J. Comput. Chem. Jpn., Vol. 1, No. 1, pp. 29–36 (2002)

メッセージ通信ライブラリを用いたプログラムの並列化例と計算速度および計算精度の評価

田村 克浩[a], 稲富 雄一[b], 長嶋 雲兵[b]*

[a] 静岡県静岡工業技術センター, 〒 421-1298 静岡市牧ヶ谷 2078
[b] 産業技術総合研究所グリッド研究センター, 〒 305-8561 つくば市東 1-1-1
**e-mail: u.nagashima@aist.go.jp*

(Received: December 31, 2001; Accepted for publication: February 12, 2002; Published on Web: March 22, 2002)

2 進数における循環小数である 10 進数の 0.1 を 10^9 回逐次加えるプログラムと 10^4 回の部分和をとり、これを 10^5 回加え最終的に 10^9 回加えるプログラムを作成し、メッセージ通信ライブラリ MPI を用いてプログラムの並列化を行った。これらプログラムを Alta Technology AltaCluster、Hitachi SR8000、IBM RS/6000 SP、SGI Origin2000 の計 4 種類の並列計算機を用いて 8CPU までの実行速度および計算精度の評価を行った。その結果、通信量が演算量に比べて小さいため、いずれの並列計算機でもほぼ CPU 数に応じた高速化が図ることができた。また、計算精度はそれぞれのプログラムとも 4 つの並列計算機で全く同一であった。逐次加えていく場合は、情報落ちにより正確な解が得られないが、並列化により精度向上が観測された。10^4 回の部分和を加える場合は、逐次計算でも正しい解が得られるが、並列処理によりわずかな精度劣化が見られた。

キーワード : 並列計算, SPMD モデル, MPI

1 はじめに

Linpack ベンチマークによる計算速度のコンピュータ世界ランキング TOP500 Super Computer Site[1] のランキングホルダはほとんどが 256 個以上の CPU をもつ並列計算機であり、上位の計算機の中には 1 万近くの CPU をもつ計算機もみられる。10 年前のスーパーコンピュータは、高速な単一の CPU または数台程度で構成されていたが、ハードウエア性能の限界と開発コスト、およびソフトウエア上の並列プログラミング環境の向上から数百以上の複数の CPU を持つ並列計算機に取って代わられた。

スーパーコンピュータのような特殊なコンピュータだけでなく大学等の個々の研究室においても、コストパフォーマンスに優れたコモディティハードウエアを組み合わせたコンピュータ、いわゆる DOS/V パソコンを、複数台ネットワークで接続し分散メモリマルチプロセッサ型クラスタを形成し、安価なスーパーコンピュータとして利用することもしばしば行われている。この場合、コストが非常に重要になるため Linux または FreeBSD 等 を OS として採用し、MPICH、Parallel Virtual Machine (PVM) などのフリーのメッセージ通信ライブラリを利用した並列プログラミングが行われている。

並列計算は、並列台数に応じた高速化が図れることが最大の利点であるが、上記分散メモリマルチプロセッサ型クラスタではこれに加え、1）一台ではメモリ容量不足で扱えない問題を扱うことができる、2）大規模なシステム構築が容易である、3）共有メモリ型並列計算機（Shared Memory Parallel（SMP）machine）では大きな問題になるタイミングに依存するバグを内包しにくい等の利点を有している一方で、1）プログラミング時に通信をすべてスケジュールしなければならない、2）通信遅延のオーバヘッドが非常に大きい

という欠点を有している。そのため、プログラミング時のコーディングの良し悪しによって、実行速度が大きく変化する。[2]

本報告では、メッセージ通信ライブラリ MPI をもちいて、簡単な並列プログラムの実行速度の高速化および計算精度の挙動を Alta Technology AltaCluster、Hitachi　SR8000、IBM RS/6000 SP、SGI Origin2000 の計 4 種類の並列計算機で測定したので報告する。

2　プログラムと MPI を用いた並列化

2 進数における循環小数である 10 進数の 0.1 を 10^9 回分加える Fortran のプログラムを 2 種類作成した。一つは、0.1 を 10^9 回逐次加え、総和を求めるプログラム（Program 1 in Figure 1）、もう一つは 10^4 回の部分和の和を 10^5 回加えることで、最終的に 0.1 の 10^9 回の総和を計算するプログラム（Program 2 in Figure 2）を作成した。この 2 つのプログラムを MPI を用いて SPMD（Single Program Multiple Data）モデルによる並列化を行った（それぞれ Figure 3（program 1）および Figure 4（program 2））。Figure 3 と Figure 4 には並列化のために加えた命令を太字で示した。計算は計算誤差が現れやすいようすべて単精度で行った。

