AutoMT, a library for tensor operations and its performance evaluation for solid continuum mechanics applications

Hiroshi KAWAI*, Kohmei SATOH**, Yasunori YUSA***, Takayuki UOMOTO**, Ryuji SHIOYA† and Hiroshi OKADA***

* Department of Computer and Media Engineering, Faculty of Engineering, Tokyo University of Science, Suwa
5000-1 Toyohira, Chino, Nagano 391-0292, Japan
** Department of Mechanical Engineering, Graduate School of Science and Technology, Tokyo University of Science
2641 Yamazaki, Noda, Chiba 278-8510, Japan
*** Department of Mechanical Engineering, Faculty of Science and Technology, Tokyo University of Science
2641 Yamazaki, Noda, Chiba 278-8510, Japan
† Department of Information Sciences and Arts, Faculty of Information Sciences and Arts, Toyo University
2100 Kujirai, Kawagoe, Saitama 350-8585, Japan

E-mail: kohmei.satoh@gmail.com

Received 1 July 2015

Abstract
In this paper, AutoMT that is a library for tensor operations in 2-D/3-D spaces is presented. AutoMT stands for “Auto (Automation)”-“M (Matrix)”-“T (Tensor)” and is a kind of DSL (Domain Specific Language). DSL is a programming language specialized for a specific domain, such as graphics, communication, linear algebra, financial, chemical, etc. AutoMT is a library-based DSL/framework which is aimed at enhancing efficiencies in both the computations and the developments of computer programs involving many tensor operations. The paper describes the design, usage, and implementation of AutoMT, first. Then, the results of performance benchmark and an application example in large strain elastic-plastic analysis are presented. AutoMT accelerates the tensor operations that are often involved in the element-wise operations of finite element procedures. Usages of processor registers and SIMD (Single Instruction Multiple Data) mechanism are optimized to enhance the computational efficiencies in tensor operations. The performance of AutoMT is demonstrated by the results of performance benchmark test. It is shown that AutoMT at least doubles the performance. Finally, an example problem of finite strain elastic-plastic analysis is presented along with a small portion of example source program for elastic-plastic analysis. It is seen from the example program that it becomes much simpler than that without using AutoMT. Therefore, AutoMT can accelerate computer program developments. Finally, the problem of diffused necking is presented for a demonstration purpose.

Key words: Framework, Domain specific language, High performance computing, Supercomputing, Continuum mechanics, Solid mechanics

1. Introduction

When developing a program, and also if a specific type of knowledge about a given problem domain is required, it is very helpful if the corresponding software library and framework were available. Recently, DSL (Domain Specific Language) is appearing to ease such a programming task further. This is a programming language specialized for a specific domain, such as graphics, communication, linear algebra, financial, chemical, etc.

Here, we consider libraries, frameworks and DSL to develop a program of computational solid continuum mechanics whose formulae are mainly based on tensor operations. Especially, finite element (FE) applications are shown in this paper. Nowadays, it is very often required to incorporate non-linear capabilities, such as large deformation, large strain, plasticity, hyper-elasticity, contact, etc. In non-linear FE programs, a part of computation time may be generally dedicated to file I/O and a linear algebraic solver, either band, skyline, sparse direct, conjugate gradient or some others. Then, the rest
of the computation time is dominated by, so called, element-wise operations, which evaluate, for example, the stiffness matrix for each element and perform stress increment at each integration point of the element. In a practical situation, acceptable time for analysis (mostly, several hours or several days) is decided first, FE mesh is then designed to spend about half of the acceptable time in the linear algebraic solver usually; the other half is assigned to file I/O and element-wise operations. Coarser mesh causes bad accuracy, whereas finer mesh considerably increases computation time of a linear algebraic solver whose computational order is higher than file I/O and element-wise operations.

As for these libraries, frameworks and DSL, many systems for solving partial differential equations have been available. Most of them are based on finite difference method, but their FEM/FVM versions are also emerging in last ten years. They can be roughly classified into two groups, equation/formula-based high level system and stencil-based low level one.

The former group (Logg and Mardal, 2012) (Ootsuka and Takaishi, 2014) (FreeFem++) is classified as relatively high level. It allows a user to directly specify differential equations or variational forms. Some of the systems can handle not only scalar and vector quantities, but also tensor quantities. Because these systems generate code automatically by transforming the user-specified formula, however, it is difficult to control its implementation details, especially for the purpose of performance optimization. In terms of performance tuning, their main focus is currently limited on linear equation solvers only, ignoring element-wise operations.

