ParODE - Parallel linear ODE solver

Alexandru Patriciu

February 26, 2014

Chapter 1

Introduction

ParODE is a small package for the parallel simulation of linear dynamic systems. It can called from Scilab, python, or C/C++.

Currently the solver implements Runge-Kutta and Adams-Bashforth-Moulton methods.

Contact Information:

- Developer and Maintainer: Alexandru Patriciu
- email: apatriciu@gmail.com

Chapter 2

4^{th} order Runge-Kutta algorithm

The basic Runge-Kutta algorithm is

Algorithm 1 Runge-Kutta algorithm 4th order. Solve Equation $\frac{dy}{dx} = f(x, y)$, h is the step size. Initial state $y(x_0) = y_0$

$k \leftarrow 0$
while $k < NSteps$ **do**
  $k_1 = hf(x_k, y_k)$
  $k_2 = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}k_1)$
  $k_3 = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}k_2)$
  $k_4 = hf(x_k + h, y_k + k_3)$
  $x_{k+1} = x_k + h$
  $y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
end while

For a linear system the differential equation is $\dot{y} = Ay + Bu(t)$ ($x \equiv t$). We assume that the initial state is $y(x_0) = y_0$, the size of the system is n and the size and the size of the input vector is m. The previous formulas become.

The cost per iteration is $4n^2 + 4nm + 15n$. The total cost for $NSteps$ is $NSteps(4n^2 + 4nm + 15n)$

The serial algorithm can be converted into a parallel algorithm by expanding the formulas for k_1, k_2, k_3, and k_4 and replacing them in y_{k+1}. The stepping formula becomes

$$t_{k+1} = t_k + h \tag{2.1}$$

$$y_{k+1} = \overline{A}y_k + \begin{pmatrix} B_1 & B_2 & B_3 \end{pmatrix} \begin{pmatrix} u(t_k) \\ u(t_k + \frac{1}{2}h) \\ u(t_k + h) \end{pmatrix} \tag{2.2}$$

Algorithm 2 Runge-Kutta algorithm 4th order. h is the time step size.

$k \leftarrow 0$
while $k < NSteps$ **do**
  $k_1 = hAy_k + hBu(t_k)$; FLOP : $n^2 + nm + 2n$
  $k_2 = hAy_k + \frac{1}{2}hAk_1 + hBu(t_k + \frac{1}{2}h)$; FLOP : $n^2 + nm + 2n$
  $k_3 = hAy_k + \frac{1}{2}hAk_2 + hBu(t_k + \frac{1}{2}h)$; FLOP : $n^2 + nm + 2n$
  $k_4 = hAy_k + \frac{1}{2}hAk_3 + hBu(t_k + h)$; FLOP : $n^2 + nm + 2n$
  $t_{k+1} = t_k + h$; FLOP : 1
  $y_{k+1} = y_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$; FLOP : $7n$
end while

where

$$\overline{A} = I + hA + \frac{1}{2}h^2A^2 + \frac{1}{6}h^3A^3 + \frac{1}{24}h^4A^4 \tag{2.3}$$

$$B_1 = hB + h^2AB + \frac{1}{2}h^3A^2B + \frac{1}{4}h^4A^3B \tag{2.4}$$

$$B_2 = \frac{1}{6}\left(4hB + 2h^2AB + \frac{1}{2}h^3A^2B\right) \tag{2.5}$$

$$B_3 = \frac{1}{6}hB \tag{2.6}$$

Each of these matrices will be computed once at the beginning of the simulation. The general recurrence formula is:

$$y_{k+1} = \overline{A}y_k + \overline{B}\overline{u}_{k+1} \tag{2.7}$$

where

$$\overline{B} = \begin{pmatrix} B_1 & B_2 & B_3 \end{pmatrix} \in R^{n\times 3m} \tag{2.8}$$

$$\overline{u}_{k+1} = \begin{pmatrix} u(t_k) \\ u(t_k + \frac{1}{2}h) \\ u(t_k + h) \end{pmatrix} \in R^{3m} \tag{2.9}$$

