I5"から"I7"，NMOST は"I5"から"I6"のデータが読み込めるようにした。

ベクトル処理の粒子補充処理によるコード全体に対する速度向上率は１．４倍，処理時間短縮は２６．４％である。

(3)処理の優先順序の変更

Fig.4.8 にサブルーチンNXTCOLの流れ図を示す。NXTCOLは粒子の次の衝突点を求めるルーチンであり，すべての粒子がescapeかcollision に分類されるまで処理が継続される。つまり，boundary-crossする粒子があるかぎり，サブルーチンGOMST が呼び出され，次の領域でのboundary-crossのチェックが行われる。GOMST で求められた状態フラグ変数MARKの値から，各粒子はboundary-cross,escape,collision に分類される。boundary-crossの回数は各粒子でまちまちで，GOMST において処理される粒子数は次第に減少していく。粒子のほとんどがescape，collisionしてしまっても，boundary-crossするものが１つでもあれば処理は続けられるため，ベクトル処理の効率は悪くなっていく。collision 粒子の処理はエネルギー，方向余弦，重み等をcollision 後のものに変換する。変換された粒子はあらためてNXTCOLが呼び出されてboundary-crossのチェックが行われる。当然，NXTCOLの呼び出し回数が増えていくにつれて処理粒子は減少していく。ベクトル処理の効率を上げるために，従来のboundary-crossする粒子がなくなるまでNXTCOLの処理を継続する方法，これを方法１と呼ぶことにする，にかえてboundary-crossのチェックが１回終了したら，collision 粒子の処理，粒子補充処理を行ってから次のboundary-crossチェックを行う方法，これを方法２と呼ぶことにする，を用いることにした。方法１がcolisionの回数の同じ粒子をまとめて処理しているのに対し，方法２はboundary-crossのチェックの回数が同じ粒子をまとめて処理している。方法２を用いることによって，１回のboundary-crossのチェックで処理される粒子数が増加し，回数（GOMST を呼び出す回数）が減少する。Fig.4.9 に簡単な例を示す。１０個の粒子を(a) 方法１と(b) 方法２を用いて追跡し，粒子補充処理は行わず，escape以外の粒子の消滅はないとする。粒子の状態を次のように表す。粒子をX_n-m（X は粒子の状態，n は粒子の番号，m はboundary-crossのチェックの回数）で表す。粒子の状態はS が粒子が発生してcollision していない状態，C1が１回collision ，C2が２回collision を表す。また，GOMST において行われるboundary-crossのチェックをcheck-m （m は回数）で表す。(a) では粒子S_2-0はcheck-1 でcollision し，check-2 からcheck-4 の間は処理が中断される。check-4 の後，collision 粒子の処理としてエネルギー，方向余弦等を変換してからcheck-5 でboundary-crossのチェックが再開される。(b) ではcheck-1 の後すぐにcollision 粒子の処理を行い，check-2 で他の粒子と一緒に処理を開始している。１０個の粒子が消滅するまでに(a) が８回のboundary-crossのチェックを行っているのに対し，(b) では５回しか行われない。また，１回のboundary-crossのチェックの平均処理粒子数は(a) で約３．２個，(b) で約５．２個である。このように(b) はベクトル処理が効率良く行われるようになる。Table.4.2 に２つの方法を用いた場合のGOMST,G1V,GGVの呼び出し回数，平均ベクトル長を示す。方法２を用いた場合，各ルーチンの呼び出し回数が減少し，平均ベクトル長が改善されているのがわかる。特に最もコストの高いGGV の呼び出し回数が大幅に減少している。また，コード全体では平均ベクトル長は４８．２から５２．９に改善された。コード全体の速度向上率は１．１倍であり，処理時間短縮は１０．２％である。

この値はメモリ競合による性能低下を取り除くために，並列版を１プロセッサ・４タスクで測定した。方法２を実現するためにNXTCOLをランダムウォーク・コントロール・ルーチンBATCHに展開し，boundary-crossのチェックとcollision 粒子の処理等の優先順序を変更した。

4.5 モンテカルロ・パイプラインの適用

Monte-4 には３種類のモンテカルロ・パイプラインが用意されている。今回の作業では幾何形状パイプライン及び事象分類パイプラインを適用した。並列版（１プロセッサ・４タスク）の場合，これによるコード全体に対する速度向上率は１．２倍，処理時間短縮は１７．５％である。

(1)幾何形状パイプライン

粒子の属する領域形状によって粒子を分類するための多分岐の条件分岐文を含むDOループを高速処理する。Fig.4.10(a) の多分岐の条件分岐文は粒子を１２の形状もしくはエラー処理に分類している。したがって，Fig.4.10(b) のように１２分岐の幾何形状パイプラインの適用することができる。しかし，このままでは4010ループのIF文の処理に時間がかかってしまい，あまり効果が得られなかった。そこでRCC とTRC を別バンクに分類し，後からTRC のリストベクトルをRCC のリストベクトルに追加する方法をとり，Fig.4.10(c) のように１３分岐の幾何形状パイプラインを適用した。(c) は(b) にくらべ4010ループのIF文削除された分だけ高速処理される。

(2)事象分類パイプライン

粒子を事象によって高速に分類し，粒子バンクを作成する。二分岐の条件分岐文を含むDOループはベクトル処理可能であるが，この処理は頻繁に行われるため高速化が必要である。

Fig.4.11のように二分岐の条件分岐分を含むDOループは事象分類パイプラインが適用することができる。

4.6 その他の最適化手法

我々は最高のオブジェクト・コード，つまり最も高速に処理を行うオブジェクト・コード，を生成することをコンパイラの最適化に期待している。しかし，より最適なコードを強く求めた場合，コンパイラだけでは足りず，チューニングや手作業でのコーディングによってさらに最適化される部分が存在する。不要なロード／ストア命令やキャッシュミスヒットは，次に挙げるようなチューニングによって削減され，より最適なコードを生成することができる。

・わかりやすいプログラミング

一般にプログラムは誰が見ても論理の流れが良くわかるように書くと，コンパイラは最適なオブジェクトを生成しやすい。例えばFig.4.12(a) のように，ｉ）GOTO文はIF－THEN－ELSEの形に書き換える，ii）変数の定義と参照はできるだけ近い位置にする，などの書き換えによって命令数が削減され，プログラムの効率が良くなり，処理時間が多少短縮されることがある。

(Fig.4.12(a) 参照) ii) によってベクトルロード命令が減り，効率が良くなる。定義と参照が離れていると，ベクトルレジスタにロードされているデータが他のデータなどによって消されてしまい，参照するときに再びベクトルロードが必要になってしまうためである。

・ループ融合

Fig.4.12(b) のように同一リストを使用しているループを1つにまとめることによって，DOループのたちあげに要する時間とロード命令を減らすことができる。

・リストベクトルの出現頻度の削減

Fig.4.12(c) のようにDOループ内に複数回現れるリストベクトルについて，スカラ変数による共用化を行うことによって，ロード命令を減らすことができる。

・ループ・アンローリング

ループ・アンローリングすることにより，ループ内の演算量を増やすことができ，さらにロード／ストア命令を減らすことができる。Fig.4.12(d) のような二重ループを内側ループ内で展開した。

Table.4.1 CPU time in sample DO loop

(a) scalar

ループ長（L）	L＜5	L≧5	Total
演算回数（N）	349,279	18,522,225	18,871,504
演算時間（T）	0.65 sec	32.4 sec	33.0 sec
T／N	1.8 micro sec	1.7 micro sec	1.7 micro sec

(b) vector

ループ長（L）	L＜5	L≧5	Total
演算回数（N）	349,279	18,522,225	18,871,504
演算時間（T）	1.59 sec	5.21 sec	6.81 sec
T／N	4.5 micro sec	0.28 micro sec	0.36 micro sec

(c) vector, altcode=loopcnt

ループ長（L）	L＜5	L≧5	Total
演算回数（N）	349,279	18,522,225	18,871,504
演算時間（T）	0.69 sec	5.56 sec	6.25 sec
T／N	1.9 micro sec	0.30 micro sec	0.33 micro sec

Table.4.2 Frequency and Average of vector length in subroutine GOMST,G1V,GG

	(a) method-1			(b) method-2		
SUBROUTINE	GOMST	G1V	GGV	GOMST	G1V	GGV
FREQUENCY	3256	3256	690028	606	606	270209
AVE.V.LEN	61.8	47.7	46.8	63.4	55.1	51.3

SUBROUTINE GGV(LOCAT_, MA_, FPD_, -------, NGGBK, NGGN, NP)