The latter group (DeVito et al., 2011) (Liszter (Maruyama et al., 2011) deals with much lower level objects, and it allows a user to specify its dominant calculation patterns as stencil, the relation between grids, nodes, cells or elements. The user-specified stencil information can be used to select appropriate performance optimization strategies, typically related to thread parallelization and message passing pattern. They lead to the automatic generation of OpenMP and MPI-based codes. However, these systems cannot handle the detailed implementation of further lower level, where proposed library is mainly intended for.

As for tensor operations, there are some frameworks and DSL available. They are designed mainly for the quantum mechanics field (Hammond and Deprince, 2011) (Sanders et al., 2011). Instead, tensor operations may be substituted by matrix and vector of small size. For example, Matlab is intended for mathematical operations in general, but especially for linear algebra. NASTRAN DMAP is the scripting environment for a famous general purpose structural code, NASTRAN. It can be intended for users to customize its algorithms based on assembled global-scale matrices and vectors as a unit. All of them here are intended for relatively large-sized, higher dimensional matrix, vector or tensor. However, tensor operations used often in the solid/continuum mechanics field are limited to 3-D space mainly and sometimes even 2-D. Using these packages for this purpose is not only redundant but also less efficient.

In this paper, we propose “AutoMT,” a library for tensor operations in 2-D/3-D spaces, dedicated for the solid/continuum mechanics field. AutoMT stands for “Auto (Automation)”-”M (Matrix)”-”T (Tensor)” and is pronounced as a-te-mo-to which means cutlery close at hand in Japanese. It concretely means that AutoMT is a convenient tool to program (eat) tensor operations (food) in your program (by hand). It facilitates not only the ease of use, but also provides the code efficiency on modern scalar processors. The main features of AutoMT library are explained first, followed by its performance optimization aspect. Then, some performance benchmark results and its application to a more serious, practical research task are demonstrated.

2. AutoMT: library for tensor operations

AutoMT (Matrix and Tensor) library is designed for the manipulation of tensor objects in 2-D/3-D spaces, which are heavily utilized in the solid/continuum mechanics and non-linear finite element field. It can also handle small-sized matrix and vector. It is basically the substitution of originally 2-D and 3-D tensor operations. Their typical size is 3, 6, 9, etc. In this paper, we focus on the tensor operations of AutoMT. As for the tensor operations, AutoMT supports entities such as scalar, vector, 2nd-order tensor and 4th-order tensor.

Let us consider the following example. This is a simple formula using scalar, vector and tensor quantities.

\[ s = (aX) \cdot b \]  

(1)

Here, \( s \) is a scalar, \( a \) and \( b \) are vectors, and \( X \) is a 2nd-order tensor. This example requires a product operation between a vector and a tensor, followed by a dot product operation between two vectors. AutoMT supports both C and Fortran languages. Using C, AutoMT library is called in the following way.

```c
1 double a[3], X[3][3], b[3], s, tmp[3];
2```
It can also be written in Fortran, as follows.

```
real*8 a(3), X(3, 3), b(3), s, tmp(3)