This recurrence relation can be parallelized using a prefix-scan. The previous relation can be written as

$$x_i = \begin{cases} b_0 & i = 0 \\ (x_{i-1} \otimes a_i) \oplus b_i & 0 < i < NSteps \end{cases} \tag{2.10}$$

where

$$\begin{array}{c} x_i = y_i^T \in R^{1\times n} \quad i = 1 \ldots NSteps \\ a_i = \overline{A}^T \in R^{n\times n} \quad i = 1 \ldots NSteps \\ b_0 = y_0^T \\ b_i = u_i^T\overline{B}^T \in R^{1\times n} \quad i = 1 \ldots NSteps \end{array} \tag{2.11}$$

and operators $\oplus$ and $\otimes$ are respectively vector - vector summation and row-vector matrix multiplication. Assess if the operators satisfy the conditions required to convert the recursive relation into a pre-scan.

1. $\oplus$ is associative. This is satisfied by the definition of $\oplus$ as a row-vector summation.

2. $\otimes$ is semiassociative (i.e. there exists a binary associative operator $\odot$ such that $(a \otimes b) \otimes c = a \otimes (b \odot c)$); $a \in R^{1\times n}$, $b, c \in R^{n\times n}$. The operator $\odot$ is defined as the matrix to matrix multiplication; the semi-associativity of $\otimes$ is satisfied.

3. $\otimes$ distributes over $\oplus$ (i.e. $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$); $a \in R^{1\times n}$, $b, c \in R^{n\times n}$. Here there is some abuse in notation, $\oplus$ is defined on the left side of the equation as matrix - matrix summation and in the right side of the equation as vector-vector summation. The distributivity is satisfied for matrix and vector operations.

Let's define a new element as $c_i = [a_i, b_i]$ and a new operator as

$$c_i \bullet c_j \equiv [a_i \odot a_j, (b_i \otimes a_j) \oplus b_j]$$

Then we can define the sequence s_i

$$s_0 = [a_0, b_0]$$

$$s_i = s_{i-1} \bullet c_i$$

The execution time of the simulation if we assume that we have enough processors is

$$2lg(NSteps)\,(T_\odot + T_\otimes + T_\oplus)$$

where $T_\odot$, $T_\otimes$, $T_\oplus$ are the execution times of the operators $\odot$, $\otimes$, $\oplus$. Since, $T_\odot = n^3$, $T_\otimes = n^2$, $T_\oplus = n$ the execution time of the parallel simulation algorithm will be

$$2lg(NSteps)\left(n^3 + n^2 + n\right)$$

The serial execution time was of the order $O(NStepsn^2)$; the parallel execution time is of the order $O(lg(NSteps)n^3)$. Therefore the parallel implementation may be faster than the serial implementation if $NSteps > lg(NSteps)n$. The analysis phase of the algorithm reads n and provides the minimum number of steps that provides some speed-up.

Examples:

If $n = 10$; If $NSteps > 60$; $NSteps > 10lg(NSteps)$. We need more than 60 simulation steps.

If $n = 100$ we need more that 1000 simulation steps to justify the parallel implementation.

Discussion:

The operator timing formulas are for serial implementation and dense matrices. If we have sparse matrices the same conclusion apply if the operators have serial implementations.

There is another trade-off to consider for a parallel implementation. That is the grid adaptivity that can easily be implemented in the serial algorithm. For the parallel algorithm we can estimate the error after the simulation and have to re-run the simulation.

Another avenue is to use a serial-like implementation that would parallelize the vector matrix operations.

