Concurrent algorithms for integrating three-dimensional B-spline functions into machines with shared memory such as GPU

Maciej Woźniak[1], Anna Janina Szyszka[1]
[1] Institute of Computer Science,
Faculty of Electronics, Telecommunication and Computer Science,
AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Kraków, Poland
macwozni@agh.edu.pl jerboamouse@gmail.com

Keywords: Isogeometric Finite Element Method, Numerical integration, Trace theory, Sum factorization

1. Introduction

The process of FEM calculations is split into two stages - the assembly of the discrete form and the resolution of algebraic equations. While there are various types of problems that require multiple integrations, modern solvers for IGA-FEM have greatly reduced the computation cost, such as by using the Alternating Direction Solver (ADS). In fact, with ADS, integration can make up as much as 80% of the overall cost. Historically, integration was done by processing each element concurrently, but now, with the advent of multi-level parallel computational clusters [4], there are additional two levels of parallelism available for the integration.

Incorporating concurrency in the integration process at the element level is key to improving performance. To achieve this, the use of Trace Theory [5] can be employed to derive the Foata Normal Form (FNF) and create a Diekert's dependency. By utilizing FNF, the creation and implementation of parallel algorithm on a GPU becomes much more manageable and efficient. Additionally, FNF offers near-optimal scheduling and GPU implementation, as well as theoretical validation of parallel algorithms. This methodology can be applied to a variety of integration algorithms to achieve improved results.

2. Model problem and IGA-discrete variational formulation

The goal of this study is to evaluate the cost of using different integration methods for assembling IGA matrices. To illustrate this, we will use the heat equation discretized in time using the forward Euler method and focus specifically on the cost of assembling the Mass matrix.

Find $u \in C^1((0,T), \mathrm{H}^1(\Omega))$ such that $u = u_0$ at $t = 0$ and, for each $t \in (0,T)$, it holds:

$$\int_\Omega \frac{\partial u}{\partial t} v dx = -\int_\Omega \nabla u \nabla v dx, \qquad \forall v \in \mathrm{H}^1(\Omega) \tag{1}$$

For simplicity, we consider a discrete-in-time version of problem by employing the forward Euler method.

$$\int_\Omega u_{n+1} v dx = \int_\Omega u_n v dx - \Delta_t \int_\Omega \nabla u_n \nabla v dx, \qquad \forall v \in H^1(\Omega) \tag{2}$$

The publication is co-financed from the state budget under the programme
of the Minister of Education and Science called "Excellent Science" project no. DNK/SP/548041/2022

Doskonała Nauka
Ministry of Education and Science
Republic of Poland
Republic of Poland

This study focuses on utilizing 3D-tensor B-spline basis functions with a uniform polynomial degree order and regularity at the interior faces of the mesh for ease of demonstration. However, it should be noted that the techniques presented can be easily adapted to other types of B-spline basis functions. For construction of B-spline basis functions we used Cox-de-Boor recursive formulae [6].

$$B_{i;0}(\xi) := \begin{cases} 1 & \text{if } \gamma_i \leq \xi < \gamma_{i+1} \\ 0 & otherwise \end{cases} \tag{3}$$

$$B_{i;q}(\xi) := \frac{\xi - \gamma_i}{\gamma_{i+q} - \gamma_i} B_{i;q-1}(\xi) + \frac{\gamma_{i+q+1} - \xi}{\gamma_{i+q+1} - \gamma_{i+1}} B_{i+1;q-1}(\xi), \qquad for\ 1 \leq q \leq p \tag{4}$$

3. Algorithms and computational cost

We compared two algorithms, the classical integration algorithm, and the sum factorization algorithm. In the classical integration algorithm, local contributions to the left-hand-side matrix A are represented as a sum over quadrature points. In this case, the associated computational cost scales, concerning the polynomial degree p as $\mathcal{O}(p^9)$ [2].

$$A^{\gamma}_{\beta,\delta} = \sum_{n_1,n_2,n_3=1}^{P_1,P_2,P_3} \omega_3^n B_j(x_3^n) B_{m;p}(x_3^n) B_i(x_2^n) \omega_2^n B_{l;p}(x_2^n) D(i_3,j_3,k_1,k_2) \omega_1^n B_h(x_1^n) B_{k;p}(x_1^n) J(x^n) \tag{5}$$

On the other side, Sum factorization algorithm is based on reorganizing the integration terms to reduce the computational cost, in terms of the polynomial degree p, associated with the sum procedure $\mathcal{O}(p^7)$ [2].

In practice is written as:

$$A^{\gamma}_{\beta,\delta} = \sum_{n_3=1}^{P_3} \omega_3^n B_j(x_3^n) B_{m;p}(x_3^n) C(i_2,i_3,j_2,j_3,k_1) \tag{6}$$

where

$$C(i_2,i_3,j_2,j_3,k_1) = \sum_{n_2=1}^{P_2} \omega_2^n B_i(x_2^n) B_{l;p}(x_2^n) D(i_3,j_3,k_1,k_2)$$

and

$$D(i_3,j_3,k_1,k_2) = \sum_{n_1=1}^{P_1} \omega_1^n B_h(x_1^n) B_{k;p}(x_1^n) J(x^n)$$

We applied methodology described in [1] to sum factorization algorithm, to obtain optimal scheduling and theoretical verification from trace theory method [3]. Next we made a series of numerical experiments to measure parallel performance. In Tables 1 and 2 we presented numerical results for both considered algorithms. We have obtained both maximum experimental speedup, as well as theoretical maximum combined one. In the Tables, $\mathcal{P}$ denotes the percentage of the algorithm which benefits from the parallel speedup, v is the number of threads, and $\mathcal{S}(v)$ is the measured speedup when using v threads.

