MPS 法への新規高速化手法の実装と評価

宮 島 敬 明

理化学研究所　計算科学研究センター

１． はじめに

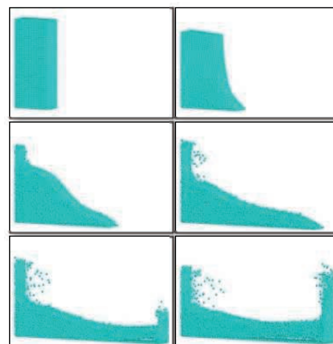

図 1:シミュレーション例

Moving Particle Simulation (MPS) 法は、水などの大変形を伴う非圧縮性流体を解析対象にしている。MPS 法は粒子系シミュレーションに分類され、計算対象を多数の仮想粒子として分割し、各粒子と近傍粒子との相互作用から物理量の計算を行う。大規模解析に対応すべく、近傍粒子の探索処理をスレッド並列化し、計算領域を複数プロセスに動的に分割して処理時間の短縮を図っている。本年度は、プログラム全体の GPU 化に取り組んだ。

２． 研究背景

1. MPS 法は大変形を伴う非圧縮性流体のシミュレーションに用いられ、粒子系シミュレーションに分類される [4]。空間を格子で分割し、各格子が物理量を保持するオイラー的な手法の格子法と異なり、移動する粒子が物理量を持つラグランジュ的な手法である。図 1 に MPS 法による水柱崩壊問題のシミュレーション結果を示す。各粒子の速度などの物理量は近傍粒子との相互作用を元に計算されるため、全粒子がそれぞれの近傍粒子を探索する必要がある。この近傍粒子探索の処理は、ランダムアクセスと間接参照、不定回ループなどを持ち、計算時間の大半を占める。MPS 法の計算精度は使用する粒子数によって大きく左右される。室谷らの京を持ちいた先行研究では、 4.0km × 3.5km の津波のシミュレーションに 2 億 6 千万個の粒子を利用している [5]。各粒子が独立して近傍粒子を探索し相互作用を計算するため、並列度は粒子数に比例して増加するが、同時に計算負荷も増加する。その結果、シングルノードでの計算時間の大半が近傍粒子探索となり、この部分の高速化が MPS 法全体の高速化の鍵となる。 MPS 法の GPU への実装はいくつかの先行研究が存在する [9]。

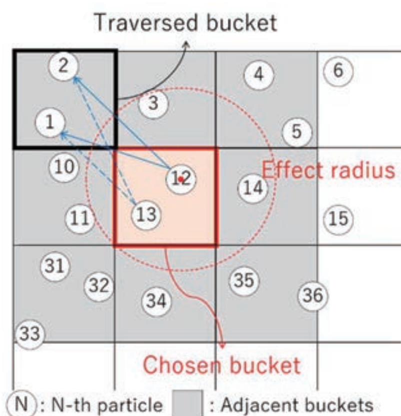

図 2: バケット法を用いた近傍粒子探索

陽解法 MPS 法は、粒子系シミュレーションの 1 つであり、水などの非圧縮性流体を多数の仮想粒子に分割し、各粒子と近傍粒子との相互作用から物理量の計算を行う。大規模解析と計算精度向上のためには粒子の個数を増やす必要があるが、粒子数に比例して近傍粒子を探索する処理（近傍粒子探索）も所要時間が大幅に増加する。図 2 に示すように近傍粒子探索にバケット法を採用した上でスレッド並列化し、計算領域を複数プロセスに動的に分割して処理時間の短縮を図っている。Reedbush-H を用いたこれまでの研究で、GPU 化と領域分割により近傍粒子探索は高速化されたが、動的領域分割とそれに伴う不規則な通信がボト

ルネックとなることがわかっている。

３． 本年度の目標

本年度は以下の問題の評価を行い、原因を探った。

（ア） Managed memory 機能を使った完全 GPU 化

これまでは近傍粒子探索を行うサブルーチンのみを GPU 化の対象としていた。実際の大規模シミュレーションを行うには、全体の移植が必要である。手始めに Reedbush-H に搭載されている P100 の Managed memory 機能を使い、全体の GPU 化を行う。データ転送の最適化は今後の課題とする。

４． MPS 法とデータ構造

ここでは内製 MPS 法の全体像と採用しているデータ構造について述べる。

４． １ 支配方程式

MPS 法は以下の計算ステップを目的のシミュレーション時間に到達するまで繰り返す。なお、$r_j - r_i$ は対象粒子 i と近傍粒子 j との距離計算を示す。

- ・Step 0） 計算領域の初期値を設定
- ・Step 1） 仮速度の計算
- ・Step 2） 仮位置を計算
- ・Step 3） 粒子数密度と圧力の計算
- ・Step 4） 圧力勾配の計算
- ・Step 5） 粒子の位置を計算
- ・Step 6） 次の時間ステップへ（Step 1～6 を繰り返す）