Chapter 3

Adams-Moulton Method

The basic Adams-Moulton algorithm is

$$y^*_{k+1} = y_k + \frac{h}{24}\left(55f(x_k, y_k) - 59f(x_{k-1}, y_{k-1}) + 37f(x_{k-2}, y_{k-2}) - 9f(x_{k-3}, y_{k-3})\right) \quad (3.1)$$

$$y_{k+1} = y_k + \frac{h}{24}\left(9f(x_{k+1}, y^*_{k+1}) + 19f(x_k, y_k) - 5f(x_{k-1}, y_{k-1}) + f(x_{k-2}, y_{k-2})\right) \quad (3.2)$$

We assume further that the system is linear $\dot{y} = Ay + Bu$; $A \in R^{n\times n}$, $y \in R^n$, $u \in R^m$, $B \in R^{n\times m}$, $x \equiv t \in R$, $f(t, y) = Ay + Bu(t)$. The algorithm becomes

$$\begin{aligned} y^*_{k+1} = y_k + \frac{h}{24}\left(55Ay_k + 55Bu(t_k) - 59Ay_{k-1} - 59Bu(tk-1)\right. \quad (3.3)\\ \left. +37Ay_{k-2} + 37Bu(t_{k-2}) - 9Ay_{k-3} - 9Bu(t_{k-3})\right) \end{aligned}$$

$$\begin{aligned} y_{k+1} = y_k + \frac{h}{24}\left(9Ay^*_{k+1} + 9Bu(t_{k+1}) + 19Ay_k + 19Bu(t_k)\right. \quad (3.4)\\ \left. -5Ay_{k-1} - 5Bu(t_{k-1}) + Ay_{k-2} + Bu(t_{k-2})\right) \end{aligned}$$

$$y_{k+1} = \left(I + \frac{28h}{24}A + \frac{495h^2}{576}A^2\right)y_k + \quad (3.5)$$

$$\left(-\frac{531h^2}{576}A^2 - \frac{5h}{24}A\right)y_{k-1} + \left(\frac{333h^2}{576}A^2 + \frac{h}{24}A\right)y_{k-2} + \quad (3.6)$$

$$\left(-\frac{81h^2}{576}A^2\right)y_{k-3} + \frac{9h}{24}Bu(t_{k+1}) + \quad (3.7)$$

$$\left(\frac{495h^2}{576}AB + \frac{19h}{24}B\right)u(t_k) + \left(-\frac{531h^2}{576}AB - \frac{5h}{24}B\right)u(t_{k-1}) + \quad (3.8)$$

$$\left(\frac{333h^2}{576}AB + \frac{h}{24}B\right)u(t_{k-2}) + \left(-\frac{81h^2}{576}AB\right)u(t_{k-3}) \quad (3.9)$$

$$y_{k+1} = A_0 y_k + A_1 y_{k-1} + A_2 y_{k-2} + A_3 y_{k-3} + \tag{3.10}$$
$$B_{-1} u(t_{k+1}) + B_0 u(t_k) + B_1 u(t_{k-1}) + B_2 u(t_{k-2}) + B_3 u(t_{k-3}) \tag{3.11}$$

where

$$A_0 = \left(I + \frac{28h}{24} A + \frac{495h^2}{576} A^2 \right) \tag{3.12}$$
$$A_1 = \left(-\frac{531h^2}{576} A^2 - \frac{5h}{24} A \right) \tag{3.13}$$
$$A_2 = \left(\frac{333h^2}{576} A^2 + \frac{h}{24} A \right) \tag{3.14}$$
$$A_3 = \left(-\frac{81h^2}{576} A^2 \right) \tag{3.15}$$
$$B_{-1} = \frac{9h}{24} B \tag{3.16}$$
$$B_0 = \left(\frac{495h^2}{576} AB + \frac{19h}{24} B \right) \tag{3.17}$$
$$B_1 = \left(-\frac{531h^2}{576} AB - \frac{5h}{24} B \right) \tag{3.18}$$
$$B_2 = \left(\frac{333h^2}{576} AB + \frac{h}{24} B \right) \tag{3.19}$$
$$B_3 = \left(-\frac{81h^2}{576} AB \right) \tag{3.20}$$
$$\tag{3.21}$$

