Multicore Programming Contest
Cell Challenge 2009
ツールキット解説書
ver.1.0対応版

Cell Challenge 2009
実行委員会

先進的計算基盤システムシンポジウムSACSIS2009併設企画
Multicore Programming Contest
Cell Challenge 2009

これはなに？

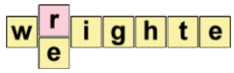
- Cell Challenge 2009規定課題に取り組む際のベースとなるツールキットの解説です
- ツールキットver0.1およびver0.5の解説も併せてご覧ください．

規定課題
「文字列の編集距離計算」

- 2つの文字列の近さを計る方法
  - かな漢字変換エンジンや，DNAの相同性検索などに利用される
- 2つの文字列の 編集距離
  - 片方の文字列から，もう一方の文字列を得るための操作回数の最小値
  - 使用可能な操作は以下の3種類
    - 削除：1つの文字を取り除く
    - 挿入：1つの文字を新たに付け加える
    - 置換：1つの文字を別の文字で置き換える
  - 参考になるURI「wikipedia:レーベンシュタイン距離」
    http://ja.wikipedia.org/wiki/%E3%83%AC%E3%83%BC%E3%83%99%E3%83%B3%E3%82%B7%E3%83%A5%E3%82%BF%E3%82%A4%E3%83%B3%E8%B7%9D%E9%9B%A2

編集距離の操作例

- 「weight」と「write」の編集距離
- 以下の操作で「weight」から「write」になる
  - ➢ weight
  1. weighte (挿入：e)
  2. wrighte (置換：e → r)
  3. wrihte (削除：g)
  4. write (削除：h)



- 3回以下の操作では「weight」を「write」できない
- よって編集距離は「4」
- 削除，挿入，置換は文字列の中のどの位置で行ってもよい

編集距離計算のアルゴリズム(概要)

- 「動的計画法(Dynamic Programming)」が有名

  文字列1　weight
  文字列2　write

- 操作回数を表を作って求める

編集距離計算のアルゴリズム(手順)

このスライドには
アニメーションが設定されています

		1	2	3	4	5	N=6
	0	w	e	i	g	h	t
1	w	0	1	2	3	4	5
2	r	1	1	2	3	4	5
3	i	2	2	1	2	3	4
4	t	3	3	2	2	3	3
5	e	4	4	3	3	3	4
M							

アニメーションでは列ごとに計算しているが，「上，左上，左のセル値」が決定しているセルならば計算手順は任意

1. 最左列，最上行のセルは1～N，1～Mの値と仮定
2. 上，左上，左のセル値と，当該の行と列の文字列を使い，以下の方法で各セルの値を計算する
   - A) 文字列が同一ならば左上セルの値
   - B) 異なるならば左上セルの値+1(置換)
   - C) 左セルの値+1(挿入)
   - D) 上セルの値+1(削除)
   - E) (A)-(E)の最小値をセルの値に設定
3. テーブルの左上から値を求める
4. 右下端のセル値が最終的な編集距離となる

アルゴリズムのブロック化

このスライドには
アニメーションが設定されています

	0	w	e	i	g	h	t
0							
w		0	1	2	3	4	5
r		1	1	2	3	4	5
i		2	2	1	2	3	4
t		3	3	2	2	3	3
e		4	4	3	3	3	4

Block1 Block3
Block2 Block4

- 編集距離計算はブロック化して部分ブロックごとに計算を行う
- 以下のデータがあれば，各ブロックの計算が可能
  - 文字列
  - 最左列，最上列の値
- ブロック計算の入力は以下
  - 部分文字列(各問題文字列中の該当部分)
  - 最左列，最上列のセル値
- 出力は以下
  - 最右列，最下行のセル値
- ツールキットでは，各ブロックの最右列，最下行の値をメモリのユーザ領域に格納，適宜読み出して使用する

new

ツールキット ver1.0

- 2つのテキストファイルを入力すると，それらの中の文字列を読み込んで編集距離を求める
- 複数のSPEを使用する
- 制約：各文字列の文字数は128の倍数
  - いろいろなサイズの例題ファイル付き
    (file1, file2, … , file20)
- getrndstr.c を使用すると，任意長のランダム文字列を生成できる

```
$ gcc -O3 -o getrndstr getrndstr.c
$ ./getrndstr 128 13 > file9999
```

乱数種13で生成される128文字の文字列を格納したファイルfile9999を生成する

実行方法(1/2)