用いた並列計算機は、産業技術総合研究所先端計算機センター (TACC) に設置されている並列計算機 Alta Technology AltaCluster、Hitachi　SR8000、IBM RS/6000 SP および SGI Origin2000 を用いた。計算機の詳細を Table 1 に示した。なお、並列計算は 1、2、4、8CPU を用いて行い、プログラムの実行時間は、MPI の初期化、ノード数およびランクの取得直後から終了処理の直前までの時間を 5 回計測し、最短時間を採用した。また、コンパイル時の最適化等のオプションは使用しなかった。

Table 1. Details of Computer Architecture

	AltaCluster	Hitach SR8000	IBM RS/6000 SP	SGI Origin2000
CPU	Pentium III (katmai)	SR8000	POWER3	R12000
CPU 数	8	512	256	64
クロック周波数 (MHz)	450	250	200	400
メモリサイズ	256MB/CPU	8GB/8CPUs	960MB/2CPUs	2GB/CPU
１次 CachSize(データ＋命令)	16KB+16KB	128KB+64KB	64KB+32KB	32KB+32KB
ネットワーク種	ether (10Base-T)	多次元クロスバー	SP スイッチ	NUMA SMP
Fortran コンパイラ名	PGI Fortran	最適化 F90 コンパイラ	XL Fortrran for AIX	MIPSpro F90
Fortran コンパイラバージョン	3.1	01-03	7.1	7.3

```
      PROGRAM main
      IMPLICIT NONE
      INTEGER nmax
      PARAMETER(nmax=1000000000)
      INTEGER i
      REAL sum
      sum=0.0e0
      DO i=1,nmax
        sum=sum+0.1e0
      ENDDO
      WRITE(6,200) sum
  200 FORMAT('The answer is ',e15.8,'.')
      END
```

Figure 1. Sequential Summation

```
      PROGRAM main
      IMPLICIT NONE
      INTEGER nmax,nunitsize
      PARAMETER(nmax=1000000000,nunitsize=10000)
      INTEGER i,j,nunit,namari
      REAL sum,psum,asum
      sum=0.0e0
      nunit=INT(nmax/nunitsize)
      namari=MOD(nmax,nunitsize)
      DO i=1,nunit
        psum=0.0e0
        DO j=1,nunitsize
          psum=psum+0.1e0
        ENDDO
        sum=sum+psum
      ENDDO
      IF(namari/=0) THEN
        asum=0.0e0
        DO i=1,namari
          asum=asum+0.1e0
        ENDDO
        sum=sum+asum
      ENDIF
      WRITE(6,200) sum
  200 FORMAT('The answer is ',e15.8,'.')
      END
```

Figure 2. Summation with Partial Sum

```
      PROGRAM main
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INTEGER nmax
      PARAMETER(nmax=1000000000)
      INTEGER i,j,k,nunit,istart,iend,
     &        ierr,myrank,isize,
     &        istatus(MPI_STATUS_SIZE)
      REAL sum,buf
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,isize,ierr)
      nunit=INT(nmax/isize)
      istart=nunit*myrank+1
      iend=nunit*(myrank+1)
      sum=0.0e0
      IF(myrank==isize-1) THEN
        DO i=istart,nmax
          sum=sum+0.1e0
        ENDDO
      ELSE
        DO i=istart,iend
          sum=sum+0.1e0
        ENDDO
      ENDIF
      IF(myrank==0) THEN
        DO i=1,isize-1
          CALL MPI_RECV(buf,1,MPI_REAL,i,MPI_ANY_TAG,
     &                  MPI_COMM_WORLD,istatus,ierr)
          sum=sum+buf
        ENDDO
      ELSE
        buf=sum
        CALL MPI_SEND(sum,1,MPI_REAL,0,1,
     &                MPI_COMM_WORLD,ierr)
      ENDIF
      IF(myrank==0) THEN
        WRITE(6,200) sum
  200   FORMAT('The answer is',e15.8,'.')
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Figure 3. Parallelized Program of Sequential Summation.(Program1)