This recursion is of the form

$$x_i = \begin{cases} b_i \in R^n & i \leq 3 \\ (x_{i-1} * a_0) + (x_{i-2} * a_1) + (x_{i-3} * a_2) + (x_{i-4} * a_3) + b_i & i \geq 4 \end{cases} \tag{3.22}$$

where $a_0 = A_0^T \in R^{n \times n}$, $a_1 = A_1^T \in R^{n \times n}$, $a_2 = A_2^T \in R^{n \times n}$, $a_3 = A_3^T \in R^{n \times n}$, $x_i \equiv y_{i+1}^T \in R^{1 \times n}$, and

$$b_i = \left(B_{-1} u(t_i) + B_0 u(t_{i-1}) + B_1 u(t_{i-2}) + B_2 u(t_{i-3}) + B_3 u(t_{i-4}) \right)^T \in R^{1 \times n}$$

and $+$ and $*$ are regular matrix operations. The previous recursive operation can be converted into a first order recursive operation by defining

$$\tilde{x}_i = \begin{pmatrix} x_i & x_{i-1} & x_{i-2} & x_{i-3} \end{pmatrix}$$

Then,

$$\tilde{x}_i = \left(\tilde{x}_{i-1} \otimes \overline{A}\right) \oplus \tilde{b}_i \tag{3.23}$$

where

$$\overline{A} = \begin{pmatrix} a_0 & I & 0 & 0 \\ a_1 & 0 & I & 0 \\ a_2 & 0 & 0 & I \\ a_3 & 0 & 0 & 0 \end{pmatrix} \in R^{4n \times 4n}$$

$$\tilde{b}_i = \begin{pmatrix} b_i & 0 & 0 & 0 \end{pmatrix}$$

Similarly with the previous case we can define the element

$$c_i = [a_i, b_i]; a_i \in R^{4n \times 4n}; b_i \in R^{1 \times 4n}$$

The $\bullet$ operator is defined in a similar way

$$c_i \bullet c_j \equiv [a_i \odot a_j, (b_i \otimes a_j) \oplus b_j]$$

We can define the sequence

$$s_0 = [I, \tilde{b}_0]$$

$$s_i = s_{i-1} \bullet [\overline{A}, \tilde{b}_i]$$

The execution time for $NSteps$ time steps simulation is

$$2*lg(NSteps)((4n)^3+(4n)^2+4n) = 2*lg(NSteps)(64*n^3+16*n^2+4*n) \tag{3.24}$$

the serial execution time is

$$NSteps * (k_1 n^2 + k_2 n) \tag{3.25}$$

In order to have acceleration we need to have $64 * lg(NSteps)n << NSteps$.
Example:
For $n = 10$, $NSteps$ should be greater than 10000 in order to get some speed-up.

Chapter 4

Implementation

4.1 General Algorithm

4.2 ParODE Interface and short user manual

The ParODE library can be used through a plain C interface, SciLab, or Python. Scilab and Python interfaces are wrappers for the C interface. Alternatively, the user can compile its sources together ParODE's sources.

The ParODE library is compiled on the target system in a shared library. The library also includes several kernel files (*.cl) and header files.

4.2.1 Shared Library - C API Functions

4.2.2 System Initialization

The user has the option of using for the simulation all the GPUs available on the system or select only a few of them.

void InitializeAllGPUs(char* pszKernelFolder,
char* pszIncludeFolder,
int *nErr);

Initialize all the GPUs available on the system. Parameters

- Input Parameters
- char *pszKernelFolder - the folder where the ParODE kernel files (*.cl) are stored.
- char *pszKernelFolder - the folder where the ParODE header files for kernels are stored.
- Output Parameters

Algorithm 3 Parallel simulation algorithm for ODEs; Inputs: $u(t_k)$, A, B, $NSteps$, n, GPU Memory Size

