UPPSALA
UNIVERSITET

Project

HIGH PERFORMANCE PROGRAMMING 10 CREDITS 1TD062 VT2022

Group members:

Ibrohim HAMOUD ibrohimmn@gmail.com

UPPSALA

March 19, 2022

1 Introduction

This work attempts to Implement a finite element solver in the c program using continuous piecewise linear approximations to solve Burger's equation in one dimension. For, the second part of this work, optimization procedures are implemented algorithmically and using OpenMP parallelization algorithm to improve the performance of the program.

2 Theory

The problems to be solved is represented by the Burger's equation in one dimension that is given as follows

$$\begin{aligned} \partial_t u + \partial_x(\tfrac{u^2}{2}) - \partial_x(\epsilon\partial_x u), &\quad (x,t) \in I \times (0,T], \\ u(a,t) = g_a(t) &\quad t \in (0,T], \\ u(b,t) = g_b(t) &\quad t \in (0,T], \\ u(x,0) = u_0(x) &\quad x \in I. \end{aligned} \tag{1}$$

The domain is set to be $I = (0, 2\pi)$ and the boundary conditions $g_a(0) = g_b(0) = 0$, and the initial solution is given by $u_0(x) = sin(x)$. Here, $\beta(u) = \frac{1}{2}u^2$. The problem is run until the final time $T = 2$ with parameter $\epsilon = \frac{1}{h}$, where h is the mesh size.

Applying the FM-nethod on Eq. (1) results in the following system of ODEs

$$\mathbf{M}\dot{\boldsymbol{\xi}} = \mathbf{A}\boldsymbol{\beta}(\boldsymbol{\xi}) - \epsilon\mathbf{S}\boldsymbol{\xi} - \mathbf{b}(t) - \mathbf{c}(t) - \epsilon\mathbf{d}(t), \quad t \in (0,T]. \tag{2}$$

The matrix elements $\mathbf{b}_i(t)$,$\mathbf{c}_i(t)$ and $\mathbf{d}_i(t)$ are non zero only for $i = 1$ and $i = N - 1$. The elements of the matrices $\mathbf{A}$, $\boldsymbol{\beta}$ and $\mathbf{S}$ are given to be

$$\mathbf{M}_{ij} := \int_{\mathbf{I}} \phi_j\phi_i dx, \quad \mathbf{A}_{ij} := \int_{\mathbf{I}} \partial_x\phi_j\phi_i dx, \quad \mathbf{S}_{ij} := \int_{\mathbf{I}} \partial_x\phi_j\partial_x\phi_i dx.$$

Time discretization of the ODE-system in (2) is achieved with 4-order Runge–Kutta methods (RK4). For more details regarding the derivation, check the appendix A.

=

2.1 Conjugate gradient solver

In the implementation, The ODE-solver is chosen to be the conjugate gradient solver for the reason that the matrix M is sparse with strict positive entries and slightly diagonally dominant. This algorithm is an iterative technique for solving large sparse systems of linear equations $Ax = b$, and it is a useful tool in approximating solutions to linearized partial differential equations. The algorithm is given as follows [1]

```
-------------------------------------------------------
Given initial guess e.g. x0=0, put r=x0, p=x0.
For k = 1:n
1) Compute : w = A*p
2) Compute :r_norm = r*r
```

```
3) Compute :p_norm = w*p
4) Find    :alpha = r_norm/p_norm
5) Put     :r_old = r
6) Do      :r = r-alpha*w
6) Compute :r_norm_new = r*r
7) If      :r_norm_new < tol      => break the iteration
8) Else    :find beta = r_norm_new/r_norm
9) Do      :p = r+beta*p
10) Put    :r_norm = r_norm_new
11) Repeat
--------------------------------------------------------
```

3 Solution

The FM-implementation in the c program consists of two main large loops, the simulation loop where the four steps of RK4 are implemented to arrive at the solution at each time-point, and the CG-loop that resides inside the simulation loop and is implemented once in each of the four steps. The program starts with defining the variables that are required in the simulation (e.g. mesh-size, grid, final time and time-step etc), followed by assembling of matrices **M**, **A** and **S**. Each iteration in the simulation loop contains four computational blocks. in each block, a function is called to compute the right-hand side in E.q (2) that is passed as an input to CG-solver. Here, CG-solver is employed to solve a linear system of each RK-stage which involves the mass matrix In-between the blocks a small loop is run that adds the contribution of RK-step into the solution at this specific iteration. the solution at the end of the iteration is $u_{new} = u_{old} + (k_1 + 2k_2 + 2k_3 + k_4)/6$, where each of the k_i are added in the loops between the steps. This is done to spare from allocating separate memory each of the k_i.

