性能評価walkthrough

2013/9/17

野村@東工大

今日の内容

- 10月の性能調査ワークショップにて行う、TSUBAME2.5とScalascaを用いた演算数、メモリアクセス量、通信パターンの調査のやりかたを具体例とともに説明します
- 分からないところがあれば発表中に割り込んで質問してください
- 追加資料が以下のURLにあります
  - http://hpci-aplfs.aics.riken.jp/performance-analysis.html

Scalascaについて

- Julich Research CenterとGerman Research School for Simulation Sciencesにおいて作られたプロファイリングツール
- アプリケーションのCall Treeに沿って、関数毎、プロセス毎の実行回数、実行時間、通信回数、通信量、ハードウェアカウンタの増分を表示できる
- 簡単な画面の見方と操作方法は次頁(デモ)

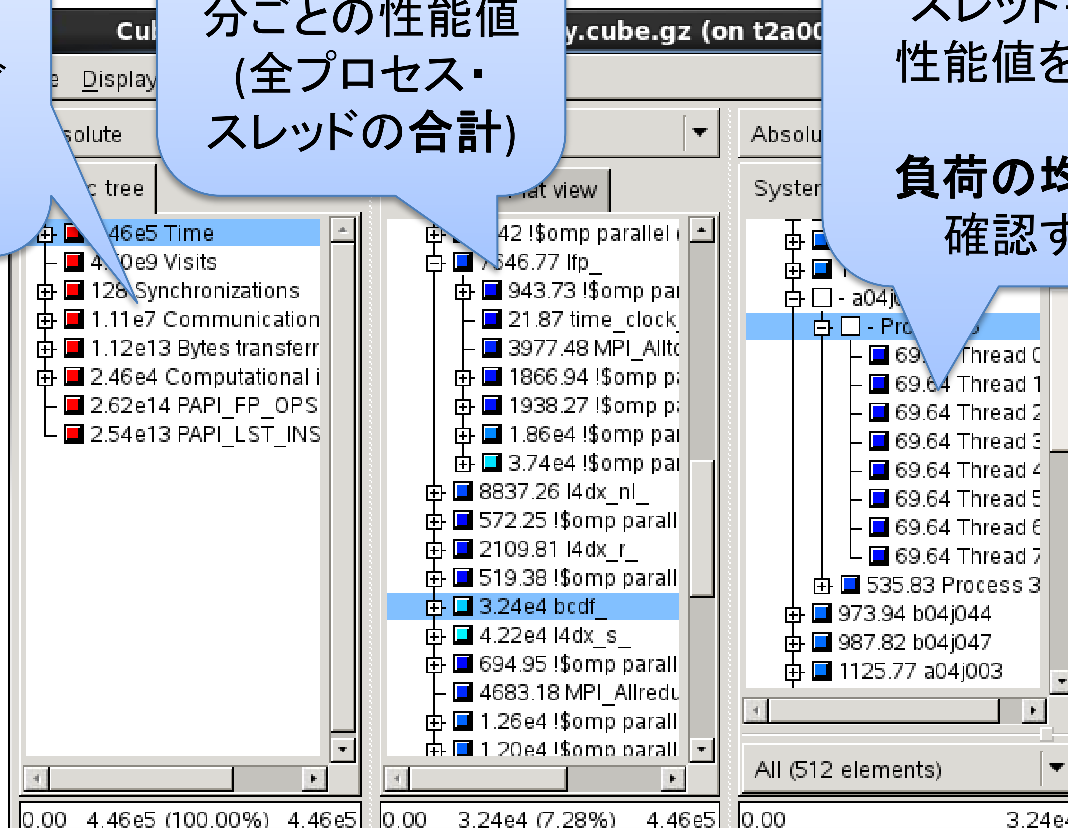
Scalas 画面と操

Metric
実行時間・
HWカウンタなど
「何を」見るか
切り替える

Call Tree
プログラムの部
分ごとの性能値
(全プロセス・
スレッドの**合計**)