- Makefileの「USERNAME」を各ユーザ名に編集

```
USERNAME = USERNAME
IDIR     = /export/home/$(USERNAME)/toolkit1.0
EXE_FILE = /export/home/$(USERNAME)/toolkit1.0/main
```

- コンパイル

```
$ make
/opt/cell/toolchain/bin/ppu-gcc -O3 -Wall -m32 -lspe2 main_ppe.c ppe.c -o main
/opt/cell/toolchain/bin/spu-gcc -O3 -Wall  spe1.c -o spe1
```

実行方法(2/2)

```
$ make run3
cellexec -t 30 /export/home/procon/toolkit1.0/main /export/home/pr
ocon/toolkit1.0/file7 /export/home/procon/toolkit1.0/file8 19168
execute command: /export/home/procon/toolkit1.0/main /export/home/
procon/toolkit1.0/file7 /export/home/procon/toolkit1.0/file8 19168
 timeout 30
answer = 19168
strnum(a)= 20480
strnum(b)= 20480
[SPE_] : 19168
[PPE_] : 19168
eclock  : 3.55188489
SUCCESSFUL.

execute time:      8978251 usec
```

実行バイナリ，第1ファイル，第2ファイル，答え
第1ファイル，第2ファイルの文字列長
SPE,PPEそれぞれの計算結果
計算時間

「答え」を与えないとPPEで検算を行います。
検算の時間は計算時間には含まれませんが、
タイムアウト時間には含まれます。

巨大な文字列を対象にした場合，計算時間が非常に長くなることがあります

modified!

規定課題への取り組み方

- 規定課題では，PPEプログラム(ppe.c)およびSPEのプログラム(spe1.c)を実装する
  - 以下のプログラムを実装してください
    - 編集距離を計算するPPE，SPEプログラム(ppe.c, spe1.cなど)
    - ppe_user()関数を実装
    - speプログラムはppe_user()関数内から呼び出し(ツールキット参照)
    - Makefile
      (SPEごとに別のバイナリを実行したい場合は修正が必要です)
  - 以下のファイルは変更が許可されません
    - PPEプログラム(main_ppe.c，define.h)

modified!

Regulation：メモリ領域の割当

- PPEからSPEに渡される変数について(先頭アドレス)
- 各文字列が配置されるメインメモリ上の領域
  - str1, str2　それぞれ1MB
- 距離計算結果格納領域(ans[0]に回答を格納すること)
  - ans　　128B
- 各文字列の長さnum1, num2

参加者は，上記を含めたPPE，SPEの各メモリを自由に使用できます

ツールキットver0.1の計算手順

- SPE1個のみ用いて計算する(PPEのプログラムは未変更)
- 各文字列に対して，128文字を1ブロックとして計算を行う(各文字はchar型)
  - ブロックの計算前に，以下をSPEのローカルストアに読み込む
    - ブロックの再左上セルの値(tlm : top-left-most)
    - ブロックの最左列(vbuf)，最上行(hbuf)を読み込む
    - 部分文字列
  - 各SPEが処理する表は，1ブロックあたり128x128x4B=64KB
  - 計算後，表の最右列(vbuf)，最下行(hbuf)を書き出す
  - 回答(最終ブロック最右下セルの値)をメインメモリ上のbuf4[0]に書き込む(buf4はver1.0ではansに名称変更)

プログラムコードにもコメントがあります

ツールキットver0.5の計算手順

- ツールキットver0.1に加え，SPEの処理を開始した後に，参加者がPPEで処理できるようコードを修正(ppe.cのppe_user()関数で処理が可能)
- 参加者が変更可能(不可能)なファイルの明確化
  - 修正不可なファイルはmain_ppe.c, define.hのみ

ツールキットver1.0での計算

new

ツールキットver1.0の計算手順

- SPEを複数使用し編集距離を計算するコードの例
  - main_ppe.c, define.h以外のファイルは編集可能
  - 各SPEでの計算は，ver0.5までと同様，128文字を1単位とするブロックで行う

new

制約の確認

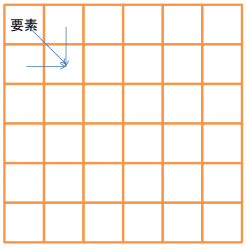
- 文字数は128の倍数(かつchar型)
- 問題で与えられる2つの文字列をそれぞれ文字列A，文字列Bと呼ぶことにする
- コードはlibspe2準拠
- PPEでも処理のプログラムを書いてよい
- SPEは7つまで使用可能
- PPEのメモリは自由に確保してよい

new

問題へのアプローチ

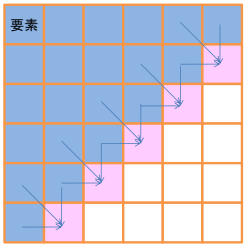
- ver.0.5でやったような文字列をブロックに分割すると，処理単位をある程度の大きさに分けることができる
- ブロックの計算を各SPEで並列に実行できると速そう！
- どこが並列計算できるんだろう？

new 並列計算できる部分の確認(1/3)