4 Algorithmic optimization

Optimization of the code is performed in two stages. The first stage proceeds with algorithmic optimization that aims to improve the performance of the serial code. The main procedures applied in this stage are loop fusion, array recycling, and transition from regular format into CSR format regarding matrix representation.

Loop fusion aims to stack as many computations in the same loop as possible to avoid building a new loop, and thus enhance performance. While array recycling refers to reuse arrays to store data instead of creating a new one. This is due to the nature of the problem at hand, where a small mesh size causes a very large memory allocation for various vectors and matrices involved in the computation. This demands the program to be memory-friendly as possible.

The transition from regular format into CSR format includes assembling the matrices directly into CSR format, where each matrix is represented by three vectors (Vector for storing values, vector for storing column indices, and a vector for storing row off-set of the first non-zero element in each row). This triplet format is wrapped in a struct data-type to represent a matrix. The CSR-format assembly proved to be the most challenging part of the program as the requirement is not only to find a pattern between the indices that helps to build the triple vectors but also an exact pattern that makes the assembly loop independent such that it allows parallelization later on (Mass matrix assembly in regular and CSR format are shown respectively in appendix B and C). However, this proved to save a lot of memory allocation as well as complexity in matrix-vector multiplication as yet to be shown.

5 Optimization and discussion

OpenMP is a library for parallel programming that provides support for parallel programming in shared-memory environments using a set of compiler directives. OpenMP identifies parallel regions as blocks of code that may run in parallel, and the directives instruct the OpenMP library to execute the region in parallel. The parallelization proceeds with creating multiple threads which are processing cores in the system, and these threads simultaneously execute the parallel region.

5.1 OpenMP in the simulation loop

The simulation loop appears to be very parallel-resistant as the iteration are strongly dependent on each other, as well as all the RK4 steps residing in it. The small loops are the only parts that can be implemented in parallel. However, even this could be futile as these loops carry very light computation and the expense of parallelization might outweigh the advantage. The action of building threads and orchestrating their teamwork in the parallel region can cost more than the gained time.

In the RHS function that computes the right-hand side of the linear system, the main computational weight resides in two matrix-vector multiplication. These were successfully run in parallel with the OpenMP section directive, and effectively halved the time required for calling this function. The most expensive single operation in the program was found to be the spMxV-algorithm for matrix-vector multiplication (Appendix D). The algorithm has independent loops and it is a perfect candidate for a parallel region. This algorithm can be prone to a defect in load-balance which is determined by the structure of the matrix involved in the operation. However, for the problem at hand, the matrices involved have a uniform and equal distribution of non-zero elements along the diagonal which makes load-imbalance a remote possibility. Thus, a static partition of the loop is suitable.

5.2 OpenMP in the CG-solver

The GC algorithm appears to be a good candidate for a parallel region. The first attempt to build the parallel region is given by the version 1 script below. The first script contains 8 barriers. Although, the first barrier that is related to the spMxV-algorithm can be easily disposed of by simply embedding the spMxV-algorithm inside the CG algorithm instead of a function call. Among the other barriers, two are related to the variables beta and alpha, and four are related to computing normp and normr. This was easily reduced by setting alpha and beta to private. Thus deleting two barriers. A further reduction of two barriers is achieved by re-ordering the operation in the algorithm such that the threads do not wait for setting normr and normp into zero (See version 2) [2].

```
( CG version 1 )
=================
#pragma omp parallel
   compute w = A*p
# end of parallel region
    <Barrier>
#pragma omp parallel
   #pragma omp single     normp = 0
   <Barrier>
   #pragma omp for        normp  += p[i]*w[i]
   <Barrier>
   #pragma omp single     alpha = normr_old/normp
   <Barrier>
   #pragma omp for        (r_old[i] = r[i], x[i] = x[i] + alpha*p[i]
                          r[i] = r[i] - alpha*w[i])
    <Barrier>
    #pragma single        normr = 0
    <Barrier>
    #pragma omp for       normr += r[i]*r[i]
    <Barrier>
    #pragma single        beta = normr/normr_old
    <Barrier>
    #pragma omp for       p[i] = r[i] + beta*p[i]
    <Barrier>
# end of parallel region
    check rnorm <tol
```

```
( CG version 2 )
================
#pragma omp parallel
   compute w = A*p
# end of parallel region
<Barrier>
#pragma omp parallel private(alpha, beta)
    #pragma omp signle nowait      norm =0
    #pragma omp for                normp+=p[i]*w[i]
    <Barrier>
                                   alpha = norm_old/normp
```

```
        #pragma omp for           (r_old[i] = r[i], x[i] = x[i] + alpha*p[i]
                                   r[i] = r[i] - alpha*w[i])
        <Barrier>
        #pragma signle nowait      normp=0
        #pragma omp for            normr += r[i]*r[i]
        <Barrier>
                                   B = normr/normr_old
        #pragma omp for            p[i] = r[i]+B*p[i]
        #pragma single no wait     normr_old = normr
        <Barrier>
    #end of parallel region
    check rnorm <tol
```