call automt_prod_v_t_v (a, X, tmp)
call automt_cdot_v_v_s (tmp , b, s)
```

In both the cases, a vector is represented as a 1-D array variable, while a 2nd-order tensor is represented as a 2-D array variable.

Each operation corresponds to one library call to AutoMT. As the name of each function/subroutine, starting with the prefix, AutoMT, the name of the operation is specified, followed by the type identifiers of function/subroutine arguments in order. In the example program prod and cdot are the product of a 2nd-order tensor and a vector, and the dot product of two vectors, respectively. They are also presented in Table 1 in which currently supported tensor operations in AutoMT library are listed. Here, s, v and t denote scalar, vector and 2nd-order tensor, respectively. The supported data types in AutoMT library are shown in Table 2. Usually, the last argument corresponds to the result value of the operation. In this example, a temporary variable tmp is introduced to store the result value of the vector-tensor product operation, and it is soon reused as an argument of the vector-vector dot product operation.

There are a total of 27 operations are available in AutoMT library for the tensorial operations as listed in Table 1. The argument types are presented in Table 2. For further reference, the comprehensive list of currently supported functions for the tensorial operations is presented in Appendix. One can find that all the function names follow the way as mentioned above. Finally, it is noted that the strains are stored by their tensorial components. The engineering strains are not used.

### 3. AutoMT and high performance computing

AutoMT library not only facilitates the ease of use in programming tasks, but also helps the programmer to write highly efficient code. This library is mainly used to handle the tensor operations. They are often required to implement a non-linear finite element code for solid mechanics. They often appear in the element-wise operations, such as handling constitutive equations, so called D matrix, and evaluation of stress increment $\delta \sigma$.

On the other hand, through our experiences of making and running highly non-linear codes, it can be said that significant portions of the execution time are consumed not only by the linear equation solver, but also by the element-wise operations. Therefore, AutoMT library is designed to improve the performance of the latter part in these finite element codes. In modern computer architecture based on multi-core scalar processors, the key points to write high performance code can be summarized in cache memory and processor registers. In addition to them, SIMD (Single Instruction Multiple Data) mechanism is becoming the third important factor recently.

As for the element-wise operations, of these three factors, the first requirement, the efficient use of cache memory, is naturally satisfied, because its typical byte per FLOP ratio (B/F ratio), namely, the ratio between memory I/O and floating point operations, is always small. Therefore, it is essential to consider other two factors for the element-wise operations to be accelerated. However, even if the working data set fits on the cache, it does not mean that the code runs fast. The next step toward high performance code is, the effective utilization of the processor registers. From the programmer’s point of view, loop unrolling is the useful way of realizing it. Additionally, function/subroutine inlining may also be very effective. From our experiences, they can usually improve the performance from 2 to 3 times.

For example, let us consider a case of tensor-vector product, shown below.

\[
b = Xa
\]

Here, \(a\) and \(b\) are vectors, and \(X\) is a 2nd-order tensor, which is unsymmetric.

Without thinking about code efficiency, it can be easily implemented in C language as follows.

```
double X[3][3], a[3], b[3];

for (i = 0; i < 3; i++) {
    b[i] = 0.0;
}
```
for (j = 0; j < 3; j++) {
    b[i] += X[i][j] * a[j];
}

However, a compiler may produce very bad machine code in terms of computational efficiency, because the loop is too short (very few iteration count) and the inner-most loop block is too small. Unluckily, this code pattern is very typical in the computation of 3-D tensor operations. According to our experiences, especially in practical situations, most of compilers currently available hesitate to unroll such loops. Sometimes the compiler requests us to insert special compiler directives, which is not only tedious but also not portable among various kinds of compilers. Furthermore, it does not guarantee actual unrolling.

Therefore, rather than simply trusting the compiler and do nothing by ourselves, it is better to handle all of the issues by ourselves only. The performance of this code can be drastically improved by introducing full loop unrolling explicitly by hand, as follows.

double X_0_0, X_0_1, X_0_2;
double X_1_0, X_1_1, X_1_2;
double X_2_0, X_2_1, X_2_2;
double a_0, a_1, a_2, b_0, b_1, b_2;

b_0 = X_0_0 * a_0 + X_0_1 * a_1 + X_0_2 * a_2;
b_1 = X_1_0 * a_0 + X_1_1 * a_1 + X_1_2 * a_2;
b_2 = X_2_0 * a_0 + X_2_1 * a_1 + X_2_2 * a_2;

First, an array variable is converted into multiple variables, each of which corresponds to its array element. Then, the two short loops are fully unrolled. Now, the compiler has a freedom to allocate most of the variables onto processor registers. As a result, multiple floating point operations, especially fused multiply-add (FMA) instruction, can be issued and executed in parallel without any significant stall. In this stage, this code could achieve about 50 % of the peak performance on a scalar processor of the previous generation, which does not have SIMD yet.

Then, SIMD should be considered. SIMD instructions allow multiple floating point operations, of the same instruction type with different data, to be executed simultaneously, thus it is called Single Instruction Multiple Data (SIMD). To actually utilize these SIMD instructions, compiler-based loop vectorization is the key tool. However, the type of loops which a compiler can vectorize is limited. The most important limitation is that, only sequential memory access pattern is supported in the loop. Indirect index access, such as the code pattern like a[i] = b[ind[i]], which frequently appears to handle the relationship between nodes and elements, should not be used because of performance reasons. Nowadays in recent modern scalar processors, 2, 4 and 8-way SIMD are implemented, further boosting its performance by the factor of 2, 4 and 8.

To incorporate SIMD into the example above, a single loop which is the target of compiler-based vectorization is necessary. Then, assuming a more practical situation as an element-wise operation, let us add an outer element loop into the original code.

double X[NELM][3][3], a[NELM][3], b[NELM][3];

for (e = 0; e < nElm; e++) {
    for (i = 0; i < 3; i++) {
        b[e][i] = 0.0;
        for (j = 0; j < 3; j++) {
            b[e][i] += X[e][i][j] * a[e][j];
        }
    }
}

Here, nElm and NELM are the number of elements and the size of arrays, respectively. The compiler-driven vectorization capability, which produces SIMD instructions eventually, can be applied only to the inner-most loop. Unfortunately, in this example, the inner-most loop is j-loop, which is too short. Therefore, we need to exchange these loops so that the
inner-most loop becomes e-loop. The order of dimensions of the array variables should also be exchanged, so that the element-wise variable e sweeps these arrays actually in sequential order in the memory space.

