CommonMP 利用における C#による実行上の課題と DLL の活用

東京建設コンサルタント　正会員　○渡辺明英
東京建設コンサルタント　曽田康秀

１．はじめに

CommonMP は C#から要素モデル DLL を呼び出すため，要素モデルは C#がベースとなる．一般に，C#は C++よりも遅く，C++はプログラムによっては C や Fortran より遅いとされる．特に C#の 2 次元以上の非定常問題で実行速度に問題が生じる様である．本文では，C#や C, Fortran による native な実行 code，C#から Fortran-DLL 呼び出しについて実行速度テストを行い，効率的かつ高速に計算するためには，要素モデルの計算部分のプログラムがどうあるべきかについて検討する．

２．実行速度テストの条件と方法

C#での実行速度を評価するために，**表−1** に示す様々な CPU，OS，コンパイラ（version）の組合せで計測されているが，傾向は同様であったので，主要な 2,4,5,6 についてのみ結果を示す．テストには，流体解析に多く見られる時間発展型の 2 次元偏微分方程式の代表として，式(1)に示される拡散方程式が用いられた．演算スキームは**図−1** に示される通りであり，浮動少数点演算回数は(loop 長 x16FP)/time-step とした．境界条件は**図−2** に示される通りであり，K は等方性だが不透過壁が内在する．計算は倍精度(8byte)で行っており，速度は通常，単精度に比べ CPU で 1/2，GPU で 1/8 に低下する．DLL 化にあたって，Fortran 引数は全て参照渡し，2 次元以上の配列で C や C#とは low と column が逆になることに注意が必要である．

実際の計算速度 FP_{real} は，[計算量/経過時間]で算出される．経過時間には計算時間の他，データ転送，over head，通信等の時間や loss-time が含まれ，式(2)で表される．式(2)からわかるように，実速度は計算速度とデータ転送速度の遅い方で決まる．時間発展型偏微分方程式に対して，現行機の多くではメモリ帯域依存の律速状態（計算時間より，メモリから CPU への data 転送時間が長い状態）になり易いことが知られている．計算速度は使用するデータ量（配列の大きさ）に依存するので，配列要素数を変えて計算時間を計測し，計算速度を求めた．試行錯誤の結果，計測誤差は概ね 10%程度である．

３．実行速度テスト結果

図−3 は，テスト環境 2,4,5,6 における実行速度を配列要素の Memory Size に対して示したものである．ここで，gnu 系 [1]の compile option[2]はそれぞれ，-O0:最適化なし，-O2:主に cache 関係の最適化，-O3，-Ofast:その他 loop の最適化に加えて vector(SIMD:SSE,AVX)化である．Intel compiler による openmp 並列(Linux)と PGI acc による GPGPU 並列(Linux, windows)の結果も比較して表している．**図−3** より,環境 2(1 core で実行)で速度は明らかに，C# << 最適化及び AVX が有効状態の[C, C#&DLL, Fortran]である．C と Fortran 及び DLL の速度は code や compiler に依存するが，この環境下でほぼ差はない．しかし，C#と Fortran 他では 7 倍-10 倍程度速度が異なる．最適化なしの Fortran と C#での速度がほぼ同一であり，C#上で unroll や block 化を直接 coding しても速度に変化はなかったことから，C#の低速度は cache 関連の最適化無効に起因していると推定される．Visual Studio の最適化 check-box では速度改善は見られない．一方，command line から [/optimize+]を指定する [3]と，C#でもわずかに改善はあったが，gnu 系-O1 にもかなり劣る程度であった．適切な option で compile された Fortran-DLL を用いた場合に速度改善が見られたため，計算速度に問題がある場合には C#にこだわる必要はなかろう．

Linux 上での並列計算[最適化+ベクトル化+openmp](環境 4)や GPGPU(環境 5,6)計算と比べれば，C#とそれらとの速度には 50 倍-100 倍近い開きがある．分布型 model の様な大規模計算で，GPGPU は選択肢の 1 つとなろう．データ量が少ない範囲で並列計算に十分な速度が出てないのは，並列化のための over-head によるものとされている．20Mbyte を越えた辺りで CPU の並列計算速度が急減するのは，計算データ量が cache 容量を大きく上回り，CPU レジスタへの実質のデータ転送速度が落ちたためである．この解消には，[CPU 数/node]を増やし[cache/node]を上げるか(MPP 等)，node 間 MPI 通信を活用して[計算量及びデータ量/node]を減らすか(クラスター化)，帯域を上げる必要がある．簡単に帯域を上げる手段の一つが GPGPU(GDDR5)の利用である． 高速計算機，いわゆるスーパーコンピュータではこれらの方法が併せて採られている．