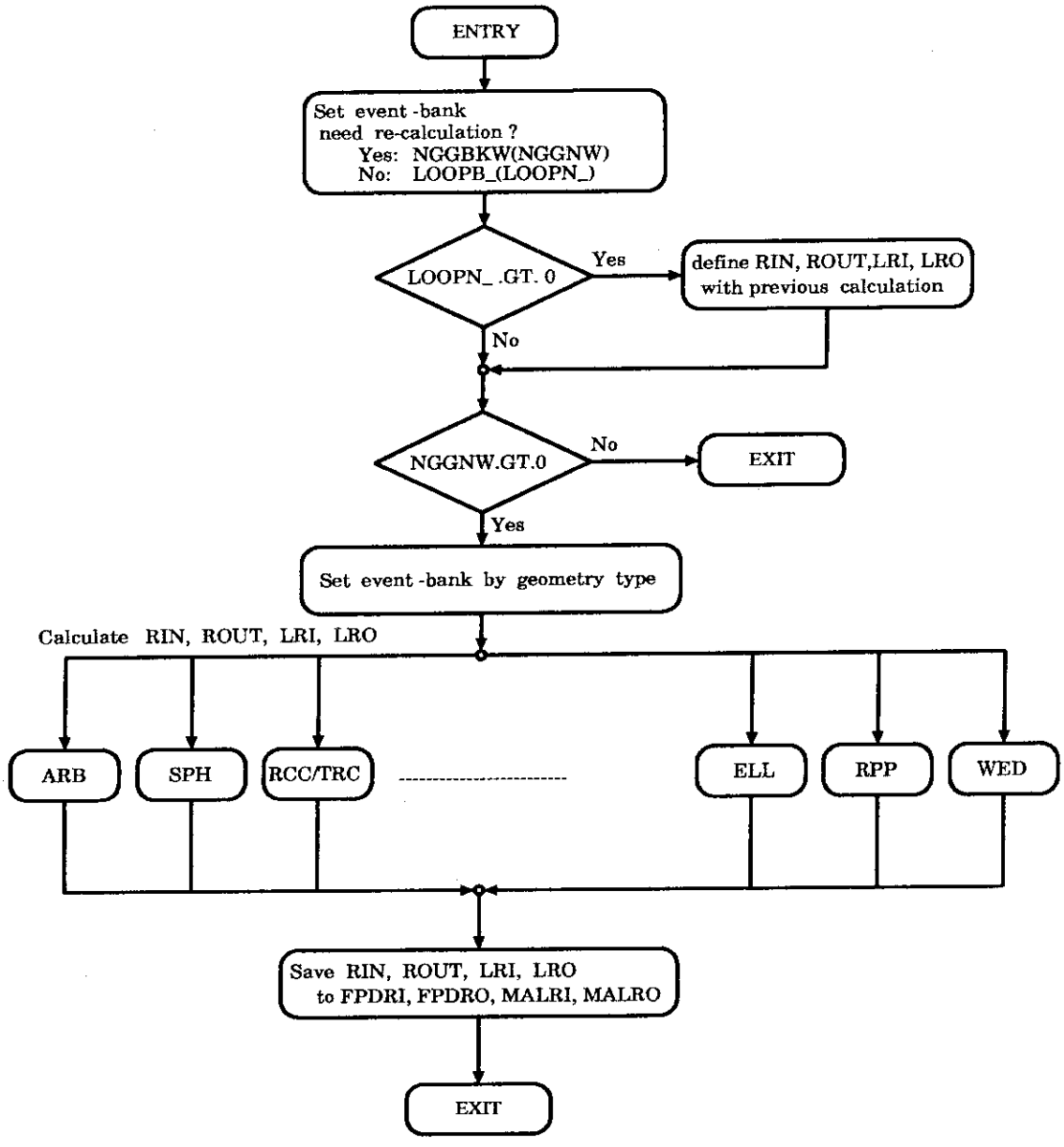


Fig.4.1 Flow diagram of vectorized subroutine GGV

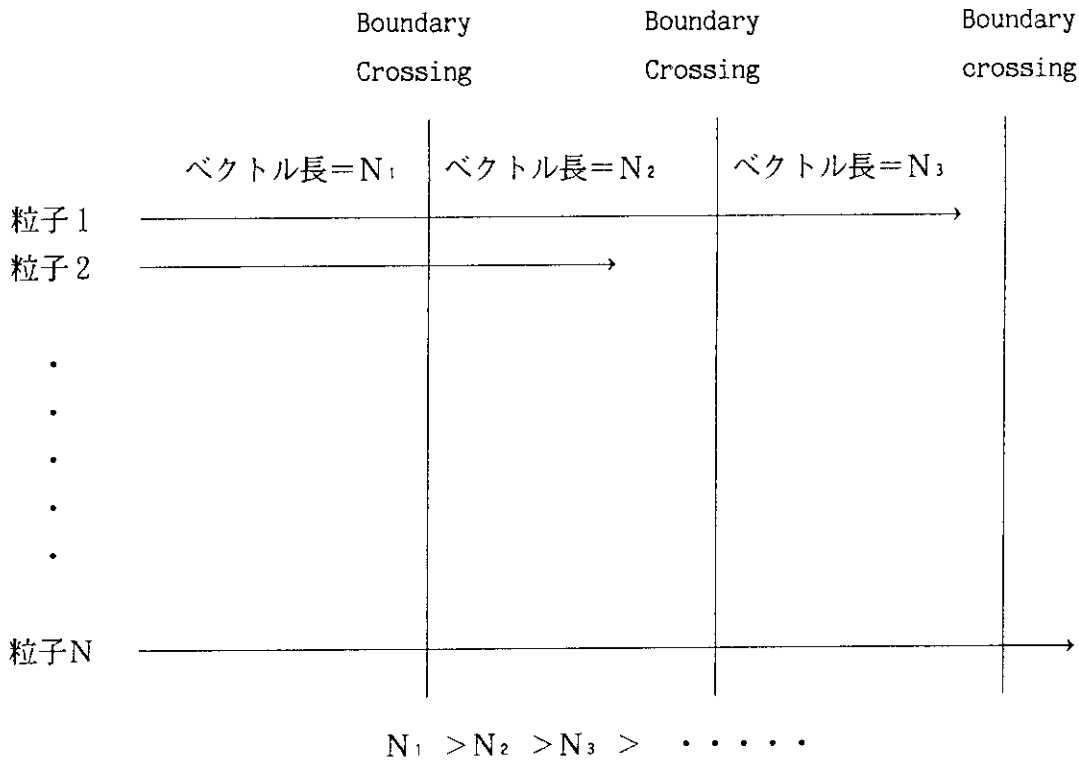


Fig.4.2 Vector length in subroutine GGV

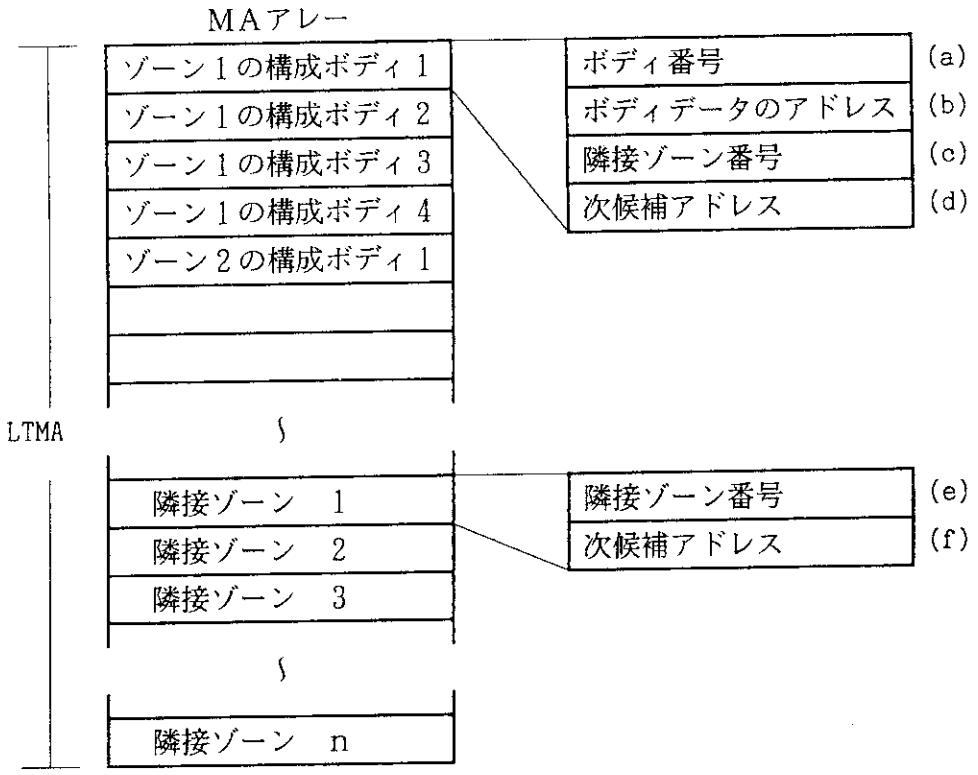
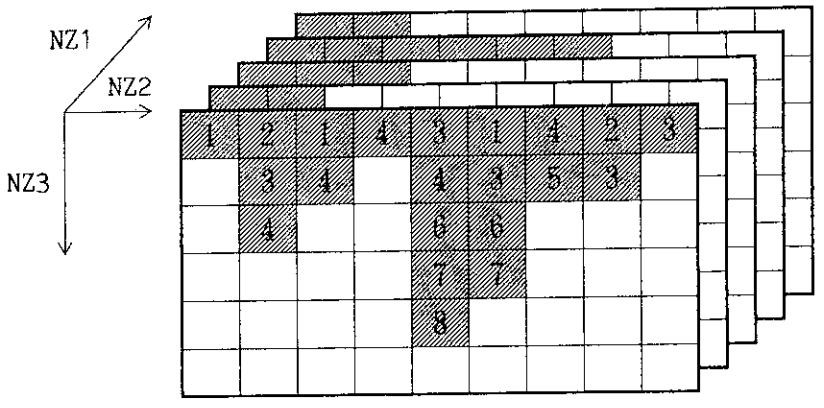


Fig.4.3 Structure of original next-zone-list



COMMON/NXTLIST/NXTBO(NZ1,NZ2,NZ3),NXTNO(NZ1,NZ2)

- NXTBO(NZ1,NZ2,NZ3) :隣接ゾーン番号
- NXTNO(NZ1,NZ2) :隣接ゾーンの数

NZ1 : 現在属するゾーンの番号

NZ2 : ゾーンを構成しているボディの番号

NZ3 : 隣接ゾーンリストの番号

Fig.4.4 Structure of vectorized next-zone-list

```
      SUBROUTINE  VINTERP(IV,KEY)
      LOCAL COMMON /VNUTRON/UU(100),VV(100),WW(100)
      LOCAL COMMON /NUTRON/U,V,W
      IF (KEY.EQ.0) THEN
         UU(IV) = U
         VV(IV) = V
         WW(IV) = W
      ELSE
         U      = UU(IV)
         V      = VV(IV)
         W      = WW(IV)
      ENDIF
      RETURN
```