```c
double X[3][3][NELM], a[3][NELM], b[3][NELM];
for (i = 0; i < 3; i++) {
    for (e = 0; e < nElm; e++) {
        b[i][e] = 0.0;
    }
}
for (j = 0; j < 3; j++) {
    for (e = 0; e < nElm; e++) {
        b[i][e] += X[i][j][e] * a[j][e];
    }
}
```

Then finally, by introducing the full loop unrolling technique, already mentioned before, here is the highly tuned, SIMD-ready code.

```c
double X_0_0[NELM], X_0_1[NELM], X_0_2[NELM];
double X_1_0[NELM], X_1_1[NELM], X_1_2[NELM];
double X_2_0[NELM], X_2_1[NELM], X_2_2[NELM];
da_0[NELM], a_1[NELM], a_2[NELM], b_0[NELM], b_1[NELM], b_2[NELM];
for (e = 0; e < nElm; e++) {
    b_0[e] = X_0_0[e] * a_0[e] + X_0_1[e] * a_1[e] + X_0_2[e] * a_2[e];
    b_1[e] = X_1_0[e] * a_0[e] + X_1_1[e] * a_1[e] + X_1_2[e] * a_2[e];
    b_2[e] = X_2_0[e] * a_0[e] + X_2_1[e] * a_1[e] + X_2_2[e] * a_2[e];
}
```

In the code example above, for each scalar, vector or tensor variable, NELM objects are allocated as a work area, and it could easily waste vast amount of memory. To save memory, the loop count of e-loop can be changed from nElm, the number of total elements, to a much smaller number, about tens or hundreds. It should be multiple of the number of SIMD-ways, for example, 64 or 128. Then, the loop is performed for each 64 or 128 elements, and the work area can be saved as well. Just before entering the loop and after exiting the loop, data should be copied between the original data structure and the work area. At the same time, to avoid indirect index access to nodal data, node-wise data should be rearranged to element-wise data. This technique is called “strip-mining.” Here is a pseudo-code of the example.

```c
double X_0_0[64], X_0_1[64], X_0_2[64]; /* work area for 64 elements only */
double X_1_0[64], X_1_1[64], X_1_2[64];
double X_2_0[64], X_2_1[64], X_2_2[64];
da_0[64], a_1[64], a_2[64], b_0[64], b_1[64], b_2[64];
for (e = 0; e < nElm; e += 64) { /* for each 64 elements */
    for (e2 = 0; e2 < 64; e2++) { /* indirect index access is allowed */
        a_0[e2] = node_a[ element_node_id[e + e2] ][0];
        b_0[e2] = X_0_0[e2] * a_0[e2] + X_0_1[e2] * a_1[e2] + X_0_2[e2] * a_2[e2];
    }
}
```

In this loop, SIMD-vectorization is performed.