キーワード　Common MP, C#, C/C++, Fortran , DLL，実行速度，メモリ帯域
連絡先　〒170-0004 東京都豊島区北大塚 1-15-6 (株) 東京建設コンサルタント環境防災研究所 TEL: 03-5980-2633

4．C# における高速化と DLL の活用

演算部を Fortran/C で DLL 化[4)5)]すれば，C#の要素 model はある程度の高速化は可能であると思われる．ただし，十分な性能を引き出すためには，プログラムが構造化されている必要があり，行番号で制御されている既存の F77 プログラムをそのままるごと DLL 化するよりは fortran90 以降の文法形式で書き換えた方が良い．手間がかかることになるが，一度 C#に書き換えてから，遅い演算メソッドを改めて C か Fortran で DLL を作成し直して，どちらででも実行可能にする方が望ましい．

Win32API 仕様で作成された DLL は，Windows 上であれば CommonMP 要素 model や C#だけでなく言語に依らずに利用できる．このため，今後の発展を望むのであれば DLL の活用が望ましいが，CPU/OS/compiler & option(特に SIMD 関係部分)によっては，動作が確実ではないのでバイナリ配布する場合には注意が必要である．計算の高速化を目指すのであれば，DLL を介した GPGPU(アクセラレータ)の利用や通信による外部演算サーバの活用なども視野に入れる必要があろう．

表−1 ベンチマークテスト環境（［*1:倍精度では 1/8 相当］，コンパイラの数字は version を示す．）

	core name	code name (発売年)	clock GHz	cpu 数	core 数	register 数 32bit	cache MB	帯域 GB/s	Inter-connect	OS	Compiler				
											gnu	g95	intel	pgi	C#
1	Core2 Duo	Penryn (2008)	2.4	1	2	4 (x2)	3	8.3	FSB	XP	3.4.5	0.93		13	2010
2	Core i5	Ivy Bridge (2012)	2.4TB	1	2	8 (x2)	3	25.6	QPI	W-7	4.7.1				2010
3	Xeon X5482	Harpertown (2007)	3.20	2	4(x2)	4 (x8)	6(x2)	21.3	FSB	Linux			9		
4	Xeon X5680	Westmere-EP (2010)	3.33	2	6(x2)	4 (x12)	12(x2)	64.0	QPI	Linux			11		
5	GTX570	Fermi (2010)	1.46	-	-	480c *1	-	152		Linux				12	
6	GTX670	Kepler (2012)	0.98			1344c*1		192.		W-7				13	

$$\frac{\partial C}{\partial t}=\frac{\partial}{\partial x_i}(K_i\frac{\partial C}{\partial x_i})+f \quad \cdots (1)$$

C：濃度等の変数，K_i：拡散 *or* 透過係数，

f：sink－source項，

x_i：座標軸($i=1,2$)，t：時間

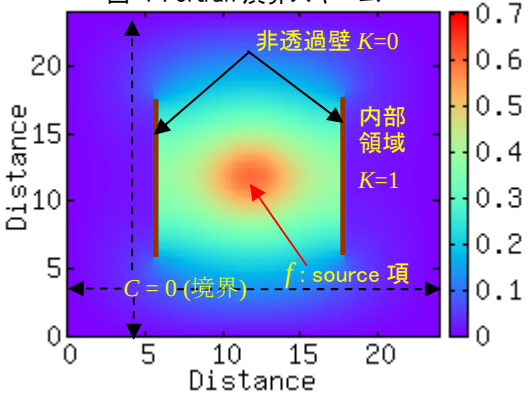
$$FP_{real}=\frac{Flop}{Flop/F_{peak}+Byte/B_{peak}+\varepsilon}=\frac{Flop/Byte}{Flop/Byte+F_{peak}/B_{peak}+\varepsilon'}F_{peak}\cdots(2)$$