Fig.4.5 Examples of interface routine

```
                *vdir nodep
                      DO 1451 IV=1,N400__
C-------------->         KFLAG(IV) = 0
!                        JV = J400B__(IV)
!                        DIS  = DIST__(JV)
!                        RODI = ROUT__(JV) - DIS
!                        RIDI = RIN__(JV)  - DIS
!                        EPDI = EPS*DIS
!                        IF(NBO__(JV).GT.0) GO TO 330
!                 320    IF(RODI.GT.EPDI.AND.RIDI.LE.0) KFLAG(IV) = 1
!                        GO TO 1420
!                 330    IF(RIDI.GT.EPDI.OR.RODI.LE.0)  KFLAG(IV) = 1
!                1420 CONTINUE
!                      IF (KFLAG(IV).NE.1) THEN
!                         N(JV) = N(JV) + 4
!                         IF (N(JV).LE.NUM(JV)) THEN
!                            KFLAG(UV) = 2
!                         ELSE
!                            KFLAG(UV) = 3
!                         ENDIF
C--------------- 1451 CONTINUE
```

Fig.4.6 Sample DO loop which altcode-option was used

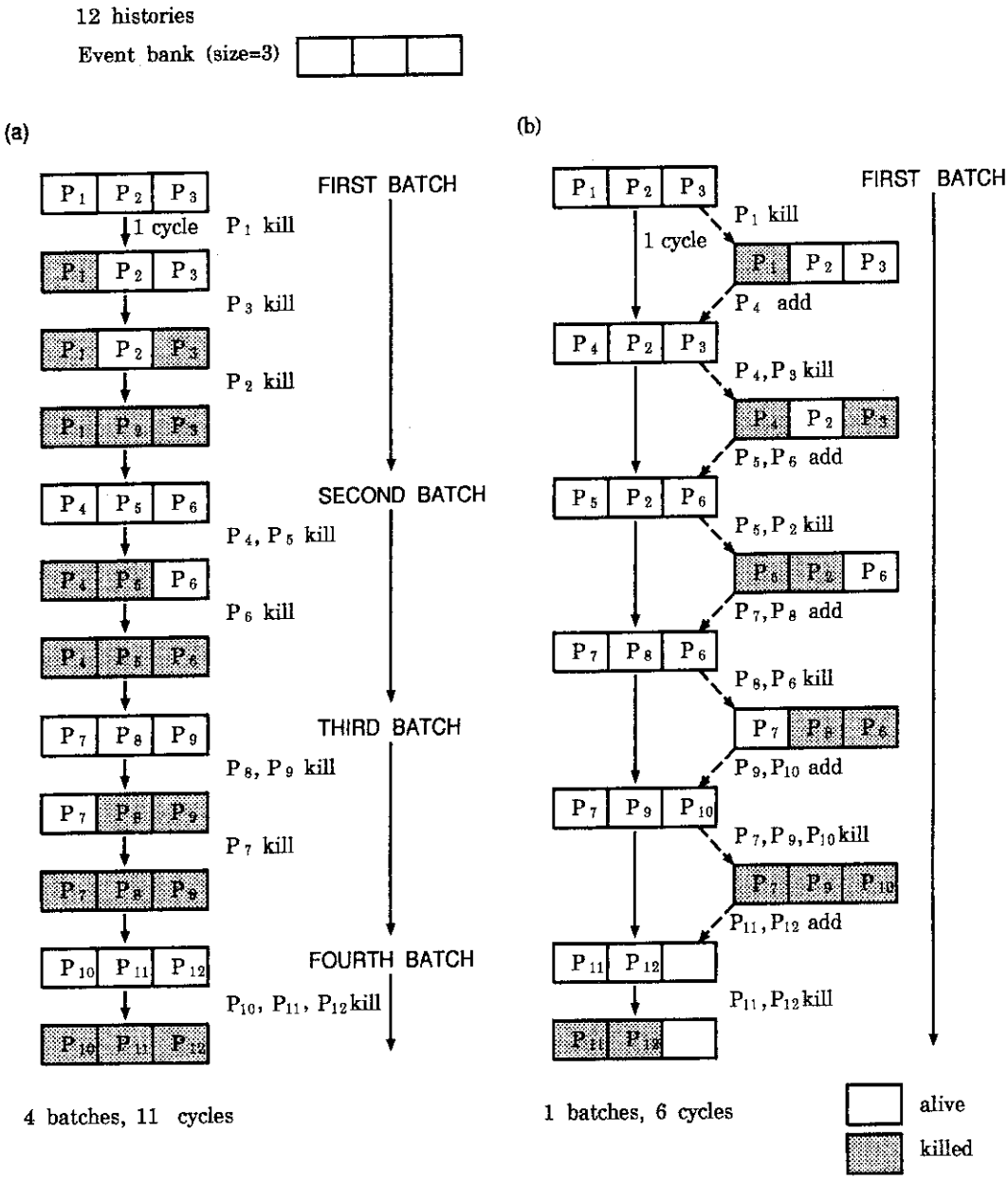


Fig.4.7 Control of particles tracked

SUBROUTINE NXTCOL(N100B_, J100_,IXR,R_,LOCR, NP)