```c
for (e2 = 0; e2 < 64; e2++) {
    /* ONLY sequential access to the work area is allowed */
    b_0[e2] = X_0_0[e2] * a_0[e2] + X_0_1[e2] * a_1[e2] + X_0_2[e2] * a_2[e2];
```

\( b_1[e2] = X_1_0[e2] * a_0[e2] + X_1_1[e2] * a_1[e2] + X_1_2[e2] * a_2[e2]; \)
\( b_2[e2] = X_2_0[e2] * a_0[e2] + X_2_1[e2] * a_1[e2] + X_2_2[e2] * a_2[e2]; \)

/* move from the work area back into node / element data */

for (e2 = 0; e2 < 64; e2++) {
    /* indirect index access is allowed */
    node_b[ element_node_id[e + e2] ][0] = b_0[e2];
}

If \( nElm \) is not multiple of 64, additional code to process the remaining elements is required.

A special version of AutoMT library implementation is also provided, according to the techniques demonstrated here in this example. In this version, functions/subroutines are implemented as C pre-processor macros, forcing the compiler to fully inline them without doubt.

4. Performance benchmark

Here in this section, the performance benchmark results of using AutoMT library are shown.

Three types of benchmark codes are prepared. The first one is an example of forming the element stiffness matrix of 3-D solid hexahedral element with isotropic linear elastic material in structural analysis. The second example is the code for making \( D \) matrix of elasto-plastic constitutive equation. Here, a consistent tangent matrix is formulated. The third example is for heat conduction analysis, and an element coefficient matrix of linear tetrahedral element is formulated through direct integration based on volume coordinate system.

In Table 3, values are the peak performance ratio (unit is \%). It is measured by the actual FLOPS value divided by the peak FLOPS value of the processor. Because all the code examples can utilize its cache memory efficiently, and also these element-wise operations are inherently parallel, we do not have to worry about multi-core performance using OpenMP.

Two types of scalar processors, Intel Core-i7 (Sandy Bridge) and Fujitsu SPARC VIII fx (Venus) are used. As compiler and its options, the former is with Intel Compiler and \(-O3 -xAVX\), while the latter is with Fujitsu Compiler and \(-Kfast\).

For each case, the original version uses double or triple loops, which are too short to optimize, while the tuned version is fully unrolled, SIMD-vectorized and also inlined. On Intel architecture, it seems like un-tuned code is still working well. However, on Fujitsu SPARC architecture, for example, using RIKEN K Computer or Oakleaf-FX in the Univ. of Tokyo, there is significant difference between two codes. With the code optimization techniques described in this paper, about 50 % or more of the peak performance of a scalar processor can be achieved in most of element-wise operations.

5. An example in finite strain elastic-plastic analysis

In this section, we show an example of a computer program using AutoMT. Some numerical results are also shown. A problem of defused necking of a prismatic bar is presented as the example. The part of computing the stress increment involves a numbers of equations (the Jaumann stress rate, radial return method, etc.) as seen in Hisada and Noguchi (1995). Among them, the equation expressing the equivalent plastic strain increment \( d\bar{\varepsilon}^p \) by the strain increment \( d\varepsilon \) is shown to be:

\[
d\bar{\varepsilon}^p = \frac{\sigma' \cdot C^e \cdot d\varepsilon}{2\sigma (G + \frac{\sigma}{3})}
\]  

(3)

where \( \sigma' \) is the deviatoric stress, \( C^e \) is the 4th-order tensor expressing the Hooke’s Law, \( \sigma \) is the equivalent stress, \( G \) is the shear modulus and \( \frac{\sigma}{3} \) is the hardening ratio. We implemented this equation with AutoMT as shown bellow.

1 double Strain_Increment[6], Deviatoric_Stress[6];
2 double C[6][6], G, Hardening_Ratio;
3 double Equivalent_Stress, Equivalent_Plastic_Strain_Increment;
4 double tmp_st_1[6], tmp_s_1, tmp_s_2;
Table 1 Supported operations in AutoMT. \( s \) is a scalar, \( a \) and \( b \) are vectors, \( X \) and \( Y \) are 2nd-order tensors, and \( A \) and \( B \) are 4th-order tensors. Names of operations which are the parts of the AutoMT function names are given in the parentheses.