```
      PROGRAM main
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INTEGER nmax,nunitsize
      PARAMETER(nmax=1000000000,nunitsize=10000)
      INTEGER i,j,ierr,myrank,isize,
     &        nunit,nunitamari,namari,istart,
     &        iend,istatus(MPI_STATUS_SIZE)
      REAL sum,psum,asum,buf
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,isize,ierr)
      nunit=INT(INT(nmax/nunitsize)/isize)
      nunitamari=MOD(INT(nmax/nunitsize),isize)
      namari=MOD(nmax,nunitsize)
      istart=nunit*myrank+1
      iend=nunit*(myrank+1)
      sum=0.0e0
      IF(myrank==isize-1) THEN
        DO i=istart,iend
          psum=0.0e0
          DO j=1,nunitsize
            psum=psum+0.1e0
          ENDDO
          sum=sum+psum
        ENDDO
        IF(namari/=0) THEN
          asum=0.0e0
          DO i=1,namari
            asum=asum+0.1e0
          ENDDO
          sum=sum+asum
        ENDIF
      ELSE
        IF(nunitamari>myrank) THEN
          iend=iend+1
        ENDIF
        DO i=istart,iend
          psum=0.0e0
          DO j=1,nunitsize
            psum=psum+0.1e0
          ENDDO
          sum=sum+psum
        ENDDO
      ENDIF
      IF(myrank==0) THEN
        DO i=1,isize-1
          CALL MPI_RECV(buf,1,MPI_REAL,i,MPI_ANY_TAG,
     &                  MPI_COMM_WORLD,istatus,ierr)
          sum=sum+buf
        ENDDO
      ELSE
        CALL MPI_SEND(sum,1,MPI_REAL,0,1,
     &                MPI_COMM_WORLD,ierr)
      ENDIF
      IF(myrank==0) THEN
        WRITE(6,200) sum
  200   FORMAT('The answer is',e15.8,'.')
      ENDIF
      CALL MPI_FINALIZE(ierr)
      END
```

Figure 4. Parallelized Program of Summation with Partial Sum. (Program2)

3 結果と考察

Table 2 にプログラムの実行時間および速度向上率を示した。Figure 5 は Table 2 の結果をグラフ化したものである。

実行速度は期待されるように CPU に応じてほぼ理想的な高速化が図られた。これは、プログラム 1、2 共に計算量に対して通信量が非常に小さく、比較的低速な TCP/IP を利用する AltaCluster においても MPI の通信オーバヘッドが問題にならないことおよび負荷分散がきちんとできているためである。