Process Tree
各プロセス・OMP
スレッド毎の
性能値を表示

負荷の均衡を
確認する



共通事項
ツリーを展開するとExclusive(子の分は含まない)表示、
畳むとInclusive(子の分を含んだ合計)表示になる

Scalascaの使い方の実際例

- 以下、GT5Dを例に以下の一連の流れについて説明します
  - Scalascaで計測できるようにする
  - ホットスポットの同定
  - 演算数の確認
  - 通信パターンの確認
  - 実行条件間の比較(weak/strong scaling)

Scalascaで測定できるようにする

1. TSUBAME2.5で動くようにする
   - 10月のワークショップでは、ここまではできていることを前提
     - アカウントが発行されたら必ず事前に試してください
   - Intel Compiler 2013 + OpenMPI 1.6.3 を想定
   - Scalascaでの測定と実行時間を比較するために、1回素で実行する
2. Scalascaで計測用のコードを埋め込む
   - 自動instrumentation
     - 各関数やOMP領域に自動でコードを埋め込む
     - お手軽
     - 遅くなるかもしれない(計測コードを埋め込みすぎる, 特にC++)
   - 手動instrumentation
     - 計測したい区間を手動で定義し、
ソースコードに埋め込む
     - 必要最低限のオーバヘッドで済む
     - ソースコードの「癖」で引っかかる確率は減る

TSUBAME環境設定

- 以下の環境変数を設定してください
  - 詳細は10月のワークショップで
  - source setompi-1.6.3-intel.sh
  - export PATH=/work1/t2g-hp120261/fs_share_tool/scalasca-1.4.3/bin:/work0/GSIC/apps/papi-5.0.0/bin:${PATH}
  - export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/work0/GSIC/apps/papi-5.0.0/lib
  - export SCAN_SETENV="-x="

GT5DをTSUBAMEで動くように

- 詳細は省略
  - 実際にやるときには、利用の手引きやエラーメッセージを参照してください
  - 以下の点を修正しました
    - OpenMP指示行の指定を修正
  - バッチジョブスクリプトを作成
    - OpenMPIでは、全プロセスで用いる環境変数は明示的に指定する必要があります (例: mpirun … -x OMP_NUM_THREADS …)
  - 実行
  - 結果の妥当性の確認
    - Validation
    - 他のマシンと比較して極端に遅くないか

Scalascaによる自動instrumentation

- Makefileの変更
  - コンパイラの前に skin をつける
    - 例: fc = skin mpif90
- コンパイル
  - 特に問題が無ければプロファイリング用のバイナリが作成される

Scalascaに向けたGT5Dの修正(1)

- プログラム先頭に PROGRAM 文を挿入
  - 自動instrumentation時にPROGRAM文を目印にしている模様
- ENDの直前にあるSTOPを削除
  - STOPで止まると実行結果の収集をせずに終了してしまう
- プリプロセスを分離
  - プリプロセス前のコードではinstrumentorが混乱するようなので、プリプロセスをScalascaを使わずに行う
    - .f.o:

```
cpp $(cflag_cpp) $*.f > $*.fsapp.f
$(fc) –o $*.o –c $*.fsapp.f
```

Scalascaに向けたGT5Dの修正(2)

- OpenMP指示行の継続関係の修正
  - 空行が入らないように #ifdef を削る
  - $OMP&のあとに空白が入るようにする

- 詳細は本日の公開資料の
  プロファイラツール連携作業履歴 - GT5D
  をご覧ください

手動instrumentation

- こんなときはどうする?
  - 自動instrumentationでエラーになる上、何が原因か分からない
  - 自動instrumentationで動くんだけど、何もしないときの数倍時間がかかる
    - そんな状況でHotspotと言われても信用できない
  - 1関数のなかに色々な処理を入れすぎて、関数ごとの計測だとどこが重いか分からない
- 手動で計測関数を埋め込むことで回避
  - 自動の埋め込みを無効にする
  - 詳細は10月

今回の計測サイズ

- 問題サイズ
  - 小: (x, y, z, v, w) = (120, 120, 32, 64, 64)
  - 大: (x, y, z, v, w) = (240, 240, 32, 64, 64)
- MPI分割
  - 16ノード: (x, y, w) = (2, 2, 4)
  - 64ノード: (x, y, w) = (4, 4, 4)

