1　問題

二粒子間の重心間距離で決まる法線方向の斥力/引力（JKR theory）に加え、粒子のスピンに関連した様々な剪断応力も考慮した N体計算コードの OpenMP による並列化を試しているがシリアルコードに比べ実時間は短縮されない。（計算時間の9割を占める「接触ペアの相互作用計算」部分などを OpenMP にて並列化した。）加速しない原因や改善策についてアドバイスを頂けると助かります。

2　効率が上がらない原因として考えられる事象（一般論）

2.1　メモリへのランダムアクセス

ランダムなメモリアクセスが多い場合が多い場合は、メモリからの読み出し、メモリへの書き込みがボトルネックとなる場合があります。N体計算ということですので、各粒子の位置や速度の情報を得るためにメモリに対してランダムアクセスを行っていると想像されます。その場合、キャッシュを有効に使えないため、アクセスの度にメモリ本体へのアクセスが必要となり、ここがボトルネックとなります。すると、いわゆるアムダールの法則において、「並列化できない部分」が大半を占めることになり、OpenMP による並列化が意味をなさない可能性があります。

2.2　スレッド生成・解放に伴うオーバーヘッド

スレッドの立ち上げ・解放が繰り替えされる場合、オーバーヘッドの割合が大きくなり、全体の並列化効率が向上しない場合があります。`!$OMP parallel` 構文を使用すると、その内側のブロックでスレッドの立ち上げと解放が行われますが、このプロセスには一定の時間を要します。一組の立ち上げ・解放には実感できるほどの時間を要しませんが、`!$OMP parallel` 構文が多数存在する場合（ループの内側にある場合や、`!$OMP parallel` ブロック自体が多数存在する場合など）は、このプロセスが非常に多い回数繰り返されるため、やはりここがボトルネックとなって並列化効率があまり向上しません。`http://www.jicfus.jp/field5/jp/wp-content/uploads/2011/09/Freja.pdf` の4章で、単純にループの内側（正確にはループから呼び出される関数の中）に `!$OMP parallel` を記述した場合と、`!$OMP parallel` はプログラム中で一回だけ呼び出されるように改良した場合（SPMD的なコーディング）とでの並列化効率の違いを議論しています。ここで行われたテスト計算によりますと、前者の場合、core数の増加に伴って効率が急激に悪くなることが分かっています。

2.3　原理的な困難

すべての粒子ペアについての相互作用が求まったあと、相互作用 f_ij を粒子 i と粒子 j に書き戻す作業がどうしても必要になりますが、この箇所はマルチスレッドで処理しようとするとメモリの書き込み競合が発生するため、正しく動いてなおかつスケールするような実装を作るのはかなり困難と思われます。

3　プロファイリングに関する注意点

3.1　公平な測定

依頼者ご自身による事前のベンチマークテストでは、シングルスレッドでのテスト時のみ `-ipo -no-prec-div` 等の最適化オプションを使用して計算時間の測定を行われていました。スレッド数を変えてベンチマークテストを行う場合は、すべての場合について同じ条件下で行わなければ、並列化の効力を正しく見積もることができません。

3.2　詳細な測定

単に全体の実行時間やプロファイラを取るだけでなく、OpenMP ディレクティブで囲まれた箇所の時間をできるだけ分解能の高いタイマー（FORTRAN なら `MPI_Wtime()` など）で測定してみるのがいいと思います。

4　メモリアクセス向上に関する提案

4.1　連続的なワークスペース

もしメモリに余裕があるならば、相互作用している粒子の情報をワーク スペースにコピーし、そこで相互作用計算を行うことでキャッシュミスを遙減 できると期待されます。一方で、相互作用状態にある／ないの切り替えが頻 繁に行われる場合は、データをコピーするオーバーヘッドが大きすぎて、速度が低下こと も懸念されます。

4.2　データの連続性（構造体の利用）

粒子の様々なデータを保管する巨大な配列が複数存在しており、ループインデッ クス k でこれらを参照している（`A(k), B(k), ...` のように）。キャッシュ ミスを低減する方法として、`array(1,k) = A(k), array(2,k) = B(k), ...` のように組み直すと効果的な場合があります（構造体を用いることも可能です）。この場合、インデックス k による参照で、当該粒子の各種データ `A,B,...` が一挙にキャッシュに入る可能性があります。

4.3　粒子番号の並べ替え

粒子や相互作用している粒子対の番号を適度なタイミングで付け替えること で、メモリ上のデータをできるだけ連続的に配置することができます。この措置は、「相互作用していた粒子のうち1ステップ後にどの粒子 とも相互作用しなくなる粒子数」が「現在相互作用をしている粒子数」に 対して十分少ない場合に、より効果的になると思います。

5　ロードバランス

5.1　提案1

メインルーチンの入り口に「もし粒子ペアが相互作用状態にあるならば以下の相互作用 を計算する」という分岐があり、false の場合、ルーチンのほとんどが読み飛ばされる 構造になっています。ペアの相互作用状態の on/off を記述した配列の分布が どのようになっているかに依りますが、これに著しい偏りがある場合は、CPU core の負荷のバランスが悪く、並列化の意味が薄れる場合があります。もしそうならば、ループインデックス k をブロック分割（$1 \sim 25, 26 \sim 50, \ldots$ のように）するのではなく、剰余で分割（$k \equiv 0, 1, 2, 3 (\mathrm{mod} 4)$ のように）すると効果があるか もしれません。

5.2　提案2

相互作用がとても疎な粒子系では、

```
for each pair(i, j)
evaluete_force_between(i, j)
```

のようになっていれば、メモリの間接参照を除けば、この部分のループはごく単純な OpenMP で良くスケールするはずです（データとして、`int pair[NPAIR][2];` みたいなものを想定しています）。ただし、ペアごとの計算負荷が大きくばらつく場合、あるいは NPAIR 個あるのは相互作用するペアの候補であって実際に相互作用するのはそのうちのごく一部である場合、`schedule(dynamic)` や `schedule(guided)` を使う必要が出てくるか もしれません。

6　リダクション処理

ペア相互作用計算の最終段階でそれぞれの力の合力を計算する部分は、通常の reduction 処理（`!$OMP DO REDUCTION (+:SUM)`）で対応できるかもしれません。足し込む先の配列要素は粒子番号で指定されるため、「番号の付け替え」と合わせて並列化を行う必要があると見込まれます。しかし、合力計算の処理時間は相互作用計算ルーチンの中でも CPU 時間を占め る割合が低い方と見込まれるため、並列化の労力の割に全体の効率はあまり上が らない可能性があります。