Table 3 に 2 つのプログラムによる和の値および誤差率を示した。

計算精度では、プログラム 1 および 2 においてそれぞれ 4 つの並列計算機で全く同じ結果が得られた。これは、現在製造されているほぼすべての CPU は 2 進浮動小数点数の標準規格 IEEE754 に従っており、今回用いた計算機では浮動小数点数の取り扱いに違いがないために、結果が同一になった。また、コンパイラによる最適化を行わなかったことも、同一の結果を与え

ることに寄与している。

Table 2. Execution Time and Speed-up Ratio* in Number of Processors

	Number of Processors							
	1		2		4		8	
	Execution time	Speed-up ratio	Execution time	Speed-up ratio	Execution time	Speed-up ratio	Execution time	Speed-up ratio
	sec	—	sec	—	sec	—	sec	—
Alta Cluster								
Program 1	14.41	1	8.80	1.64	4.45	3.24	2.21	6.52
Program 2	17.78	1	8.88	2.00	4.44	4.00	2.21	8.05
SR-8000								
Program 1	82.57	1	41.11	2.01	20.56	4.02	10.30	8.02
Program 2	89.23	1	44.51	2.00	22.31	4.00	11.15	8.00
RS/6000 SP								
Program 1	34.72	1	17.34	2.00	8.67	4.00	4.34	8.00
Program 2	34.73	1	17.35	2.00	8.68	4.00	4.34	8.00
Origin2000								
Program 1	25.63	1	12.76	2.01	6.37	4.02	3.19	8.03
Program 2	25.63	1	12.83	2.00	6.41	4.00	3.21	7.98

* Speed-up ratio = execution time with 1 processor / execution time with N processors.

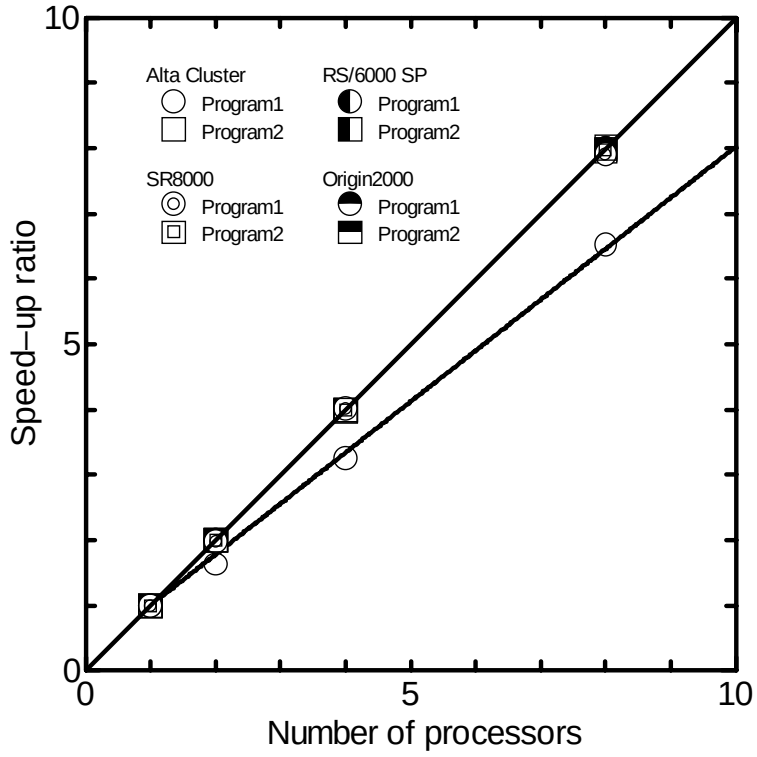

Figure 5. Execution Time and Speed-up Ratio in Number of Processors

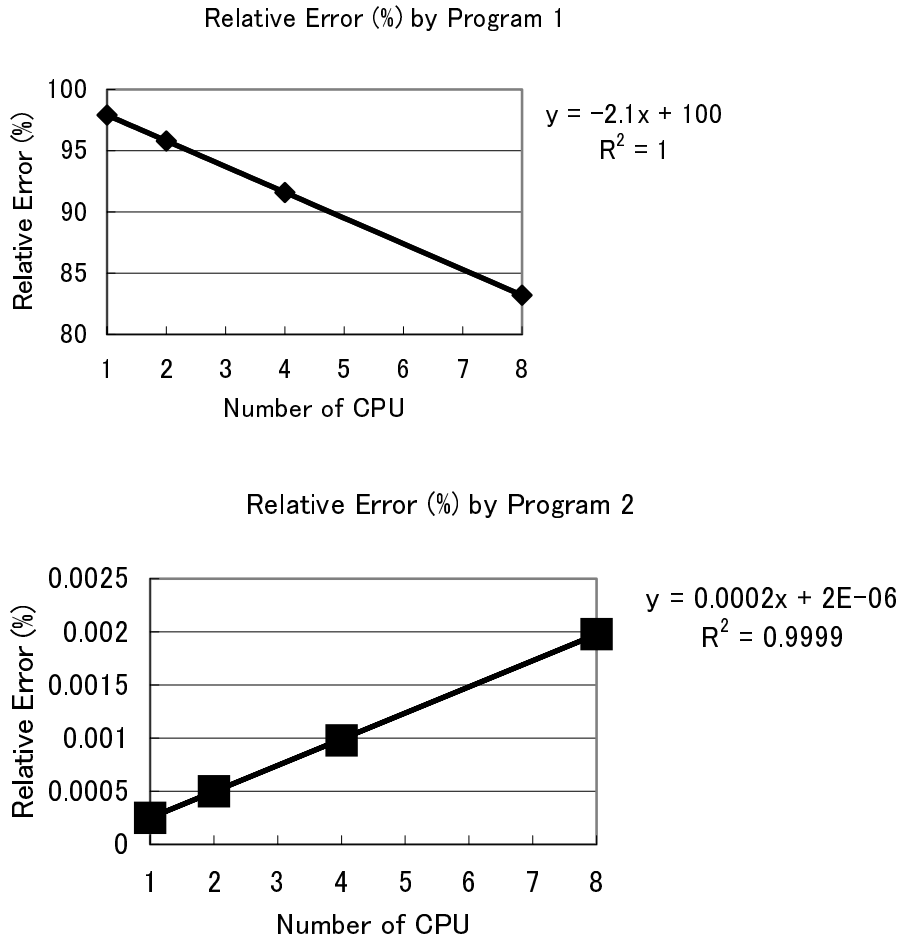

Figure 6. Relative Error (%) by Programs 1 and 2

Table 3. Calculated Value and Relative Error * in Number of Processors

	Number of Processors							
	1		2		4		8	
	Calculated value	Rel. error*	Calculated value	Rel. error*	Calculated value	Rel. error*	Calculated value	Rel. error*
	—	%	—	%	—	%	—	%
All computers								
Program 1	2.09715E+06	-97.9	4.1943E+06	-95.8	8.38861E+06	-91.6	1.67772E+07	-83.2
Program 2	9.99998E+07	-0.00025	9.99995E+07	-0.00050	9.99990E+07	-0.00098	9.99980E+07	-0.00198

*Rel. error = (calculated value - exact value) / exact value100.0 ： (exact value=1.000E+8).

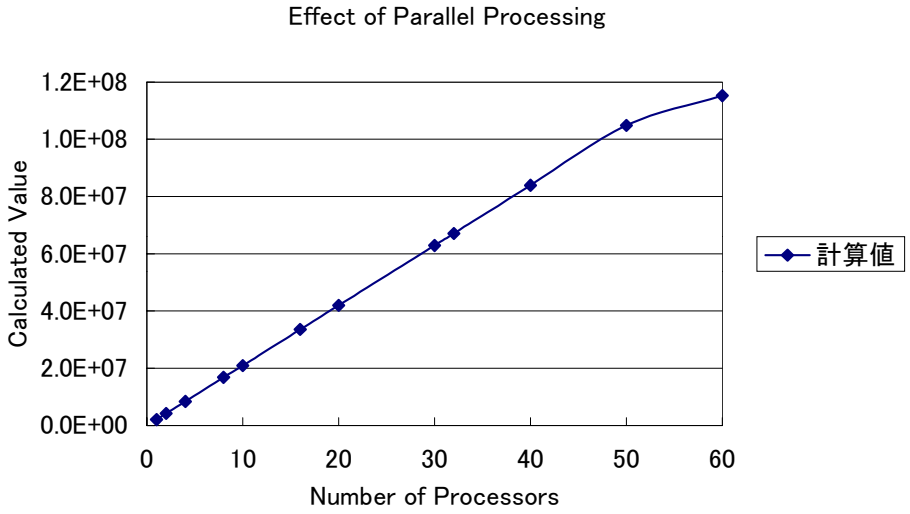