Compute the number of time steps per batch NSB based on the total global memory available for all devices.
$NDevs$ the total number of devices available
Allocate the memory required for vectors $c[0 \ldots NDevs - 1]$
for $nB = 0$ **to** $NSteps/NSB$ **do** ▷ Distribute the work to all devices
  Compute the number of steps per device $nElems[0 \ldots NDevs - 1]$
  $nElementStart[0] \leftarrow 0$
  for $nDI = 0$ **to** $NDevs - 1$ **do**
    if $nDI > 0$ **then**
      $nElementStart[nDI] \leftarrow nElementStart[nDI - 1] + nElems[nDI]$
    end if
  end for
  for $nDI = 0$ **to** $NDevs - 1$ **do** ▷ In Parallel
    Copy/build $u(t_k)$ to GPU for
    $k = nB * NSB + nElementStart[nDI], \ldots,$
    $nB * NSB + nElementStart[nDI] + nElems[nDI] - 1$
    Build/initialize the vector to be scanned for the current device $c[nDI]$
    for $d = 0$ **to** $\log_2(nElems[nDI]) - 1$ **do** ▷ Up-sweep
      for $i = 0$ **to** $nElems[nDI] - 1$ **by** 2^{d+1} **do** ▷ In Parallel
        $c[nDI][i + 2^{d+1} - 1] \leftarrow c[nDI][i + 2^d - 1] \bullet$
        $c[nDI][i + 2^{d+1} - 1]$
      end for
    end for
  end for
  Synchronize Execution
  $Acc \leftarrow 0$
  for $nDI = 1$ **to** $NDevs - 1$ **do**
    $Temp \leftarrow c[NDevs][nElems[NDevs] - 1]$
    $c[NDevs][nElems[NDevs] - 1] \leftarrow Acc$
    $Acc \leftarrow Acc \bullet Temp$
  end for
  for $nDI = 0$ **to** $NDevs - 1$ **do** ▷ In Parallel
    for $d = \log_2(nElems[nDI]) - 1$ **downto** 0 **do** ▷ Down-sweep
      for $i = 0$ **to** $nElems[nDI] - 1$ **by** 2^{d+1} **do** ▷ In Parallel
        $tc \leftarrow c[nDI][i + 2^d - 1]$
        $c[nDI][i + 2^d - 1] \leftarrow c[nDI][i + 2^{d+1} - 1]$
        $c[nDI][i + 2^{d+1} - 1] \leftarrow c[nDI][i + 2^{d+1} - 1] \bullet tc$
      end for
    end for
  end for
  Copy the results from GPU to CPU for batch nB
end for

- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero if the system was properly initialized. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

void GetAvailableGPUs(int* nGPUs, int GPUIds[], int nErr[]);

Queries the system for the available GPUs.
Parameters

- Input Parameters
- none
- Output Parameters
- int *nGPUs - address of the variable that will receive the number of available GPUs.
- int GPUIds[] - array where the (*nGPUs) available Ids will be stored. This is an user allocated array; the user is responsible to allocate an array large enough to accommodate all the Ids. By default ParODE looks for at most eight GPUs on the system.
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero on success; a value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

void GetDeviceName(int *DeviceId, char strDeviceName[], int nErr[]);

Returns the name of device (*DeviceId). This function can only be called after a call to GetAvailableGPUs(); (*DeviceId) is one of the values stored by GetAvailableGPUs() in the output parameter GPUIds.
Parameters

- Input Parameters
- int *DeviceId - device Id;
- Output Parameters
- char strDeviceName[] - char array where the function will store the name of the device. The user is responsible for the allocation and deallocation of this array. The user should provide an array large enough to accommodate the name of the device.
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero on success; a value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

void InitializeSelectedGPUs(int *nGPUs, int GPUIds[],
char* pszKernelFolder,
char* pszIncludeFolder, int nErr[]);

Initialize GPUs. This function can only be called after a call to GetAvailableGPUs(); (GPUIds[]) is a subset of the values provided by GetAvailableGPUs().

Parameters

- Input Parameters
- int *nGPUs - number of GPUs to be initialized
- int GPUIds[] - array with the GPUs to be initialized.
- char *pszKernelFolder - the folder where the ParODE kernel files (*.cl) are stored.
- char *pszKernelFolder - the folder where the ParODE header files for kernels are stored.
- Output Parameters
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero if the system was properly initialized. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

void CloseGPUC();

Close the library; release the associated resources.