We observed unexpected performance behaviour of the parallel sum factorization. Despite utilizing parallel loops across all elements, it was scaling only up to 4 cores. Beyond 4 cores, there was a plateau in speedup, indicating poor performance in multi-core environment.

The maximum speedup of the classical method, as shown in Table 1, mirrors findings from [1]. It is worth noting that sum factorization demands significantly more memory synchronization compared to the classical method.

Table 1. Classical integration method. Bottom index *i* stands for "inside element", *e* over all elements, and *c* combined.

p	t_{base}	v_i	$\mathcal{O}_i(v)$	$\mathcal{P}_i$	$\mathcal{S}_i(\infty)$	v_e	$\mathcal{S}_e(v)$	$\mathcal{P}_e$	$\mathcal{S}_e(\infty)$	$\mathcal{S}_c(\infty)$
1	0.08	6	1.35	0.31	1.45	3	2.5	0.9	10.00	14.50
2	0.91	6	2.32	0.68	3.15	6	5.4	0.98	45.00	141.75
3	6.70	9	4.01	0.84	6.43	8	7.8	0.98	52.00	314.36
4	36.56	12	5.64	0.90	9.75	10	7.8	0.97	31.91	311.12
5	143.86	12	4.6	0.85	6.84	12	11.29	0.99	174.92	1 196.45
6	562.03	11	6.98	0.94	17.36	12	11.15	0.99	144.29	1 984.07
7	1 622.20	12	7.28	0.94	16.97	12	10.75	0.99	94.60	1 605.36
8	4 586.81	12	5.16	0.88	8.30	12	10.88	0.99	106.86	886.94
9	11 752.54	12	8.54	0.96	27.15	12	10.44	0.99	73.62	1 998.78

Table 2. Sum factorization. Bottom index *i* stands for "inside element", *e* over all elements, and *c* combined.

p	t_{base}	v_i	$\mathcal{O}_i(v)$	$\mathcal{P}_i$	$\mathcal{S}_i(\infty)$	v_e	$\mathcal{S}_e(v)$	$\mathcal{P}_e$	$\mathcal{S}_e(\infty)$	$\mathcal{S}_c(\infty)$
1	0.05	1	1	0	1	2	1.5	0.67	3	3.00
2	0.29	1	1	0	1	4	2.9	0.87	7.91	7.91
3	1.38	1	1	0	1	4	2.9	0.87	7.91	7.91
4	5.12	10	1.16	0.15	1.18	4	3.3	0.93	14.14	16.69
5	15.15	11	1.34	0.28	1.39	4	3.5	0.95	21	29.19
6	40.63	10	1.44	0.34	1.51	4	3.5	0.95	21	31.71
7	109.47	11	1.68	0.45	1.80	4	3.2	0.92	12	21.60
8	202.17	9	1.52	0.38	1.63	4	3.4	0.94	17	27.71
9	394.28	10	1.63	0.43	1.75	4	3.5	0.96	21	36.75

Finally we evaluated the performance of classical integration and sum factorization in various scenarios by analysing the computational times from Tables 1 and 2. We focused on scenario of $p = 9$ as it was expected to be the optimal scenario for sum factorization. We examined three scenarios for a mesh size of 30^3:1) single-core CPU execution, 2) shared-memory CPU computation, and 3) GPU execution. The **classical integration** on a single core took 11 752.54 seconds, the 12-core OpenMP implementation took 1125.72 seconds, and the estimated **GPU implementation** was expected to take **5.87 seconds**. For **sum factorization** integration, the single-core execution took 394.28 seconds, the 4-core OpenMP implementation took 112.65 seconds, and the estimated **GPU implementation** was expected to take **10.78 seconds**.

4. Conclusions

Our approach validates the scheduling for integration algorithm by utilizing trace theory. We compared the integration algorithm's execution on both a CPU and GPU. We can extrapolate its scalability for various elliptic problems. Furthermore, the trace-theory based analysis of concurrency in the integration algorithm can be adapted to different integration methods. The methodology is versatile and can be expanded to include higher-dimensional spaces.

References

1. Szyszka A., Woźniak M., Schaefer R.: *Concurrent algorithm for integrating threedimensional B-spline functions into machines with shared memory such as GPU*. Computer Methods in Applied Mechanics and Engineering, 398, 2022, 115201.
2. Hiemstra R.R., Sangalli G., Tani M., Calabrò F., Hughes T.J.: *Fast formation and assembly of finite element matrices with application to isogeometric linear elasticity*. Computer Methods in Applied Mechanics and Engineering, 355, 2019, 234–260.
3. Woźniak M., Szyszka A., Rojas S.: *A study of efficient concurrent integration methods of B-Spline basis functions in IGA-FEM*. Journal of Computational Science, 64, 2022, 101857.
4. *Summit*, Oak Ridge National Laboratory, https://www.olcf.ornl.gov/summit/.
5. Diekert V., Rozenberg G.: *The Book of Traces*. World Scientific, 1995.
6. de Boor C.: *Subroutine package for calculating with B-splines*. SIAM Journal on Numerical Analysis, 14, 1971, 441–472.

Acknowledgements. The work has been supported by The European Union's Horizon 2020 Research and Innovation Program of the Marie Skłodowska-Curie grant agreement No. 777778, MATHROCKs. Scientific paper published within the framework of an international project co-financed with funds from the program of the Ministry of Science and Higher Education entitled „PMW" in years 2022-2023; contract no. 5243/H2020/2022/2.