Scalascaでのプログラム実行

- 計測用設定ファイルを書く(環境変数でも可)
  - EPIK.conf
    - EPK_TITLE=fs_gt5d
      EPK_METRICS=PAPI_SP_OPS:PAPI_DP_OPS:PAPI_VEC_SP
      ESD_BUFFER_SIZE=100000000

演算数をとるための設定

- 実行コマンドの前に scan を付ける
- mpirunのオプションのうち、-np以外のオプションをクォートする
  - scanがオプションの引数と実行ファイル名を区別できるように
  - 例 scan mpirun –np 64 –report-bindings
    –bysocket "–cpus-per-proc 6" –bind-to-core
    "–x OMP_NUM_THREADS" "–hostfile
    ${PBS_NODEFILE}" ./GT5D >& ./log.mpi
- ジョブを投入し、結果を待つ

プロファイル結果を見る

- X転送している環境で square epik_fs_gt5d/ を実行
  - 冒頭にお見せした画面が出てきます
  - あとは見たい項目をどんどんクリックするだけ
- で、どこを見ればいいの?
  - 次スライド以降でデモをしながら説明

Hotspotを捜す

- 実行時間が長いところを捜せばよい
  - TimeでCall Treeのうち、暖色のところを捜す
  - 項目をクリックすると、下の方に全体に占める割合などが表示されます
    - Ctrlで複数選択すると、選択したところの和になります(Coverage)

演算

- メトリック
  - 演算回数 (単精度・倍精度)
- 手段
  - ハードウェアカウンタの値をScalasca/PAPI経由で取得

- 計測時にハードウェアカウンタを取得しているので、そちらに切り替えればよい
  - Call Tree(中央)の値はあくまでも全プロセス合計であることに注意
  - Process Tree(右)を開くと、スレッド間のLoad balance具合を見ることができます
- Intel Xeon 特有の問題
  - SP, DP混合精度演算の場合、カウンタ値の補正が必要（詳細は別途資料）
  - SPのみ、もしくはDPのみであれば問題なし

メモリアクセス

- メトリック
  - メインメモリとCPUの間の通信量
- 手段
  - Scalasca/PAPIによるハードウェアカウンタの取得
  - 最外殻キャッシュ(TSUBAMEではL3)のミス回数と、L3のプリフェッチ回数を取得し、計算する

- 現在、IntelのL3プリフェッチ回数を求める方法を調査中
  - TSUBAMEでは、Linuxカーネルバージョンの問題で取得できない?
  - L2-L3間のアクセスで代用・FX10の利用などを検討中

他のサイズの実行結果と比較

- 複数の square(結果ビューア) を起動して、数字の変化を確認
  - 実行時間・演算数などが思った通りのオーダで増えているか
  - 実行規模を大きくしたときにホットスポットが変化していないか

通信

- メトリック
  - MPI関数の種類と回数、およびメッセージサイズ
  - MPI通信の宛先・Communicator
    - e.g. 隣接通信, Y軸方向のBroadcast, …
- 手段
  - ScalascaのCommunications, Bytes Transferredを参照
  - ソースコード上の各通信関数の通信先の確認

通信パターンの確認(1)

- まずは通信関数の列挙
  - MetricsでCommunicationを選択、Call Tree上で色がついている箇所を捜す
    - 前後関係や回数などから明らかに初期化・終了時の処理のもの以外は全部検討対象です
  - 通信待ち時間はMPI関数を選択状態でTimeへ
  - 通信サイズ(合計)はBytes Transferred
    - Communicationsの回数で割れば平均サイズに

通信パターンの確認(2)

- ソースコードで通信相手/Communicator確認
  - アプリ作成者の皆様は通信相手は分かっているはず
  - 通信量の変化についても、設計時の思惑通りのオーダか確認してください
  - 通信時間がノード間でばらついている場合、負荷の不均衡が原因のロスがあると予想できます
    - Computational Imbalanceを見ると原因箇所が見えると思われます

NICAM-DC(gravitywave)の場合

- デモ
  - 計測実行するところまでは省略
  - ホットスポット
  - 演算数
  - 通信