6 Result

The performance check throughout this work was performed on a grid size of $n = 641$ grid-point with the result exhibited in the table 1. Algorithmic optimization gives the expected results in the time reduction. Additionally, the parallelization of the spMxV algorithms appears the crucial single optimization step done for the simulation. However, the parallelization of the CG-step is unexpectedly ineffective. The same is applicable for all the small loops in the program. However, when measuring the time consumption solely for CG-sector, it hardly exceeds 0.0001 seconds. Thus, the algorithm was separated from the rest of the body and run for $n = 1000000$ grid-size. The obtained results are as follows 0.007, 0.0055, and 0.004 for unparallel-GC, parallel GC-version 1, and parallel GC-version-2, respectively. Here, the spMxV-algorithm that resides inside the GC-block is parallelized in all cases. These results might point to that the advantage of fully parallelizing this block is an advantage only for a large problem. Therefore, it is postulated that repeated thread creation and orchestration in computationally inexpensive loops (Apart from the spMxV) might cause a lot of overhead. On the other hand, it is possible that the culprit is the false sharing problem especially when it comes to array updation. However, big chunks are used to each thread statically. This means that the false sharing on the border of the threads should not statistically matter. This issue was not possible to mitigate, and it is still hooked in the program.

Non-Optimized	Algorithmically-optimized	spMxV parallelized	RHF parallelized	CG parallelized
≈ 7	≈ 5.1	≈ 3.8	≈ 3.1	≈ 15.1

Table 1: Simulation time for different optimization steps. The results are given in seconds for a grid $n = 641$

References

[1] “https://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf.”

[2] “http://www.it.uu.se/research/publications/reports/2004-048/2004-048-nc.pdf.”

A Derivation

To proceed in implementing FEM to Eq. (1), The trial and test functions are constructed for the purpose of obtaining weak formulation of the differential equations. Thereafter, the domain is split into N equal sub-intervals and the system is discretized with a piecewise linear function. The before-mentioned procedure results in a system of ODEs that can be effectively solved with 4RK discretization in time.

The trial space is constructed as follows

$$u \in V_g = \{\nu; ||\nu||^2 + ||\nu^{'}||, \nu(a) = g_a(t), \nu(b) = g_b(t)\},$$

and the test space follows

$$\nu \in V_0 = \{\nu; ||\nu||^2 + ||\nu^{'}||, \nu(a) = 0, \nu(b) = 0\}$$

.

Thus, the weak formulation is given by

$$\begin{aligned}&\text{Find} \quad u \in V_g \quad \text{such that}\\ &\textstyle\int_a^b u_t \nu dx + \int_a^b \frac{\partial}{\partial x}\beta(u)\nu dx + \int_a^b \epsilon u^{'}\nu^{'} dx = 0,\\ &\qquad\qquad\qquad\qquad\qquad\qquad \forall \nu \in V_0.\end{aligned}$$

Next, the construction of a space of continuous piece-wise linear function gives

$$V_{g,h} = \{\nu; \nu \in C^0(t), \nu|_{I_i} \in P_1(I_i), I_i \in (a,b), \nu(a) = g_a(t), \nu(b) = g_b(t), \}$$

$$V_{0,h} = \{\nu; \nu \in C^0(t), \nu|_{I_i} \in P_1(I_i), I_i \in (a,b), \nu(a) = 0, \nu(b) = 0.\}$$

Thus, the Galerkin Finite element formulation is given by

$$\begin{aligned}&\text{Find} \quad u \in V_{g,h} \quad \text{such that}\\ &\textstyle\int_a^b \dot{u}_h \nu_h dx + \int_a^b \frac{\partial}{\partial x}\beta(u_h)\nu_h dx + \int_a^b \epsilon u_h^{'}\nu_h^{'} dx = 0\\ &\qquad\qquad\qquad\qquad\qquad\qquad \forall \nu_h \in V_{0,h}.\end{aligned} \tag{3}$$