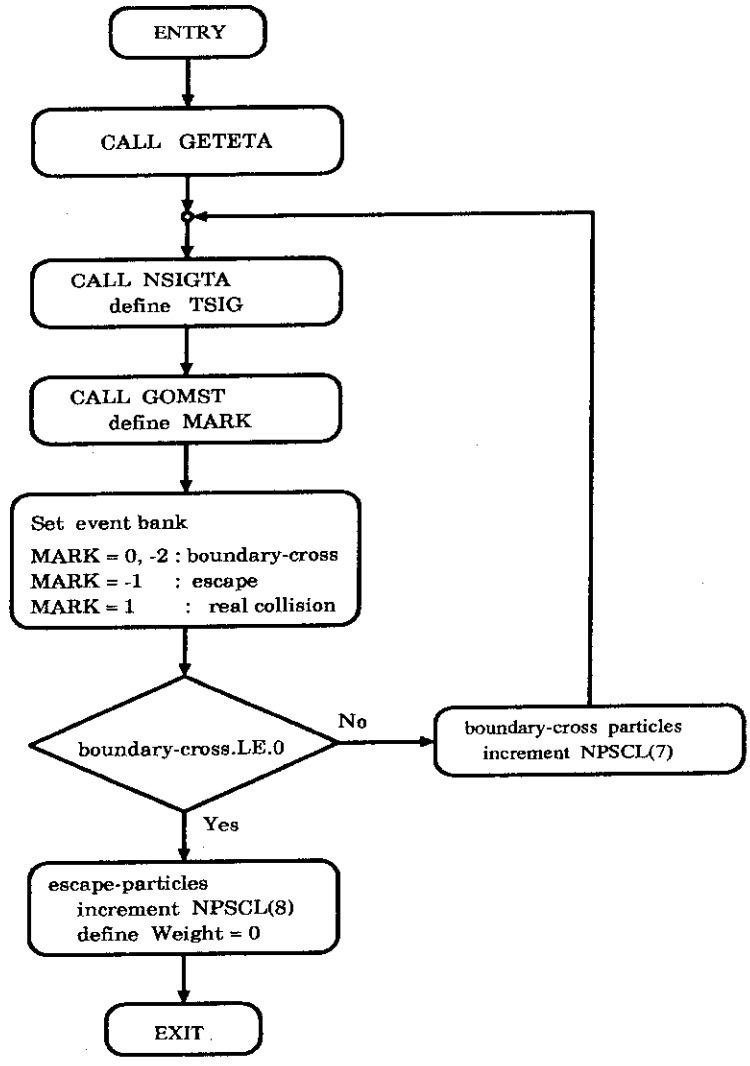


Fig.4.8 Flow diagram of vectorized subroutine NXTCOL

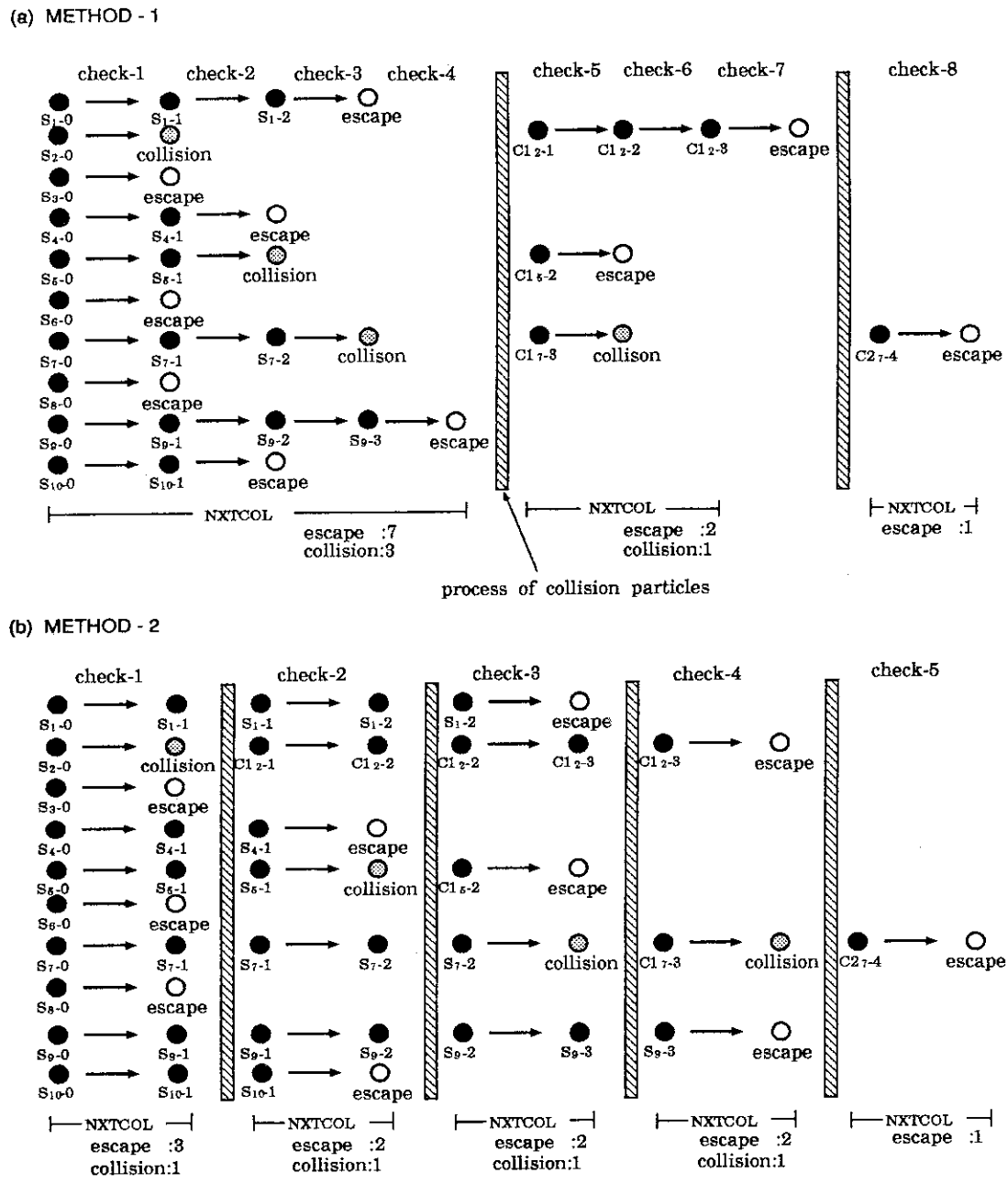


Fig.4.9 Methods to check boundary crossing

```
(a) Geometry branch ( not pipeline-processed )
                *vdir novector
                      DO 4010 IV=1,NGGNW
  4---------->        !  JV       = NGGBKW(IV)
  !                   !  IF( (ITYPE__(JV).LE.0).OR.(ITYPE__(JV).GT.12) ) THEN
  !                   !  !  WRITE(IOUT,2012) ITYPE__(JV),IR__(JV),NBO__(JV)
  !                   !  !  CALL PR(1)
  !                   !  !  CALL ERROR
  !                   !  !  GO TO 4010
  !                   !  END IF
  !                   !  GO TO (1101,1201,1401,1301,1401,1601,1701,1801,1711,
  !                  +!         1211,1212,1213) ,ITYPE__(JV)
  !                   !  "      ARB  SPH  RCC  REC  TRC  ELL  BOX  WED  RPP
  !                   !  "      GEL  TOR  QUA
  !                   !  GO TO 4010
  !               1101 !  N1100__          = N1100__ + 1
  !                   !  J1100__(N1100__) = JV
  !                   !  GO TO 4010
  !               1201 !  N1200__          = N1200__ + 1
  !                   !  J1200__(N1200__) = JV
  !                   !  GO TO 4010
  !               1401 !  N1400__          = N1400__ + 1
  !                   !  J1400__(N1400__) = JV
  !                   !  GO TO 4010
  !               1301 !  N1300__          = N1300__ + 1
  !                   !  J1300__(N1300__) = JV
  !                   !  GO TO 4010
  !               1601 !  N1600__          = N1600__ + 1
  !                   !  J1600__(N1600__) = JV
  !                   !  GO TO 4010
  !               1701 !  N1700__          = N1700__ + 1
  !                   !  J1700__(N1700__) = JV
  !                   !  GO TO 4010
  !               1801 !  N1800__          = N1800__ + 1
  !                   !  J1800__(N1800__) = JV
  !                   !  GO TO 4010
  !               1711 !  N1710__          = N1710__ + 1
  !                   !  J1710__(N1710__) = JV
  !                   !  GO TO 4010
  !               1211 !  N1201__          = N1201__ + 1
  !                   !  J1201__(N1201__) = JV
  !                   !  GO TO 4010
  !               1212 !  N1202__          = N1202__ + 1
  !                   !  J1202__(N1202__) = JV
  !                   !  GO TO 4010
  !               1213 !  N1203__          = N1203__ + 1
  !                   !  J1203__(N1203__) = JV
  4-------------  4010 CONTINUE
```

Fig.4.10 Application of Geometric Pipeline to MORSE code (continue)

(b) VGSORT12 (ordinary pipeline-processed)

```
                      NWK1 = 0
                      NERROR = 0
                      *vdir nodep
                      DO 4010 IV = 1,NGGNW
C------------>        !  JV      = NGGBKW(IV)
!                     !  ITY = ITYPE__(JV)
!                     !  IF ( ITY.EQ.1 ) THEN
!                     !  !  NVA1(IV) = 1
!                     !  ELSE IF ( ITY.EQ.2 ) THEN
!                     !  !  NVA1(IV) = 2
!                     !  ELSE IF ( (ITY.EQ.3).OR.(ITY.EQ.5)) THEN
!                     !  !  NVA1(IV) = 3
!                     !  ELSE IF ( ITY.EQ.4 ) THEN
!                     !  !  NVA1(IV) = 4
!                     !  ELSE IF ( ITY.EQ.6 ) THEN
!                     !  !  NVA1(IV) = 5
!                     !  ELSE IF ( ITY.EQ.7 ) THEN
!                     !  !  NVA1(IV) = 6
!                     !  ELSE IF ( ITY.EQ.8 ) THEN
!                     !  !  NVA1(IV) = 7
!                     !  ELSE IF ( ITY.EQ.9 ) THEN
!                     !  !  NVA1(IV) = 8
!                     !  ELSE IF ( ITY.EQ.10 ) THEN
!                     !  !  NVA1(IV) = 9
!                     !  ELSE IF ( ITY.EQ.11 ) THEN
!                     !  !  NVA1(IV) = 10
!                     !  ELSE IF ( ITY.EQ.12 ) THEN
!                     !  !  NVA1(IV) = 11
!                     !  ELSE
!                     !  !  NVA1(IV) = 12
!                     !  ENDIF
C------------  4010 CONTINUE
                      CALL VGSORT12(NVA1,NGGBKW,NGGNW,12,
                     +              J1100__,N1100__,J1200__,N1200__,J1400__,N1400__,
                     +              J1300__,N1300__,J1600__,N1600__,J1700__,N1700__,
                     +              J1800__,N1800__,J1710__,N1710__,J1201__,N1201__,
                     +              J1202__,N1202__,J1203__,N1203__,NERROR,JERRB)
```

Fig.4.10 Application of Geometric Pipeline to MORSE code (continue)

```
(c) VGSORT13 ( improved pipeline-processed )
                    *vdir nodep
                    DO 4010 IV = 1,NGGNW
C----------->       !  JV      = NGGBKW(IV)
!                   !  NVA1(IV)= ITYPE__(JV)
C------------  4010 CONTINUE
                    CALL VGSORT13(NVA1,NGGBKW,NGGNW,13,
                   +              J1100__,N1100__,J1200__,N1200__,J1400__,N1400__,
                   +              J1300__,N1300__,J1500__,N1500__,J1600__,N1600__,
                   +              J1700__,N1700__,J1800__,N1800__,J1710__,N1710__,
                   +              J1201__,N1201__,J1202__,N1202__,J1203__,N1203__,
                   +              NERROR,JERRB)
                    IF(N500__.GT.0) THEN
                    DO 4011 IV = 1,N1500__
C----------->       !  JV      = J1500__(IV)
!                      J1400__(N1400__+IV) = J1500__(IV)
C------------  4010 CONTINUE

                    N1400__ = N1400__ + n1500__
```

Fig.4.10 Application of Geometric Pipeline to MORSE code (continued)

```
                    NWK1=0
                    NWK3=0
                    *vdir nodep
C------------       DO 1313 IV=1,N300__
!                   !  JV=J300B__(IV)
!                   !  IF(LFLAG(JV).NE.0) THEN
!                   !  !  NWK1=NWK1+1
!                   !  !  JWK11(NWK1)=JV
!                   !  ELSE
!                   !  !  NWK3=NWK3+1
!                   !  !  JWK33(NWK3)=JV
!                   !  ENDIF
C------------  1313 CONTINUE

                                    ⇩

                    CALL VESORT(N300__,J300B__,LFLAG,NWK1,JWK11,NWK3,JWK33)
```

Fig.4.11 Application of Event pipeline in MORSE code

```
(a)                     *vdir nodep
                        DO 10 IV=1,NG1
  C------------>        !  JV=G1B(IV)
  !                     !  SP(JV)=0.
  !                     !  IRP(JV) = IRPRI__(JV)
  !                     !  IF(NASC__(JV).GT.0) GO TO 110
  !                     !  KLOOP__(JV)=KLOOP__(JV)+1
  !                     !  DIST__(JV)=0.
  !                 110 !  SMIN(JV)=PINF
  !                     !  ISAVE(JV)=0
  !                     !  ISAVE2(JV)=0
  !                     !  NK(JV) = 1
  !                 130 !  CONTINUE
  !                     !  IRR(JV) = IRP(JV)
  !                     !  N(JV)=LOCREG(IRR(JV)) +1
  !                     !  NUM(JV)=NUMBOD(IRR(JV))*4+N(JV)-4
  !                     !  J300B__(IV)=JV
  !                     !  G1BW(IV)=JV
  C------------      10 CONTINUE
                                    ⇩
                        *vdir nodep
                        DO 10 IV=1,NG1
  C------------>        !  JV=G1B(IV)
  !                     !  IF(NASC__(JV).LE.0) THEN
  !                     !  !  KLOOP__(JV)=KLOOP__(JV)+1
  !                     !  !  DIST__(JV)=0.
  !                     !  ENDIF
  !                 110 !  SMIN(JV)  =PINF
  !                     !  ISAVE(JV) =0
  !                     !  ISAVE2(JV)=0
  !                     !  NK(JV)  = 1
  !                     !  SP(JV)  = 0.
  !                     !  IRTMP   = IRPRI__(JV)
  !                     !  IRP(JV) = IRTMP
  !                     !  IRR(JV) = IRTMP
  !                     !  N(JV)   = LOCREG(IRTMP) +1
  !                     !  NUM(JV) = NUMBOD(IRTMP)*4+N(JV)-4
  !                     !  J300B__(IV) = JV
  !                     !  G1BW(IV)   = JV
  C------------      10 CONTINUE
```