Figure 7. Calculated Values of Program 1 on SGI Origin 2000

プログラム 1 は、非常に単純なプログラムであるにもかかわらず、良く知られているように全く正しい解を与えない。これは、“ 情報落ち ”という誤差の累積によるものであることが広く知られている。Table 3 をみると予想どおり並列処理を行うことによって情報落ちが少なくなるため、CPU 数を増加させ処理を分散させることにより精度が向上しているが、８ CPU 程度では十分な計算誤差の回復には至っておらず、でたらめな答えとなっている。Figure 6 に示す回帰直線（y=-2.1x+100.0）から外装すると、50CPU 程度を用いた並列処理を行うと正しい計算結果を与えるようになることが期待される。詳細な議論は行わないが、SGI Origin2000 60CPU までの program 3 の実行結果を Figure 7 に示した。50CPU 程度までは、直線的に精度が向上している。Figure 6 の回帰直線からも示唆されるように 50CPU よりも少ないところで正しい値 (1.0E+08) となり、さらに用いる CPU 数が 60 程度に大きくなるとまたエラーが増大するが、50CPU 程度までにみられるような直線的増加ではない。

プログラム１とプログラム２は原理的に同一の解を与えるはずであるが、数値的には全く異なる解を与える。この例は、プログラムの書き方によって計算結果が異なってくることの良い例となっている。

プログラム 2 は、部分和を取ることにより情報落ちを回避するプログラムとなっているためほぼ正しい解が得られている。しかし、Table 3 および Figure 6 から判るように、並列処理を行うことで、非常にわずかであるが見かけ上精度が悪くなっており、並列分散処理を行うと精度が向上するという一般的な傾向とはずれている。これは、以下のような考察により理解できる。まず、１つの CPU において 10^4 回の部分和が数回計算されるが、この時丸め誤差を含んだ数を加えたことによる誤差が蓄積され、例えば 0.1 の 10^4 回の部分和は、正確な値（1000.0）にはならず、999.902 … と誤差を含んだ値をとる。部分和の和をとっていくごく初期の段階では、部分和の和と各部分和の大きさがそれほど違わないため、丸めに起因する部分和の段階での誤差が次第に蓄積していく。ところが、部分和の総和が、各部分和の絶対値に比べて非常に大きくなっていくと、部分和段階での演算誤差が丸めにより切り上げられ、偶然真値（1000.0）として加えられる。そのため部分和の和をとるごく初期の段階を除けば、部分和の和に誤差が蓄積しない。逆に並列処理を行うと部分和の和のごく初期にたまった誤差が和になって表れ、CPU 数が増すことにより誤差が比例的に増す。したがって、プログラム１および 2 の並列実行時にみられた現象は、MPI による並列処理に起因する誤差でなく、部分和の取り方の違いによるよる現象であると考えられる。これは、1CPU を用いて MPI での並列処理による和の取り方をシミュレートすることにより全く同様の結果を得たことからも確かめられた。

4　まとめ

2 進循環小数の 0.1 を 10^9 回加えるプログラムを例にとり、MPI による並列プログラムの実行速度の高速化および計算精度の挙動を 4 種類の並列計算機を用いて測定した。２つのプログラムにおいて、通信量は演算量に比べて非常に小さい。そのためいずれの並列計算

機でも、ほぼ CPU 数に応じた高速化が図れた。演算精度はそれぞれのプログラムとも 4 つの並列計算機で全く同一であった。0.1 を逐次加えていく場合（Program 1）では、並列処理により著しい精度向上が観測された。計算誤差の蓄積が非常に少ない 10^4 回の部分和を 10^5 回加える場合 (Program 2) では CPU が増加することにより、非常にわずかな精度劣化が見られた。これは部分和の取り方による誤差の挙動変化であり、この現象は偶然生じた現象であると考えられる。

並列処理をおこなうことで、これまでに扱うことが困難であった大規模な数値計算を短時間に行うことが可能となってきた。しかしながら演算回数が増えるにつれ、丸め誤差や情報落ちなど、数値誤差が入り込む部分も多くなるため、小規模な数値計算の際にはあまり問題にならなかったような場合においても、誤差が小さくなるような計算手順の工夫が必要である。