Parameters

- Input Parameters
- none
- Output Parameters
- none

4.2.3 Timing functions

void StartTimer(int nErr[]);

Initialize the timer. The function can only be called after the library initialization.

Parameters

- Input Parameters
- none

- Output Parameters
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero if the system was properly initialized. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

StopTimer(float *fTime, int nErr[]);

Return the time in ms passed since the last call to StartTimer. The function can only be called after the library initialization.

Parameters

- Input Parameters
- none
- Output Parameters
- float *fTime - address where the time in ms passed since the last call to StartTimer will be stored.
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero if the system was properly initialized. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

4.2.4 Simulation functions

The library enables LTI system simulation using Runge-Kutta 4th order or Adams-Bashforth-Moulton methods. The user provides the system matrices A, B, C, D, initial state x_0 and an input function u.

The system is defined as usual

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} \tag{4.1}$$

$$y = C\mathbf{x} + D\mathbf{u} \tag{4.2}$$

void RegisterInput(char uFunc[], int uIndex[], int nErr[]);

Register an input function into the ParODE. The function can only be called after the library initialization.

Parameters

- Input Parameters
- char uFunc[] - string array with the input function code.
- Output Parameters

- int *uIndex - address where the input id will be stored.
- int *nErr - address where the return error code will be stored.

The input function has to have the following template
"fType UFunc(fType t, int nInputIndex){ ... return value; }"

- UFunc - this is the predefined name of input functions inside ParODE. Do not Change.
- fType is the floating point type used for GPU computations. fType is defined to a floating type within ParODE.
- t is the time.
- nInputIndex is the input number.

For example, for a single input system with a sinusoidal input the input may be registered as follows:

```
...
int uIndex, nErr;
char *strU = "fType UFunc(fType t, int nInputIndex){ return (fType)sin(t); }";
RegisterInput(strU, &uIndex, &nErr);
...
```

In principle UFunc can include any self contained C code that can be executed on the GPU. However, for best performance it is recommended to avoid branching instructions. For example, for a two inputs system, where the first input is a sine and the second one is a cosine the input can be defined as:

```
...
int uIndex, nErr;
char *strU = "fType UFunc(fType t, int nInputIndex){"
"fType fVal = nInputIndex == 0 ? (fType)sin(t) :"
"nInputIndex == 1 ? (fType)cos(t) :"
"0.0;"
"return fVal;"
"}"
RegisterInput(strU, &uIndex, &nErr);
...
```

(*nErr) is set to zero on success. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

void SimulateODE(fType A[], fType B[], fType C[], fType D[],
int *nInputs, int *nStates, int *nOutputs,
int *ZeroB, int *ZeroC, int *ZeroD,
int *uIndex,
double *tStart, double *tEnd, double *tStep,
fType x0[],
int *solver,
fType tVect[], fType xVect[], int *nSteps, int nErr[]);

Simulate the LTI system (A, B, C, D) with initial state $x0$ and input index $uIndex$ from $tStart$ to $tEnd$, the output values are computed every $tStep$.

Parameters

- Input Parameters
- fType A[] - row major representation of the system matrix A is $nStates \times nStates$.
- fType B[] - row major representation of the input matrix B is $nStates \times nInputs$. If B is zero than ZeroB should be 1;
- fType C[] - row major representation of the output matrix C is $nOutputs \times nStates$. If C is zero than ZeroC should be 1;
- fType D[] - row major representation of the input - output matrix D is $nOutputs \times nInputs$. If D is zero than ZeroD should be 1;
- int *nInputs - number of inputs
- int *nStates - number of states
- int *nOutputs - number of outputs
- int *ZeroB - flag indicating that B is zero
- int *ZeroC - flag indicating that C is zero
- int *ZeroD - flag indicating that D is zero
- int *uIndex - index of the input used for simulation
- double *tStart - simulation time start
- double *tEnd - end simulation time
- double *tStep - result time step. It not required that this will be the internal time step but rather that the output will be sampled every *tStep.
- fType x0[] - initial state.
- int *solver - solver; 0 - 4th order Runge-Kutta, 1 - Adams - Bashforth - Moulton with 4 states memory