Fig.4.12 Examples of optimization for performance tuning (continue)

```
(b)              *vdir nodep
  C------------>       DO 4401 IV=1,N1400__
  !                       JV     = J1400__(IV)
  !                       K      = K__(JV)
  !                       RB__(JV)= FPD(K+8)
  !                                 ⟆
  !
  !                       HH__(JV) = HH
  !                       RTRB__(JV) = RTRB
  C------------  4401 CONTINUE
                       KFLAG = 0
                 *vdir nodep
  C------------>       DO 4412 IV=1,N1400__
  !                       JV     = J1400__(IV)
  !                       IF(DABS(DEN__(JV).LE.1.0D-6) KFLAG=KFLAG+1
  C------------  4412 CONTINUE
                                    ⇩
                       KFLAG = 0
                 *vdir nodep
  C------------>       DO 4401 IV=1,N1400__
  !                       JV     = J1400__(IV)
  !                       K      = K__(JV)
  !                       RB__(JV)= FPD(K+8)
  !                                 ⟆
  !
  !                       HH__(JV) = HH
  !                       RTRB__(JV) = RTRB
  !                       IF(DABS(DEN__(JV).LE.1.0D-6) KFLAG=KFLAG+1
  C------------  4412 CONTINUE

(c)                  DO 4600 IV=1,N1600
  C------------>        JV = J1600(IV)
  !                     A1 = (DX1*WB(JV,1) + DX2*WB(JV,2) + DX3*WB(JV,3))*2.0
  !                     A2 = (DX4*WB(JV,1) + DX5*WB(JV,2) + DX6*WB(JV,3))*2.0
  C------------ 4600 CONTINUE
                                    ⇩
                       DO 4600 IV=1,N1600
  C------------>        JV = J1600(IV)
  !                     WB1 = WB(JV,1)
  !                     WB2 = WB(JV,2)
  !                     WB3 = WB(JV,3)
  !                     A1 = (DX1*WB1 + DX2*WB2 + DX3*WB3)*2.0
  !                     A2 = (DX4*WB1 + DX5*WB2 + DX6*WB3)*2.0
  C------------ 4600 CONTINUE
```

Fig.4.12 Examples of optimization for performance tuning (continue)

```
(d)                  DO 4705 I=1,3
  1------------>        I1 = 3*I + 2
  !                     I2 = 3*I + 3
  !                     I3 = 3*I + 4
  !            *vdir nodep
  !                  DO 4700 IV=1,N1700__
  ! C---------->        IF(IFLAG(IV).EQ.1) GO TO 4700
  ! !                   LFLAG__(IV) = 0
  ! !                   KK(IV)      = 0
  ! !                   JV          = J1700__(IV)
  ! !                   K           = K__(JV)
  ! !                   DX1         = DX1__(JV)
  ! !                   DX2         = DX2__(JV)
  ! !                   DX3         = DX3__(JV)
  ! !                   FPD1        = FPD(K+I1)
  ! !                   FPD2        = FPD(K+I2)
  ! !                   FPD3        = FPD(K+I3)
                                .
                                .
                                .
                                ⇩
            *vdir nodep
                  DO 4700 IV=1,N1700__
C------------->        LFLAG__(IV) = 0
!                       KK(IV)      = 0
!                       JV          = J1700__(IV)
!                       K           = K__(JV)
!                       DX1         = DX1__(JV)
!                       DX2         = DX2__(JV)
!                       DX3         = DX3__(JV)
!           C I=1,I1=5,I2=6,I3=7
!                       IF(IFLAG(IV).EQ.1) GO TO 4700
!                       LFLAG__(IV) = 0
!                       KK(IV)      = 0
!                       JV          = J1700__(IV)
!                       K           = K__(JV)
!                       FPD1        = FPD(K+5)
!                       FPD2        = FPD(K+6)
!                       FPD3        = FPD(K+7)
                                .
                                .
                                .
```

Fig.4.12 Examples of optimization for performance tuning (continued)

5．性能評価結果

Table.5.1 に各問題におけるMORSE コードの速度向上率を示す。高速化のチューニングは問題１を中心に行い，問題２及び問題３に対しては，作業量等の問題で，きめ細かなチューニングはせずに性能評価を行った。問題１と問題２で９倍を越える速度向上率を得た。ただし，問題２は4.6(3)の(b) のチューニングを適用せずに測定した結果である。問題３は７．８倍の速度向上率しか得られなかった。Table.5.3 に各問題の並列版MORSE コードのコスト分布を示す。各問題ともオリジナル・スカラ版のコスト分布と同様，サブルーチンGGV,G1V に計算コストが集中している。GGV,G1V は，粒子の次の衝突点を求めるためにサブルーチンGOMST から呼ばれる場合と，粒子の現在位置と各検出器間の距離を求めるためにサブルーチンEUCLIDから呼ばれる場合がある。4.6(3)の(b) のチューニングはGOMST の呼び出し回数を減らすことができるが，EUCLIDの呼び出し回数は増やしてしまう。そのため，EUCLID以下のコストが高い問題２では処理時間が逆に増加してしまった。

Table.5.1 の速度向上率（A）の１プロセッサ・ベクトル処理の値，速度向上率（B）の４プロセッサ・ベクトル処理の値は，それぞれベクトル化による速度向上率，並列化による速度向上率と考えられる。問題１ではベクトル化によって３．４倍の速度向上率を得た。１プロセッサ処理（ベクトル版）におけるベクトル化率は９４．３％，加速率は７．０倍である。一方，並列化による効果が２．６倍しか得られなかった。今回の問題は遮蔽問題であることから，バッチ数を少なくし，１バッチ当りの粒子数を多くすることによって，粒度は十分に大きくすることができた。Fig.5.1 は１プロセッサ・４タスク処理（子タスクを生成せずに４タスクを親タスクで順番に処理する）における各タスクの処理時間を示している。これをみると１バッチ目は１５．７～１５．９秒，２バッチ目は１２．９～１３．０秒で処理されており，負荷分散も非常によいことがわかる。Fig.5.2 にMORSE コードをベクトル化，並列化したときの処理時間の変化を示す。これから粒子分割とメモリ競合によって性能が低下していることがわかる。スカラ処理ならば粒子分割したとしても両者に演算量の違いはないが，Monte-4 のようなベクトル処理の場合は，粒子分割することによって演算量が多少増加してしまう。Fig.5.1 から４並列処理時の理想値は２９．９秒と推定できる。並列処理のために使用される関数FORK，JOIN等のオーバヘッドは非常に小さく，無視できるので，理想値と実測値の差はメモリ競合による性能低下といえる。実測値４０．７秒であるからメモリ競合によって２７％の性能低下が起きている考えられる。

問題２ではベクトル化による速度向上率は低いが，並列化による速度向上率が高く，９．９倍の速度向上率を得た。Table.5.2 を見てわかるように粒子分割によって逆に処理時間が減少している。boudary-cross,collison等の履歴を両者で比較すると，４並列のほうがcollisionの回数が少なくなっている。このような現象は乱数を用いているために起きたと考えられる。

問題３ではベクトル化による速度向上率が３倍を越えているが，並列化による速度向上率が低いため，７．８倍の速度向上率しか得られなかった。並列化による速度向上率は粒子分割によって１０％，メモリ競合によって３０％の性能低下が起きていると考えられる。

Table.5.1 Speed up Ratios of MORSE code on Monte-4

(a) 問題1

プロセッサ数	1		2	3	4
処理モード	ｽｶﾗｰ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理
処理時間	374.0sec	107.6	60.6	47.7	40.7
速度向上率（A）	1.0	3.4	6.1	7.8	9.1
速度向上率（B）	———	1.0	1.7	2.2	2.6

(b) 問題2

プロセッサ数	1		2	3	4
処理モード	ｽｶﾗｰ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理
処理時間	328.5sec	118.3	54.1	40.0	33.0
速度向上率（A）	1.0	2.7	6.0	8.2	9.9
速度向上率（B）	———	1.0	2.1	2.9	3.5

(c) 問題3

プロセッサ数	1		2	3	4
処理モード	ｽｶﾗｰ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理	ﾍﾞｸﾄﾙ処理
処理時間	99.8sec	31.6	———	———	12.7
速度向上率（A）	1.0	3.1	———	———	7.8
速度向上率（B）	———	1.0	———	———	2.4

注1）compiler:FORTRAN77/M4 Rev.067 Path No.001-017,019-020

注2）1プロセッサ・スカラ処理は，オリジナル・コードのスカラー処理時間を示す。

注3）1プロセッサ・ベクトル処理は，ベクトル版コードのベクトル処理時間示す。

注4）速度向上率Aは，オリジナル・コードのスカラー処理時間に対する速度向上率を，同じくBは，1プロセッサ・ベクトル処理時間に対する速度向上率を示す。

注5）問題3でプロセッサ数2，3で処理するとcore dump してしまうため，測定できなかった。

注6）プロセッサ数とタスク数は同じである。

Table.5.2 Performance in parallel processing of MORSE code

(a) 問題1

タスク数	1	2	3	4
シリアル実行	107.6 sec	109.5	113.0	116.8
並列実行	——	60.6 sec	47.7	40.7
シリアル／並列	——	1.8	2.3	2.8