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Vector</th>
<th>2nd-order tensor</th>
<th>4th-order tensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td></td>
<td>( X^T ) (transpose)</td>
<td>( A^{-1} ) (inv)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \det X ) (det)</td>
<td>( A - B ) (sub)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{tr} X ) (tr)</td>
<td>( A : X ) (colon)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{sym} X ) (sym)</td>
<td>( A : B ) (colon)</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td>( X + Y ) (add)</td>
<td>( A + B ) (add)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( X - Y ) (sub)</td>
<td>( A - B ) (sub)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( X: ) prod</td>
<td>( X : A ) (colon)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( X \cdot ) prod</td>
<td>( A : ) prod</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( X \times ) (times)</td>
<td>( A \times ) prod</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( X \odot Y ) (otimes)</td>
<td>( A \times Y ) (prod)</td>
</tr>
</tbody>
</table>

Table 2 Supported data types in AutoMT.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Identifier</th>
<th>Declaration in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>s</td>
<td>double s;</td>
</tr>
<tr>
<td>Vector</td>
<td>v</td>
<td>double a[3];</td>
</tr>
<tr>
<td>2nd-order tensor</td>
<td>t</td>
<td>double X[3][3];</td>
</tr>
<tr>
<td>2nd-order symmetric tensor</td>
<td>ttp</td>
<td>double X[3][3];</td>
</tr>
<tr>
<td>4th-order tensor</td>
<td>t4</td>
<td>double A[3][3][3][3];</td>
</tr>
<tr>
<td>4th-order minor symmetric tensor</td>
<td>mnst4</td>
<td>double A[6][6];</td>
</tr>
</tbody>
</table>

Table 3 Performance benchmark results (percent against the peak performance). It is shown that the use of AutoMT at least doubled the performance.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Intel (Sandy Bridge)</th>
<th>Fujitsu (SPARC VIIIx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>22 %</td>
<td>2 %</td>
</tr>
<tr>
<td>Plastic</td>
<td>12 %</td>
<td>10 %</td>
</tr>
<tr>
<td>Thermal</td>
<td>24 %</td>
<td>12 %</td>
</tr>
</tbody>
</table>

6 | AutoMT_colon_mnst4_st_st (C, Strain_Increment, tmp_st_1);
7 | AutoMT_colon_st_st_s (Deviatoric_Stress, tmp_st_1, &tmp_s_1);
8 | AutoMT_colon_st_st_s (Deviatoric_Stress, Deviatoric_Stress, &tmp_s_2);
9 | Equivalent_Stress = sqrt (tmp_s_2 * 3.0 / 2.0);
10 | tmp_s_2 = 2.0 * (G + Hardening_Ratio / 3.0) * Equivalent_Stress;
11 | Equivalent_Plastic_Strain_Increment = tmp_s_1 / tmp_s_2;

where \( \text{Strain}_\text{Increment}[6] \) and \( \text{Deviatoric}_\text{Stress}[6] \) express the strain increment and the deviatoric stress which are 2nd-order symmetric tensors. \( C[6][6] \) represents a 4th-order minor symmetric tensor which expresses the Hooke’s law for linear isotropic elasticity. \( G \), \( \text{Hardening}_\text{Ratio} \), \( \text{Equivalent}_\text{Stress} \) and \( \text{Equivalent}_\text{Plastic}_\text{Strain}_\text{Increment} \) are scalar variables expressing the shear modulus, the equivalent stress, the equivalent plastic strain increment and the hardening ratio, respectively. \( \text{tmp}_\text{st}_1[6] \), \( \text{tmp}_\text{s}_1 \) and \( \text{tmp}_\text{s}_2 \) are temporary variables.

If we implemented it without AutoMT, the computer program becomes much more complex, as

6 | for (i = 0; i < 6; i++) {
7 |   tmp_st_1[i] = 0.0;
8 | for (j = 0; j < 3; j++) {
9 |   tmp_st_1[i] += C[i][j] * Strain_Increment[j];
10 | }
11 | for (j = 3; j < 6; j++) {
12 |   tmp_st_1[i] += C[i][j] * Strain_Increment[j] * 2.0;
13 | }
14 |}
15 | tmp_s_1 = 0.0
16 | for (i = 0; i < 3; i++) {
17 |   tmp_s_1 += Deviatoric_Stress[i] * tmp_st_1[i];
18 | }
19 | for (i = 3; i < 6; i++) {
20 |   tmp_s_1 += Deviatoric_Stress[i] * tmp_st_1[i] * 2.0;
21 | }
22 | tmp_s_2 = 0.0
23 | for (i = 0; i < 3; i++) {
24 |   tmp_s_2 += Deviatoric_Stress[i] * Deviatoric_Stress[i];
25 | }
26 | for (i = 3; i < 6; i++) {