この研究は静岡県職員資質向上研修として行ったものである。計算は独立行政法人産業技術総合研究所先端情報計算センター (TACC) で行われた。

参考文献

[1] http://www.top500.org

[2] 湯淺, 安村, 中田編, はじめての並列プログラミング, *bit* 別冊, 共立出版 (1998).

Parallel Programming with Message Passing Library and Its Precision of Calculation

Katsuhiro TAMURA[a], Yuichi INADOMI[b] and Umpei NAGASHIMA[b]*

[a]Shizuoka Industrial Research Institute of Shizuoka Prefecture
2078 Makigaya, Shizuoka city, Shizuoka 421-1298, Japan
[b]National Institute of Advanced Industrial Science and Technology, Tsukuba Advanced Computing Center
1-1-1 Higashi, Tsukuba, Ibaraki 305-8561, Japan
**e-mail: u.nagashima@aist.go.jp*

Using two programs about summing up 0.1, which is the circulating decimal in binary numbers, 10^9 times, the efficiency of parallel processing for performance and accuracy of calculation was demonstrated. One program uses sequential summing up (program1), and the other involves summing up using partial sum technique of 10^4 times 10^5 times (program2). These programs were parallelized with a message passing interface: MPI. They were executed on 4 parallel computers, Alta Technology AltaCluster, Hitachi SR8000, IBM RS/6000 SP and SGI Origin2000, up to 8 processors. The performance is proportionally improved with increasing number of processors, because the communication process is small compared with the computation process. The computing precision was quite similar for the 4 computers. In the case of program1 the precision was improved drastically by increasing the number of processors, but little improvement was observed in the case of program 2. It was clearly shown that the numerical error accumulation, namely the loss of digits, was avoided by parallel processing.

Keywords: Parallel Computing, Single Program Multiple Data (SPMD) model, Message Passing Interface (MPI)