Both the trial and test function can be represented as a linear combination of the Lagrangian basis functions,

$$\begin{aligned}&u_h \in V_{g,h} \Rightarrow u_h = \textstyle\sum_{j=1}^{n-1}\xi_j(t)\phi_j + g_a(t)\phi_0 + g_b(t)\phi_n,\\ &\nu_h \in V_{0,h} \Rightarrow \textstyle\sum_{i=1}^{n-1}\xi_i\phi_i.\end{aligned} \tag{4}$$

Here, $\{\phi_i\}|_{i=0}^{N}$ is a set of basis functions that spans the constructed space of piecewise linear function $V_h = \{\nu; ||\nu||^2 + ||\nu^{'}||, \nu|_{I_i} \in P_1(I_i)\}$, where $V_{g,h}, V_{0,h} \in V_h$.

Inserting the functions in Eq. (5) into the GFEM in Eq. (3) gives

$$\int_a^b (g_a^{'}\phi_0 + \sum_{j=1}^{N-1}\xi_j^{'}(t)\phi_j + g_b^{'}\phi_N)\phi_i dx + \int_a^b \frac{\partial}{\partial x}(\beta(g_a(t))\phi_0 + \sum_{j=1}^{N-1}\beta(\xi_j(t))\phi_j + \beta(g_b(t))\phi_N) + \int_a^b \epsilon \frac{\partial}{\partial x}(g_a(t)\phi_0 + \sum_{j=1}^{N-1}\xi_j(t)\phi_j + g_b(t)\phi_N)\phi_i^{'} dx, \tag{5}$$

Which results in the following system of ODEs

$$\mathbf{M}\dot{\boldsymbol{\xi}} = \mathbf{A}\boldsymbol{\beta}(\boldsymbol{\xi}) - \epsilon\mathbf{S}\boldsymbol{\xi} - \mathbf{b}(t) - \mathbf{c}(t) - \epsilon\mathbf{d}(t), \quad t \in (0, T].$$

The matrix elements $\mathbf{b}_i(t)$,$\mathbf{c}_i(t)$ and $\mathbf{d}_i(t)$ are non zero only for $i = 1$ and $i = N-1$. The elements of the matrices $\mathbf{A}$, $\boldsymbol{\beta}$ and $\mathbf{S}$ are given to be

$$\mathbf{M}_{ij} := \int_{\mathbf{I}} \phi_j\phi_i dx, \quad \mathbf{A}_{ij} := \int_{\mathbf{I}} \partial_x\phi_j\phi_i dx, \quad \mathbf{S}_{ij} := \int_{\mathbf{I}} \partial_x\phi_j\partial_x\phi_i dx.$$

Eq. (A) is solved using 4RK discretization in time. In the implementation, The ODE-solver is chosen to be the conjugate gradient solver for the reason that the matrix M is sparse with strict positive entries and slightly diagonally dominant.

B Assembly of the mass matrix in regular format

```
void Mass(int n, double h, double **M){
int i;
for ( i=0; i<n-1; i++ ){
M[i][i]     += h/3;
M[i][i+1]   += h/6;
M[i+1][i]   += h/6;
M[i+1][i+1] += h/3;
}
M[0][0] += h/3;
M[n-1][n-1] += h/3;
}
```

C Assembly of the mass matrix in CSR format

```
void Mass_CSR(int n, double h, double **V, double **C, double **I){
int N = (n-1)*2+n;
*V = realloc(*V,N*sizeof(double));
*C = realloc(*C,N*sizeof(double));
*I = realloc(*I,(n+1)*sizeof(double));
int i;
int r = n-1;
int l = N-2;
for (i=N-1; i>0; i-=3){
r= (n-1)-( (N-1-i)/3 ); // equivalent to r--
```

```
        l = (N-2)-(N-i)+1;  // equivalent to l -=3
(*V)[i]   = 2*h/3;
(*V)[i-1] = h/6;
(*V)[i-2] = h/6;
(*C)[i]   = r;
(*C)[i-1] = r-1;
(*C)[i-2] = r;
(*I)[r] = l;
}
(*I)[0]=0;
(*V)[0]=2*h/3;
(*C)[0]=0;
(*I)[n]=N;
}
```

D spMxV algorithm

```
void CSR_matrix_array(int n, double *V, double *C, double *I, double *arr, double *result){
int i,j;
for (i=0; i<n; ++i ){
result[i] = 0;
     for (j=I[i]; j<I[i+1]; ++j){
      result[i] += V[j] * arr[(int)C[j]];
        }
}
}
```