並列処理時におけるタスクの確保と解放について

--SX-4プログラム相談員より--

並列処理プログラムをチューニングする時に、確保するタスク数の設定とタスクを確保／解放するタイミングの調整があります。

確保されるタスク数の設定方法として

(1) 翻訳時オプションあるいは実行時オプションを指定をしない。

タスク数は実行される各ジョブクラスで設定されている値となります。すなわち、ジョブクラス毎にとり得る最大の値でプログラム実行することができます。

(2) (1)に加え実行時オプション F_RSVTASK でタスク数を指定。

タスク数は実行毎に実行時オプションによりタスク数を変更することができます。

(3) 翻訳時オプションを指定

タスク数は翻訳時に指定された値となります。

タスク数の決定方法については次回改めて詳細に説明いたします。

タスクは並列化指示行 reserve で OS に対して必要数を要求することにより確保され、指示行 release でタスクを OS に返すことにより解放されます。

以下では並列処理プログラムのチューニングの一つとしてタスクの確保／解放の処理をプログラム中のどの時点で行うかについて説明いたします。

自動並列化では以下の３つのレベルでのタスクの確保／解放が用意されています。

(a) プログラム全体

プログラム中に並列化されたループのある／なしにかかわらず、主プログラムの先頭でタスクを確保し、プログラム終了時に解放する。

```
f77/f90 -P auto -Wf"-fopp res=whole"    sample.f
```

(b) 手続き毎

自動並列化されたループを含む手続きにおいて、その先頭でタスクを確保し、return 文や end 文の直前でタスクを解放する。

```
f77/f90 -P auto -Wf"-fopp res=parunit"  sample.f
```

(c) ループ毎

並列化されたループの先頭と最後でタスクの確保と解放をおこなう。

```
f77/f90 -P auto -Wf"-fopp res=no"       sample.f
```

既定値は (b) の手続き毎となっています。

* 特定の手続きやループにプログラムの処理が集中している時には、(b)または(c)を、そうではなくプログラム全体に散らばっている様な場合は (a) を指定するのがよいでしょう。しかし、センター運用を考慮するとすべてのプログラムで (a) のプログラム全体を指定しても問題ありません。

次にタスクの確保／解放時の処理方式について説明します。

確保されたタスクは並列化された処理が来るまで、スピンウェイト(CPU ループ)して待ち状態になります。この待ち状態で使用した時間も、CPU 時間 (User Time)に含まれています。このため並列化された処理が全くないと、確保されたタスクはすべて無駄になってしまいます。

do ループが自動並列化されると、サブルーチンとして切り出され、そのサブルーチンが並列処理される下記の様なイメージとなります。
なおこれは (a) のプログラム全体でタスクの確保が行われた場合です。

```
(オリジナルソース)          (自動並列化後のイメージ)
program main                program main
...                         ...
...                       *pdir reserve
...                         ...
do 100 i =1,n               call sub$1
...                         ...
...                       *pdir release
...                         end
...                         subroutine sub$1
```

```
100  continue                    ...
    ...                          *pdir pardo for
    ...                          do 100 i=1,n
     end                         ...
                                 ...
                               100 continue
                                 ...
                                 return
                                 end
```

この例を 3 台で並列処理した時の各 CPU の処理イメージは以下の様になります。

```
       CPU 0       CPU 1         CPU 2

       main
        |
      reserve start        start        <-- CPU の確保
        |          .           .
        |        (待ち状態)   (待ち状態)   <-- スピンウェイト
        |          .           .             (CPUループ)
      call sub$1   sub$1        sub$1        <-- 並列処理開始
        |          |           |
        |          |           |
        |          |           |
        |          |           |
      return       return      return       <-- 並列処理終了
        |          .           .
        |        (待ち状態)   (待ち状態)   <-- スピンウェイト
        |          .           .             (CPUループ)
        |          .           .
      release end          end            <-- CPU の解放
        |        (タスク 1)     (タスク 2)
        |
      end
```

CPU 0 で reserve が実行されるとタスク 1 とタスク 2 が確保されます。確保されたタスク 1 と 2 はスピンウェイトし待ち状態となり、CPU 0 で sub$1 の実行が開始されると同時に並列処理を開始します。 sub$1 の処理が終了すると次の並列処理手続きが実行されるまで、あるいは release が実行されるまで再び待ち状態になります。

このため (c) のループ毎に指定した方が待ち状態の時間を少なくできますが、次の欠点があります。

プログラムによってはタスクの確保／解放の処理(reserve/release の処理)が非常に多く行われる様になり、結果として CPU 時間の増加と経過時間の増加を引き起こす。

逆に (a) のプログラム全体でタスクの確保／解放をおこなうと次の問題があります。

全く並列処理化されないプログラムにおいて、ほとんどのタスクが待ち状態となりCPU 時間が必要以上に大きくなる。

しかし、この問題を防ぐために以下の機能がありますので、待ち状態の時間が必要以上に大きくなることはありません。

一定の時間待ち状態が続くと確保されたタスクを一時切り離して OS にタスクを返し、次の並列処理がくるとタスクを再起動する。

ここでタスクが一時切り離されるまでの時間は、下記の様に実行時オプションF_PMTUNE で変更可能です。 並列処理のベンチマークテストなどを行う時は、大きな値に変更すると若干経過時間の向上が得られる時がありますが、通常は既定値(50 ミリ秒)のままで問題ありません。

(100ミリ秒に変更する例)

(sh)

```
# F_PMTUNE=100000
# export F_PMTUNE
```

(csh)

```
% setenv F_PMTUNE 100000
```