(b) 問題2

タスク数	1	2	3	4
シリアル実行	118.3 sec	103.4	100.9	104.6
並列実行	——	54.1 sec	40.0	33.0
シリアル／並列	——	1.9	2.5	3.1

(c) 問題3

タスク数	1	2	3	4
シリアル実行	31.6 sec	36.1	35.3	35.9
並列実行	——	——	——	12.7
シリアル／並列	——	——	——	2.8

注1）粒子分割によってできるタスクを1プロセッサを用いて逐次処理することをシリアル実行といい，2，3または4プロセッサを用いて同時処理することを並列処理という。

注2）タスク1はベクトル版，タスク2～タスク4は並列版で測定した。

Table.5.3(a) Summary list of behavior analysis of morse code parallel version (probrem.1)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GGV	SUB		270209	98.729(65.9)	98.729(65.9)	320.1	57.4	97.80	51.3	2.443(2)	22.795(15)
G1V	SUB		606	120.346(80.3)	34.127(22.8)	250.8	12.2	97.67	55.1	1.305(1)	6.093(4)
BATCH	SUB		8	148.133(98.8)	3.302(2.2)	202.9	8.9	93.18	62.9	0.106(0)	0.702(0)
VLOOKZP	SUB		400	15.404(10.3)	2.894(1.9)	394.1	24.7	95.40	62.3	0.000(0)	0.129(0)
GETETA	SUB		587	2.349(1.6)	2.349(1.6)	260.5	156.5	87.90	63.8	0.005(0)	0.063(0)
GOMST	SUB		606	122.591(81.8)	2.246(1.5)	375.7	17.2	99.30	63.4	0.144(0)	0.011(0)
VTRACE	SUB		1714	1.972(1.3)	1.972(1.3)	137.3	54.8	82.39	62.8	0.018(0)	1.156(1)
COLISN	SUB		586	1.168(0.8)	1.168(0.8)	273.3	84.0	87.44	62.0	0.032(0)	0.196(0)
TESTW	SUB		598	0.888(0.6)	0.888(0.6)	154.1	6.7	84.08	62.4	0.018(0)	0.215(0)
MSOURP	SUB		104	15.795(10.5)	0.389(0.3)	592.0	226.7	97.02	62.6	0.000(0)	0.009(0)
Q	FUNC		93604	0.341(0.2)	0.341(0.2)	29.6	2.0	0.00	0.0	0.000(0)	0.186(0)
FIND	SUB		791	0.626(0.4)	0.294(0.2)	25.0	1.2	0.00	0.0	0.000(0)	0.139(0)
READSG	SUB		72	0.275(0.2)	0.275(0.2)	736.9	0.0	94.44	64.0	0.000(0)	0.012(0)
NRUN	SUB		1	0.226(0.2)	0.226(0.2)	25.5	0.2	0.88	34.5	0.000(0)	0.062(0)
JNPUT	SUB		1	1.135(0.8)	0.191(0.1)	71.5	1.1	0.65	40.6	0.000(0)	0.019(0)
JOMIN	SUB		1	0.289(0.2)	0.189(0.1)	31.4	0.2	0.02	38.8	0.000(0)	0.042(0)
GENI	SUB		1	0.095(0.1)	0.094(0.1)	38.7	0.2	0.54	29.6	0.000(0)	0.015(0)
OUTPT2	SUB		1	0.076(0.1)	0.076(0.1)	30.5	0.1	0.25	15.7	0.000(0)	0.016(0)
BANKRV	SUB		2120	2.236(1.5)	0.037(0.0)	7.9	0.0	0.09	44.7	0.000(0)	0.026(0)
OUTP1	SUB		104	0.033(0.0)	0.033(0.0)	1346.1	514.9	99.38	63.7	0.000(0)	0.000(0)
DTLIST	SUB		1	0.020(0.0)	0.020(0.0)	27.5	0.0	0.10	35.9	0.000(0)	0.005(0)
ANGLES	SUB		400	0.654(0.4)	0.015(0.0)	27.9	0.5	31.92	40.4	0.000(0)	0.007(0)
INPUT1	SUB		1	0.299(0.2)	0.009(0.0)	38.9	0.1	22.56	63.2	0.000(0)	0.002(0)
LEGEND	SUB		400	0.007(0.0)	0.007(0.0)	81.8	9.9	74.41	10.5	0.000(0)	0.003(0)
STORE	SUB		72	0.007(0.0)	0.007(0.0)	83.3	0.0	1.67	39.2	0.000(0)	0.001(0)
MORSE	SUB		1	149.889(100.0)	0.007(0.0)	564.8	7.2	94.18	52.2	0.000(0)	0.000(0)
GTVLIN	SUB		1	0.006(0.0)	0.006(0.0)	35.2	0.4	0.37	15.7	0.000(0)	0.001(0)
INPUT2	SUB		1	1.143(0.8)	0.004(0.0)	677.9	0.0	94.62	64.0	0.000(0)	0.000(0)
XSEC	SUB		1	1.139(0.8)	0.004(0.0)	29.9	0.0	0.83	39.9	0.000(0)	0.001(0)
GETMUS	SUB		400	0.004(0.0)	0.004(0.0)	36.3	2.4	27.84	5.5	0.000(0)	0.002(0)
MAIN_	MAIN		1	149.913(100.0)	0.004(0.0)	829.9	0.0	94.78	64.0	0.000(0)	0.000(0)
OUTP3	SUB		1	0.078(0.1)	0.002(0.0)	23.4	0.1	0.76	27.6	0.000(0)	0.001(0)
OUTP2	SUB		2	0.002(0.0)	0.002(0.0)	28.6	0.2	0.11	13.0	0.000(0)	0.000(0)
RANU21	SUB		1	0.001(0.0)	0.001(0.0)	99.1	64.9	0.00	0.0	0.000(0)	0.000(0)
RNDIN	SUB		1	0.000(0.0)	0.000(0.0)	30.6	0.4	2.33	31.9	0.000(0)	0.000(0)
RNDOUT	SUB		2	0.000(0.0)	0.000(0.0)	32.6	0.3	0.00	0.0	0.000(0)	0.000(0)
ALBERT	SUB		3	0.000(0.0)	0.000(0.0)	64.2	17.8	40.41	7.0	0.000(0)	0.000(0)
INITSOR	SUB		1	0.000(0.0)	0.000(0.0)	24.9	2.9	0.00	0.0	0.000(0)	0.000(0)
STRUN	SUB		1	0.000(0.0)	0.000(0.0)	19.6	0.0	0.00	0.0	0.000(0)	0.000(0)
NBATCH	SUB		2	0.000(0.0)	0.000(0.0)	205.0	54.8	80.20	11.0	0.000(0)	0.000(0)