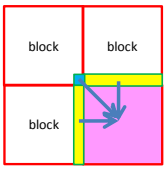
- データの依存関係
  - スコアテーブルの各要素は，左上，上，左の要素の値が定まっていると計算できる

new 並列計算できる部分の確認(2/3)



- 水色が計算済みであれば赤色は並列に計算可能
- これはブロックごとでも適用できる！

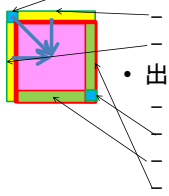
new 並列計算できる部分の確認(3/3)



- ブロックの場合は
  - 左上のブロックの(一番右下の)要素
  - 上のブロックの要素行 (最下行)
  - 左のブロックの要素列(最右列)

  の情報があれば計算できる

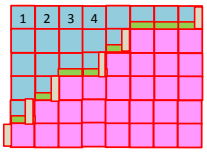
new ブロック計算の入出力



- 入力
  - 左上のブロックの(一番右下の)要素
  - 上のブロックの要素行 (最下行)
  - 左のブロックの要素列(最右列)
- 出力
  - 計算後のスコアテーブルの. .
  - (一番右下の)要素
  - 最下行
  - 最右列

new メインメモリの使用法(1/2)

- ブロック計算の中間データを別々に保持しておくとデータ量が非常に大きくなってしまうので，使いまわすことにする

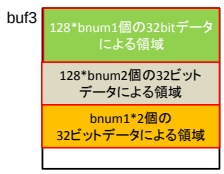


- 右下要素はbnum1 x 2個の領域を使いまわす
- 最下行は1つ上のブロックの計算，最右列は1つ左のブロックの計算以降は使わないので，それぞれ文字列A，Bのブロック数だけ保持しておけば良い

保持しなければならない最下行
保持しなければならない最右列

new メインメモリの使用法(2/2)

- buf3中に各ブロックの出力を保持する領域を決め，データを保持することにする



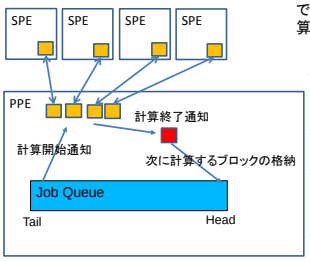
- 文字列Aがbnum1ブロック，Bがbnum2ブロックに分けられるとすると，左図の通り

保持しなければならない最下行
保持しなければならない最右列
各ブロックの右下の要素
(処理の開始終了フラグと一緒に転送する)

new

並列処理の方法

- 各SPEにどうやって処理を投げる？

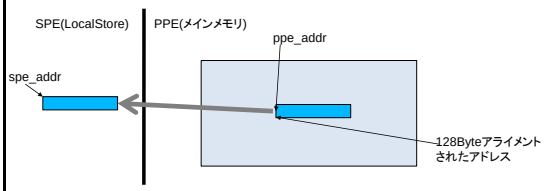


- PPE上でジョブキューを作り，各SPEはフラグを見て処理の開始を知り，そこで指定されたブロック番号に従い，計算対象のブロックの計算を行う

  PPEでの処理
  - 計算終了したブロック番号を元に，計算が可能になるブロック番号をキューに追加
  - 基本的に右と下のブロックが計算可能ブロックの候補になる
  - 処理が空のSPEに対し，キューから読み出したブロック番号を通知し，計算させる
  - 右下のブロックの計算終了まで続ける

補足)DMA転送用サブルーチン

- 転送用の命令
  - dmaget, dmaput : ツールキットで使用されるDMA読み書き関数
  - dmaget((void*))spe_addr, ppe_addr, X);
    メインメモリ領域のアドレスppe_addrを先頭としてXByteを読み出し，LocalStoreのアドレスspe_addrから始まる領域に格納する．dmaputは逆方向．



おわり