［計算時間］ ［転送時間］

$$\begin{cases} F_{peak}/B_{peak}\ll 1 \Rightarrow FP_{real}=F_{peak}: CPU律速状態 \\ Flop/Byte \gg F_{peak}/B_{peak} \Rightarrow FP_{real}=F_{peak} \\ F_{peak}/B_{peak} \gg Flop/Byte \Rightarrow FP_{real}=(Flop/Byte)B_{peak}: メモリ律速状態 \end{cases}$$

```
!$omp parallel private(i,j)↓
!$omp do↓
      do j=1,jmax↓
      do i=1,imax↓
      du(i,j) = ( kx(i+1,j)*(uu(i+1,j)-uu(i  ,j))↓
     &           -kx(i  ,j)*(uu(i  ,j)-uu(i-1,j))) * ddx2↓
     &         +( ky(i,j+1)*(uu(i,j+1)-uu(i,j  ))↓
     &           -ky(i,j  )*(uu(i,j  )-uu(i,j-1))) * ddy2↓
     &         +  ff(i,j);↓
      end do; end do↓
!$omp end do ↓
!$omp do↓
      do j=1,jmax↓
      do i=1,imax↓
      uu(i,j) = uu(i,j) + du(i,j)*dt;↓
      end do; end do↓
!$omp end do ↓
!$omp end parallel ↓
```

図−1 Fortran 演算スキーム



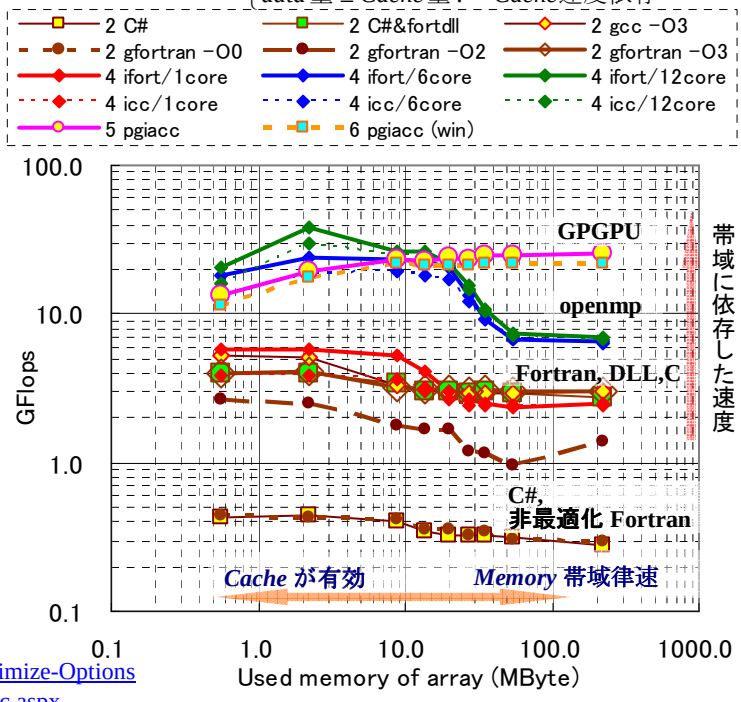
図−2 計算領域と境界条件（境界4辺単純支持）

$Flop$(計算量) $=(I_{max}\times J_{max})\times$浮動少数点演算回数$/Loop$

$Byte$(データ量) $=(I_{max}\times J_{max})\times$変数の数$\times Byte$数$/Loop$

F_{peak}(最大計算速度) $=[CPU$数$/node][core$数$/CPU][register$数$/core]$
$\times[FP$回数$/clock][clock$周波数$(GHz)]$

$$B_{peak}(\text{最大転送速度})=\begin{cases} data\text{ 量}>Cache\text{ 量}: & Memory\text{帯域依存} \\ data\text{ 量}\le Cache\text{ 量}: & Cache\text{速度依存} \end{cases}$$



図−3 各計算環境下における計算速度の比較結果

(参考資料等)
1) Windows 用 GNU コンパイラ http://tdm-gcc.tdragon.net/
2) http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options
3) C# コンパイルhttp://msdn.microsoft.com/ja-jp/library/t0hfscdc.aspx
4) DLL 関係 http://fortranwiki.org/fortran/show/C%23+Interoperability+
5) DLL 関係 http://software.intel.com/en-us/articles/calling-fortran-function-or-subroutine-in-dll-from-c-code/