Table.5.3(b) Summary list of behavior analysis of morse code parallel version (probrem.2)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GGV	SUB		676935	59.016(39.7)	59.016(39.7)	151.8	20.8	87.33	36.7	0.515(0)	17.127(12)
G1V	SUB		35474	101.382(68.1)	42.629(28.7)	158.0	7.2	93.85	42.6	0.337(0)	11.220(8)
FLUXSV	SUB		160164	14.161(9.5)	14.161(9.5)	68.6	4.0	26.41	40.9	0.003(0)	2.505(2)
RELCOV	SUB		179	78.873(53.0)	10.368(7.0)	236.1	50.8	92.54	48.4	0.011(0)	3.570(2)
EUCLIV	SUB		28487	97.342(65.4)	6.017(4.0)	157.7	6.8	85.86	50.2	0.032(0)	1.202(1)
MSOURP	SUB		104	55.758(37.5)	3.853(2.6)	274.2	191.6	94.13	63.6	0.000(0)	0.839(1)
SDATAV	SUB		104	46.985(31.6)	3.360(2.3)	113.3	34.9	57.34	44.4	0.002(0)	0.134(0)
FNSSOC	SUB		50000	3.680(2.5)	3.030(2.0)	34.2	5.9	0.00	0.0	0.000(0)	1.256(1)
MACRO4	SUB		6	1.285(0.9)	1.285(0.9)	47.2	2.0	0.23	53.6	0.000(0)	0.144(0)
SOURCE	SUB		50000	4.566(3.1)	0.886(0.6)	14.2	0.1	27.09	34.0	0.000(0)	0.586(0)
ROTAT1	SUB		100000	0.650(0.4)	0.650(0.4)	18.9	2.3	0.00	0.0	0.000(0)	0.558(0)
GTMED	SUB		158916	0.629(0.4)	0.629(0.4)	19.9	0.0	0.00	0.0	0.000(0)	0.397(0)
RESTOR	SUB		8	0.735(0.5)	0.560(0.4)	49.6	0.0	5.65	57.8	0.000(0)	0.127(0)
GOMST	SUB		2063	10.442(7.0)	0.384(0.3)	286.8	13.1	98.30	54.4	0.008(0)	0.038(0)
NXTCOL	SUB		186	10.868(7.3)	0.337(0.2)	120.6	4.0	82.87	53.7	0.002(0)	0.089(0)
NRUN	SUB		1	0.192(0.1)	0.189(0.1)	25.8	0.2	0.04	21.7	0.000(0)	0.048(0)
REARAG	SUB		864	0.175(0.1)	0.175(0.1)	66.2	3.9	1.65	63.8	0.000(0)	0.035(0)
MACRO2	SUB		12	0.164(0.1)	0.164(0.1)	81.6	26.0	0.00	0.0	0.000(0)	0.045(0)
APMATX	SUB		1	0.133(0.1)	0.133(0.1)	67.6	0.0	32.75	63.0	0.000(0)	0.034(0)
JNPUT	SUB		1	1.800(1.2)	0.110(0.1)	79.1	0.3	0.02	55.6	0.000(0)	0.003(0)
MAIN_	MAIN		1	38.034(25.6)	0.090(0.1)	88.8	0.0	0.01	41.6	0.000(0)	0.000(0)
VLOOKZP	SUB		104	0.351(0.2)	0.088(0.1)	276.8	14.9	95.42	56.9	0.000(0)	0.013(0)
MACRO1	SUB		12	1.537(1.0)	0.084(0.1)	85.9	0.0	0.00	28.0	0.000(0)	0.006(0)
COLISN	SUB		179	0.083(0.1)	0.083(0.1)	117.2	32.9	63.36	43.1	0.000(0)	0.014(0)
BATCH	SUB		8	145.662(97.9)	0.072(0.0)	131.5	2.9	85.58	59.2	0.000(0)	0.017(0)
GETETA	SUB		186	0.067(0.0)	0.067(0.0)	106.7	28.8	62.31	59.4	0.000(0)	0.005(0)
GENI	SUB		1	0.061(0.0)	0.061(0.0)	25.0	0.2	0.00	0.0	0.000(0)	0.019(0)
SCORIN	SUB		1	0.058(0.0)	0.058(0.0)	22.9	0.1	0.00	0.0	0.000(0)	0.018(0)
JOMIN	SUB		1	0.116(0.1)	0.053(0.0)	24.0	0.2	0.06	42.2	0.000(0)	0.018(0)
INPUT1	SUB		1	0.155(0.1)	0.028(0.0)	26.4	0.2	2.91	60.3	0.000(0)	0.007(0)
BANKRV	SUB		3722	126.081(84.7)	0.028(0.0)	14.4	0.0	15.72	55.4	0.000(0)	0.019(0)
OUTPT2	SUB		1	0.027(0.0)	0.027(0.0)	22.9	0.1	0.00	0.0	0.000(0)	0.008(0)
MACRO3	SUB		7	1.310(0.9)	0.025(0.0)	48.9	0.0	0.06	33.5	0.000(0)	0.000(0)
FNSSOO	SUB		1	0.014(0.0)	0.014(0.0)	29.7	1.2	0.00	0.0	0.000(0)	0.004(0)
SOINP	SUB		1	0.011(0.0)	0.011(0.0)	38.3	0.1	0.00	0.0	0.000(0)	0.002(0)
SORIN	SUB		1	0.011(0.0)	0.011(0.0)	24.0	0.2	1.14	50.9	0.000(0)	0.003(0)
MORSE	SUB		1	37.943(25.5)	0.007(0.0)	309.0	1.5	95.39	63.1	0.000(0)	0.001(0)
INPUT2	SUB		1	2.610(1.8)	0.006(0.0)	441.2	0.0	94.59	64.0	0.000(0)	0.000(0)
GTNDSK	SUB		1	0.741(0.5)	0.006(0.0)	60.0	0.0	0.85	60.0	0.000(0)	0.001(0)
XSEC	SUB		1	2.546(1.7)	0.005(0.0)	25.3	0.1	0.75	46.1	0.000(0)	0.001(0)
OUTP1	SUB		104	0.003(0.0)	0.003(0.0)	752.5	267.8	98.42	62.7	0.000(0)	0.001(0)
STBTCP	SUB		2	0.003(0.0)	0.003(0.0)	448.3	0.0	94.37	60.2	0.000(0)	0.000(0)
OUTP3	SUB		1	0.029(0.0)	0.002(0.0)	21.8	0.0	0.78	28.7	0.000(0)	0.001(0)

Table.5.3(c) Summary list of behavior analysis of morse code parallel version (probrem.3)

PROGRAM UNIT SUMMARY LIST

PROG.UNIT	ATR.	CODE	FREQUENCY	INCLUSIVE CPU TIME(%)	EXCLUSIVE CPU TIME(%)	MOPS	MFLOPS	V.OP. RATIO	AVER. V.LEN	MEMORY LOSS TIME BANK CONF.(%)	CACHE MISS(%)
GGV	SUB		97420	18.368(36.5)	18.368(36.5)	134.8	23.9	81.28	40.8	0.311(1)	6.201(12)
G1V	SUB		3909	26.870(53.4)	8.636(17.2)	184.6	7.4	95.78	46.7	0.160(0)	2.190(4)
MSOURP	SUB		104	5.477(10.9)	3.954(7.9)	267.2	186.7	94.12	63.6	0.000(0)	0.935(2)
BATCH	SUB		8	47.294(94.0)	2.886(5.7)	105.7	3.8	76.27	56.0	0.012(0)	1.013(2)
COLISN	SUB		3828	2.868(5.7)	2.868(5.7)	155.8	44.5	74.70	52.9	0.024(0)	0.518(1)
TESTW	SUB		3836	3.814(7.6)	2.367(4.7)	41.5	2.1	68.16	49.7	0.006(0)	1.731(3)
FLUXSV	SUB		11207	1.827(3.6)	1.827(3.6)	67.1	2.7	30.41	50.8	0.001(0)	0.555(1)
GOMST	SUB		3909	28.425(56.5)	1.555(3.1)	319.9	14.6	98.91	59.9	0.046(0)	0.091(0)
SOURCE	SUB		50000	1.322(2.6)	1.322(2.6)	27.3	3.4	0.00	0.0	0.000(0)	0.684(1)
VINTERP	SUB		133888	1.134(2.3)	1.134(2.3)	19.9	0.0	0.00	0.0	0.000(0)	0.716(1)
RESTOR	SUB		8	1.034(2.1)	0.866(1.7)	44.5	0.0	4.24	57.2	0.000(0)	0.228(0)
GETETA	SUB		3823	0.850(1.7)	0.850(1.7)	104.2	28.0	62.07	56.3	0.000(0)	0.079(0)
MACRO4	SUB		2	0.619(1.2)	0.619(1.2)	43.2	1.6	0.23	61.7	0.000(0)	0.103(0)
TLE7V	SUB		7379	1.591(3.2)	0.539(1.1)	211.6	65.5	88.36	50.8	0.004(0)	0.141(0)
OUTPT2	SUB		1	0.403(0.8)	0.403(0.8)	28.4	0.2	0.23	15.7	0.000(0)	0.089(0)
TLE5V	SUB		3828	1.143(2.3)	0.367(0.7)	232.7	76.0	88.54	53.8	0.003(0)	0.079(0)
GETNT	SUB		66945	0.313(0.6)	0.313(0.6)	23.5	0.0	0.00	0.0	0.000(0)	0.183(0)
STORNT	ENT		66944								
SETNT	ENT		1								
BANKRV	SUB		11317	3.127(6.2)	0.238(0.5)	10.7	0.0	0.04	54.0	0.000(0)	0.197(0)
REARAG	SUB		1000	0.167(0.3)	0.167(0.3)	70.5	3.9	2.17	64.0	0.000(0)	0.027(0)
MACRO2	SUB		9	0.159(0.3)	0.159(0.3)	84.3	26.9	0.00	0.0	0.000(0)	0.041(0)
NRUN	SUB		1	0.148(0.3)	0.146(0.3)	39.0	0.1	0.01	37.8	0.000(0)	0.008(0)
JNPUT	SUB		1	0.986(2.0)	0.104(0.2)	72.9	0.1	0.02	54.4	0.000(0)	0.002(0)
MAIN_	MAIN		1	14.477(28.8)	0.097(0.2)	82.3	0.0	0.01	41.6	0.000(0)	0.000(0)
GENI	SUB		1	0.078(0.2)	0.078(0.2)	27.9	0.1	0.00	0.0	0.000(0)	0.020(0)
JOMIN	SUB		1	0.153(0.3)	0.069(0.1)	26.3	0.1	0.04	42.2	0.000(0)	0.020(0)
VLOOKZP	SUB		104	0.200(0.4)	0.066(0.1)	125.4	9.1	64.47	62.5	0.000(0)	0.013(0)
SCORIN	SUB		1	0.055(0.1)	0.055(0.1)	21.8	0.1	0.00	0.0	0.000(0)	0.017(0)
INPUT1	SUB		1	0.215(0.4)	0.051(0.1)	40.6	0.1	28.95	49.6	0.000(0)	0.012(0)
APMATX	SUB		1	0.046(0.1)	0.046(0.1)	69.0	0.0	41.36	62.4	0.000(0)	0.012(0)
MACRO1	SUB		9	0.821(1.6)	0.041(0.1)	80.8	0.0	0.00	28.0	0.000(0)	0.004(0)
MACRO3	SUB		3	0.637(1.3)	0.017(0.0)	80.3	0.0	0.03	34.6	0.000(0)	0.000(0)
SOURCO	SUB		1	0.022(0.0)	0.013(0.0)	22.0	0.2	0.04	62.0	0.000(0)	0.004(0)
NBATCP	SUB		2	0.011(0.0)	0.011(0.0)	111.6	15.0	0.00	0.0	0.000(0)	0.002(0)
MORSE	SUB		1	14.380(28.6)	0.010(0.0)	381.9	22.4	96.49	63.1	0.000(0)	0.001(0)
SORIN	SUB		1	0.010(0.0)	0.010(0.0)	25.6	0.2	1.15	49.8	0.000(0)	0.003(0)
SOINP	SUB		1	0.009(0.0)	0.009(0.0)	27.2	0.4	0.00	0.0	0.000(0)	0.002(0)
GTNDSK	SUB		1	1.041(2.1)	0.007(0.0)	59.8	0.0	0.66	60.0	0.000(0)	0.001(0)
INPUT2	SUB		1	2.095(4.2)	0.007(0.0)	475.3	0.0	94.64	64.0	0.000(0)	0.000(0)
GTVLIN	SUB		1	0.005(0.0)	0.005(0.0)	33.6	0.4	0.35	15.7	0.000(0)	0.001(0)
NBATCH	SUB		2	0.005(0.0)	0.005(0.0)	90.7	13.9	0.00	0.0	0.000(0)	0.000(0)
XSEC	SUB		1	2.032(4.0)	0.005(0.0)	24.5	0.1	0.85	37.5	0.000(0)	0.001(0)