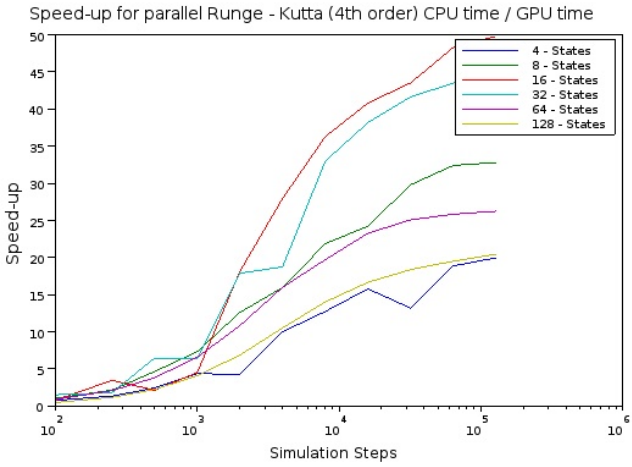

Figure 4.1: Speed up for Runge-Kutta 4th order; GPU - NVidia GTX 480; CPU - Intel I5 system; Ubuntu 12.04

- Output Parameters
- fType tVect[] - vector with time samples. The user is responsible with the memory allocation and deallocation. The vector should be large enough to accommodate all simulation steps.
- fType xVect[] - result vector. The outputs are stored in sequence. The user is responsible with the memory allocation and deallocation. The vector should be large enough to accommodate all simulation steps.
- int *nSteps - number of actual simulation steps returned. This is usually $floor(((*tEnd) - (*tStart))/(*tStep))$
- int *nErr - address where the return error code will be stored.

(*nErr) is set to zero if the system was successfully simulated. A value less than 0 is returned on failure. A description of the error can be obtained using GetErrorDescription().

4.3 Performance Tests

ParODE solvers were tested against boost::odeint RK4's implementation and against a custom ABM implementation. Figure 4.1 shows the speed-up of the parallel Runge-Kutta 4th order implementation for different system sizes with 4 inputs and 4 outputs. Figure

The plots show that for the RK method the parallel implementation outperforms the serial implementation. As expected the speed-up increases with the

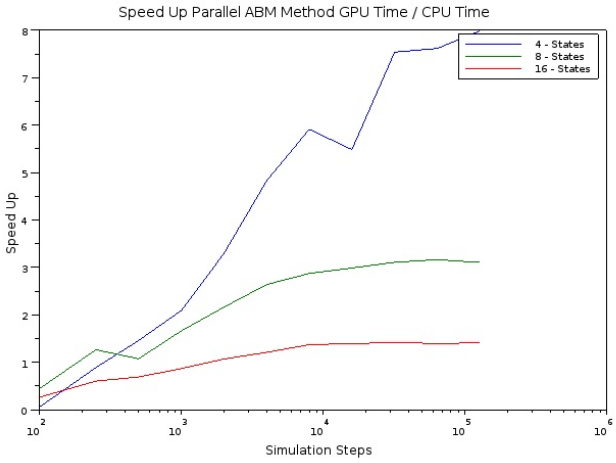

Figure 4.2: Speed up for ABM with 4 steps memory; GPU - NVidia GTX 480; CPU - Intel I5 system; OS - Ubuntu 12.04

number of simulation steps and saturates when the GPU reaches its maximum occupancy.

The ABM parallel implementation is much more demanding on GPU resources. Due to the 4 memory steps, the parallel implementation operates with matrices 4 times larger than the serial implementation leading to rapid saturation. However, the implementation is scalable to newer GPUs and it is expected that the performance is much better for newer and more powerful cards such as NVidia Titan.

The implementation uses OpenCL therefore it allows for deployment on either ATI or NVidia hardware.

References

- OpenCL Reference Manual
- Guy E. Blelloch, Prefix Sums and Their Applications
- William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery Numerical Recipes in C