以下に各計算ステップの概要を示す。

Step 0） シミュレーションの初期値を設定

MPS 法では初期値として、近傍粒子の距離の重み平均λ^0と初期の粒子数密度n^0 を求める。

$$\lambda^0 = \frac{\sum_{j\neq i'}(|r_j^0 - r_{i'}^0|)^2 \omega(|r_j^0 - r_{i'}^0|)}{\sum_{j\neq i'} \omega(|r_j^0 - r_{i'}^0|)}$$

$$n^0 = \sum_{j\neq i'} \omega(|r_j^0 - r_{i'}^0|)$$

Step 1） 仮速度の計算

計算ループでは、まず下式を用いて各粒子の仮速度を求める。右辺 第 2 項と第 3 項はそれぞれ粘性と重力である。

$$u_i^* = u_i^k + \Delta t\left\{\left(v\frac{2d}{\lambda^0 n^0}\sum_{j\neq i}(|u_j^k - u_{i'}^k|)\ \ \omega(|r_j - r_i|)\right) + g\right\}$$

ただし、 k はタイムステップ、 t は実時間、 i と j は粒子の番号、 ν は動粘性率、 d はシミュレーションの次元数、 g は重力加速度、 u_i^kは時刻 k での粒子 i の速度、 u_i^* は時刻 kでの粒子 i の仮速度である。

Step 2）仮位置を計算

続いて、 Step 1 で得られた仮速度を用いて各粒子の仮位置を求める。

$$r_i^* = r_i^k + \Delta t\, u_i^*$$

ただし、 r_i^k は時刻 k での粒子 i の位置である。

Step 3) 粒子数密度と圧力の計算

その後、各粒子の次のタイムステップの圧力を求める。

$$n_i^* = \sum_{j \neq i} \omega(|r_j^* - r_{i'}^*|)$$

$$P_i^{k+1} = c^2 \frac{\rho^0}{n^0}(n_i^* - n^0)$$

ただし、 Pik+1 は時刻 k での粒子 i の圧力、 c は音速、 n_i^*は時刻 k での粒子 i の仮の粒子数密度である。

Step 4) 圧力勾配の計算

そして、各粒子の圧力から圧力勾配を求める。

$$\langle \nabla P \rangle_i^{k+1} = \frac{d}{n^0} \sum_{j \neq i} \left(\frac{(P_j^{k+1} - P_i^{k+1})(r_j - r_{i'})}{|r_j - r_i|^2} \omega_{grad}(|r_j - r_i|) \right)$$

ただし、 ω_{grad}は後述する重み関数である。

Step 5) 粒子の位置を計算

最後に、次のステップの最終的な速度と位置を求める。

$$u_i^{k+1} = u_i^* - \Delta t \left(\frac{1}{\rho} \nabla P \right)_i^{k+1}$$

$$r_i^{k+1} = r_i^* - \Delta t \left(\frac{1}{\rho} \nabla P \right)_i^{k+1}$$

ただし、 u_i^{k+1} と r_i^{k+1}は時刻 $k+1$ での粒子 i の粒子の速度と位置である。

Step 6) 次の時間ステップへ

次のタイムステップの計算を開始するために、粒子の最大速度uから実時間Δtを求める。

４．２ データ構造

我々は、連結リストを用いたデー構造を採用している。これは、 MPS 法をプロセス並列化し、大規模シミュレーションを行う際に必須となる。特に、負荷分散のために計算領域を自由な形状で分割しなければならないことが要因である。 MPS 法は粒子が移動するため、計算領域の中で粒子の偏りが発生し、各計算ノードで負荷が一定にならない。負荷を分散させるために、グラフ分割を応用して計算領域を分割する手法が知られており、我々も同様の方法を採用している。分割された計算領域は、矩形以外の複雑な形状を取るため、隣接するバケット同士の関係は柔軟に変更できなければならない。また、隣接するプロセスが隣のバケットを計算しているとは限らない。インデックスを用いて隣接するバケット同士の関係を表現する場合、矩形以外の形状に対応することが難しい。また、隣の計算領域をどのプロセスが担当しているかを把握することも難しい。連結リストを用いて隣接するバケット同士を表現することで、形状の変化に柔軟に対応可能である。加えて、隣のバケットをどのプロセスが計算しているかを管理することが容易である。

バケット１つは、連結リストと内部の状態を保持する構造体として定義されている。内部の状態を保持する構造体は、バケット内に存在する粒子の数や粒子の物理量へのインデックスを持つ。このインデックスをもとに物理量を保持する配列へアクセスする。

５．OpenACCによるデータ構造のGPU実装

OpenACCはGPU向けのプラグマベースのコンパイラである。CPU側のデータ構造をGPU側で利用出来るようにするには、Deep Copyという問題を解決する必要がある。Deep Copyは、CPU側のポインタがGPU側では使えないという問題で、ポインタを使ったデータ構造を採用している場合には実装上の大きな問題となる。Managed memoryの機能を利用してもこの問題は解決できない。この問題に対処するために、OpenACCではdeclare dataクローゼやupdate dataクローゼ、declare linkクローゼが用意されている。なお、公開されている資料は非常に少なく、実装には手探り感が否めない。

GPU化にあたり、連結リストをポインタではなくバケットの番号とすることで、ポインタの使用を回避した。ポインタの使用を回避することで、managed memoryを利用することができ、移植性は大きく向上した。参照の回数も変更はないと思われる。PGI Compilerに”-ta=tesla:managed”のオプションを付与し、managed memoryを有効にして、データの比較を行った。その結果、GPUとCPUで結果が一致することが確認された。

６．まとめ

本年度は、データ構造のGPU化を行い、Managed memoryを用いて内製プログラム全体のGPU化を行った。今後は、Managed memoryを使わずに、OpenACCのdataディレクティブを用いた実装を行う予定である。