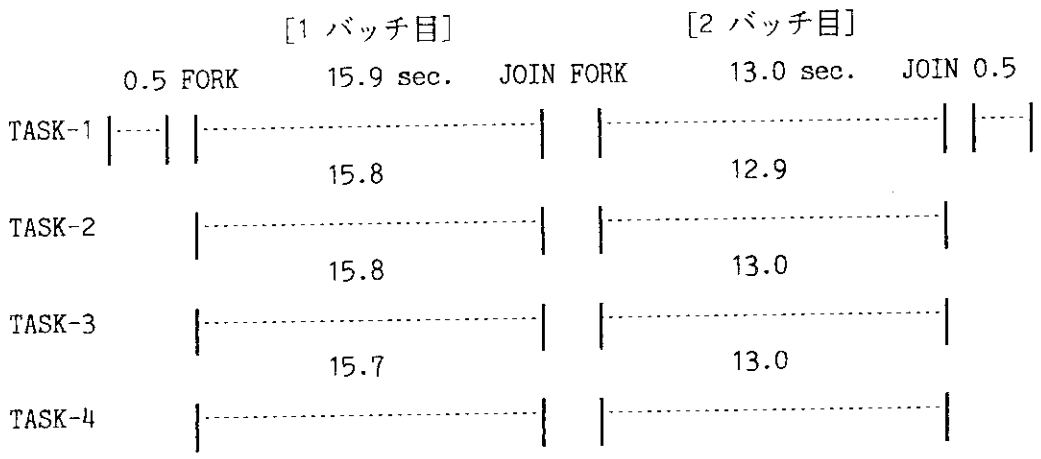


Fig.5.1 Time chart of parallelized MORSE code

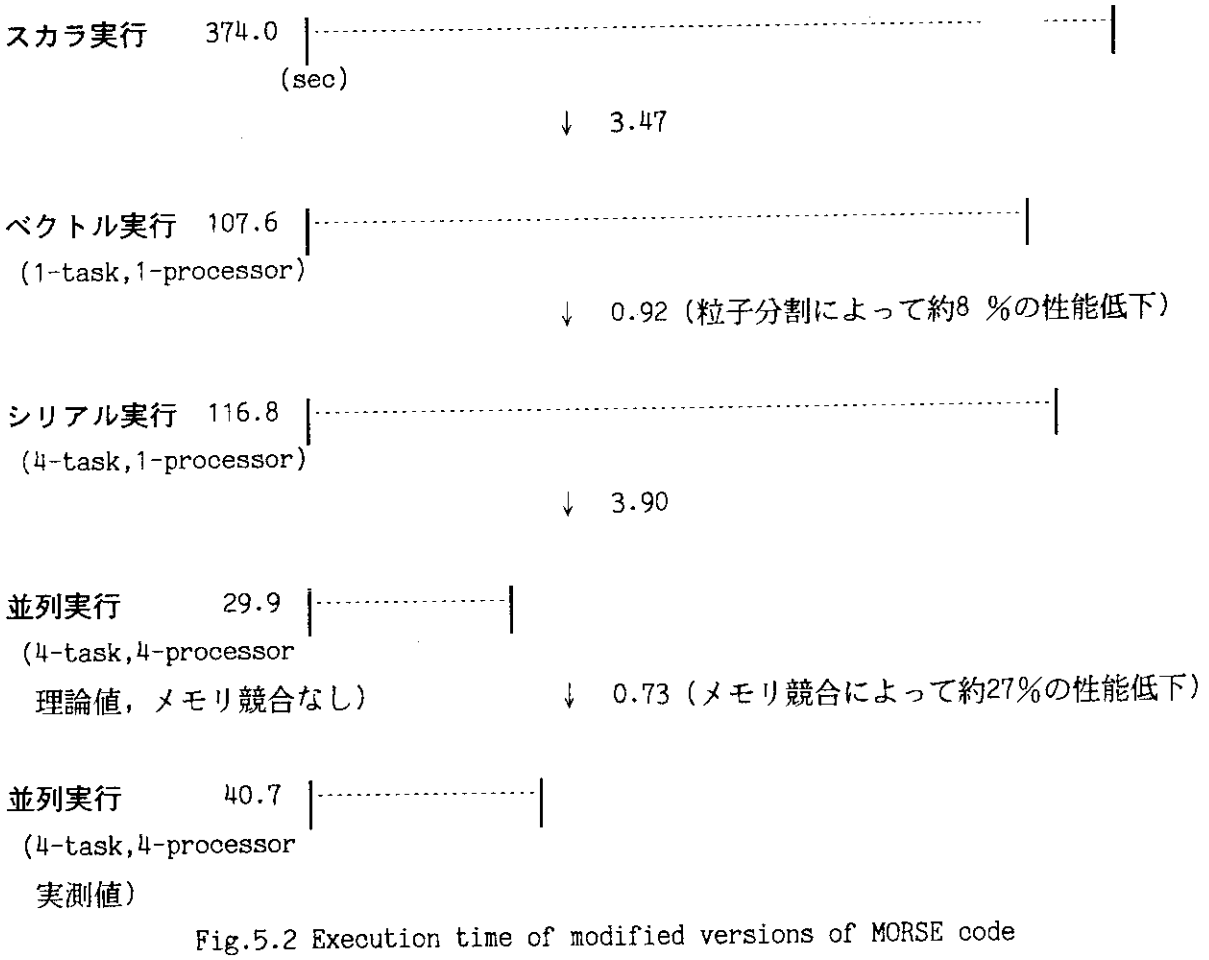


Fig.5.2 Execution time of modified versions of MORSE code

6．おわりに

今回の並列化によって「既存のコードのスカラ処理に対して１０倍」という目標に近い性能を得ることができたが，メモリ競合のために並列化による性能が十分に得られなかった。メモリ競合による性能低下が，共有メモリ型のコンピュータの欠点であることは一般に言われており，MORSE コードにおいてもこれを確認する結果となった。今後，プロセッサ数を増やし，よりモンテカルロ・コードを高速化するために，メモリ競合の少ないハードウェアの出現を期待したい。

謝辞

保健物理部・線量計測課の山口恭弘氏にはMORSE コードのソース，断面積ライブラリ及び入力データを提供して頂いた。深く感謝致します。

（株）ＮＥＣ情報システムズの浅見暁氏にはコードの並列化及び高速化作業中細部にわたり貴重なる助言を頂いた。深く感謝致します。

（株）日本総合研究所の佐々木誠氏には乱数生成法について貴重なる助言を頂いた。深く感謝致します。

計算科学技術推進センター並列処理支援技術開発グループの相川裕史氏には本稿査読の際、細部にわたり貴重なる助言を頂いた。深く感謝致します。

6．おわりに

今回の並列化によって「既存のコードのスカラ処理に対して１０倍」という目標に近い性能を得ることができたが，メモリ競合のために並列化による性能が十分に得られなかった。メモリ競合による性能低下が，共有メモリ型のコンピュータの欠点であることは一般に言われており，MORSE コードにおいてもこれを確認する結果となった。今後，プロセッサ数を増やし，よりモンテカルロ・コードを高速化するために，メモリ競合の少ないハードウェアの出現を期待したい。

謝辞

保健物理部・線量計測課の山口恭弘氏にはMORSE コードのソース，断面積ライブラリ及び入力データを提供して頂いた。深く感謝致します。

（株）ＮＥＣ情報システムズの浅見暁氏にはコードの並列化及び高速化作業中細部にわたり貴重なる助言を頂いた。深く感謝致します。

（株）日本総合研究所の佐々木誠氏には乱数生成法について貴重なる助言を頂いた。深く感謝致します。

計算科学技術推進センター並列処理支援技術開発グループの相川裕史氏には本稿査読の際、細部にわたり貴重なる助言を頂いた。深く感謝致します。

参考文献

(1)Kiyoshi ASAI 他：VECTORIZATION OF KENO IV CODE AND AN ESTIMATE OF VECTOR-PARALLEL PROCESSING, JAERI-M86-15, 1986

(2)栗田 他：モンテカルロ・コードMCNPのベクトル化, JAERI-M87-022 , 1987

(3)樋口健二, 山崎隆, 浅井清：MORSE-DDコードのベクトル化, JAERI-M87-023 , 1987.

(4)菅沼正之 他：連続エネルギー・モンテカルロ・コードVIM のベクトル化, JAERI-M86-190, 1986

(5)浅井清 他：原子力知能化システム技術の研究－平成元年度作業報告書－, JAERI-M90-060, 1990

(6)浅井清 他：原子力知能化システム技術の研究－平成２年度作業報告書－, JAERI-M91-101, 1991

(7)秋元正幸 他：原子力知能化システム技術の研究－平成３年度作業報告書－, JAERI-M92-198 , 1993

(8)秋元正幸 他：原子力知能化システム技術の研究－平成４年度作業報告書－, JAERI-M94-051 , 1994

(9)M.B.Emmett ：The MOSRSE Monte Carlo Radiation Transport Code System, ORNL-4972(1975)

(10)超高速モンテカルロ装置 ANALYZER-P/m4 利用の手引, 日本電気株式会社マニュアル

(11)超高速モンテカルロ装置 FORTRAN77/m4 並列処理機能利用の手引, 日本電気株式会社マニュアル

(12)FACOM FORTRAN SSL II 使用手引書（科学用サブルーチン） 富士通（株）マニュアル

(13)超高速モンテカルロ装置 プログラミングの手引, 日本電気株式会社マニュアル