Next, we show a defused necking problem of a prismatic bar. Figures 1 (a) and (b) show the analysis model of the prismatic bar and its boundary conditions. The Young’s modulus and the Poisson’s ratio of the material are set to be 100 GPa and 0.3, respectively. The J2-flow isotropic hardening plasticity model is assumed. The yield stress \( \sigma_y \), the strength coefficient \( H \) and the one dimensional stress-plastic strain relationship are assumed to be 100 MPa, 20 MPa and \( \sigma = \sigma_y + H (\bar{\epsilon}^p)^\frac{1}{n} \) when \( \bar{\epsilon}^p \geq 0.01 \) (as depicted in Fig. 2). Figure 3 presents the finite element model for the prismatic bar containing initial imperfections: \( \Delta x = \frac{1}{20} (1 + \cos \frac{5}{10} \pi) \) to become defused necking.

The results are shown in Figs. 4 and 5. Figure 4 shows that plastic deformation concentrates at the center of the prismatic bar. Figure 5 shows that the hardening exponent \( n \) does not make much difference to the load-deformation behavior.
6. Conclusions

In this paper, the usefulness of AutoMT (a library for tensor operations in 2-D/3-D spaces) is demonstrated by presenting the performance benchmark and the application example in large strain elastic-plastic analysis. Engineers/researchers who are specialized in the field of continuum mechanics and/or solid/structural mechanics can use the AutoMT library not only to enhance the computational efficiency but also accelerate their program development works. It is generally difficult to write highly efficient computer programs as it takes so much care in every part of computer program. AutoMT hides the difficult part behind the supported AutoMT functions. Thus, the program developers can achieve two goals, i.e., efficiencies both in computations and program developments. Therefore, it is expected that the AutoMT library be used many engineers/researchers in developing high performance computer programs for the continuum mechanics and/or solid/structural mechanics problems.

AutoMT library, as well as the performance tuning techniques explained here, can be used in any scheme whose performance depends heavily on element-wise operations. As explained in this paper, typical non-linear analysis requires some amount of element-wise operations. However, some kinds of schemes, such as Euler-type explicit code and element-by-element type iterative solver, utilize element-wise operations intensively. For those applications, AutoMT should work fine.

Acknowledgements

A part of this work was performed as a part of the JST-CREST program on Development of a Numerical Library based on Hierarchical Domain Decomposition for Post Petascale Simulation. Investigation performed by H. Okada has partly been supported by Grant-in-Aid for Scientific Research (c) No. 25390157. The supports are gratefully acknowledged. Finally, the authors’ thanks are extended to the members of ADVENTURE Project headed by Professor S. Yoshimura of The University of Tokyo for useful discussions on high performance computing including AutoMT.

References


Appendix A. The list of AutoMT functions

There are a total of 76 AutoMT supported functions in C language, as listed in Table A.1.
Table A.1 The list of AutoMT functions in C language for the tensor operations listed in Table 1. In the function names, argument types are associated. They are, v: vector, t: 2nd-order tensor, st: 2nd-order symmetric tensor, t4: 4th-order tensor and mntst4: 4th-order minor symmetric tensor. The argument symbols correspond with the variables in the equations specifying the operation. Arguments are, 2nd-order tensor: X, Y, Z; 4th-order tensor: A, B, S, C; vector: a, b, scalar: s.

Operations on vectors

(Substitutions) a = 0: void AutoMT_zero_v (double a[3]); b = a: void AutoMT_assign_v_v (double a[3], double b[3]);

(Norm of vector) s = |a|: void AutoMT_norm_v_s (double a[3], double *s);

(Addition and subtraction) c = a + b: void AutoMT_add_v_v (double a[3], double b[3], double c[3]); / = a - b: void AutoMT_sub_v_v (double a[3], double b[3], double c[3]);

(Products of scalar and vector) b = sa: void AutoMT_prod_s_v_v (double s, double a[3], double b[3]); / b = as: void AutoMT_prod_v_s_v (double a[3], double s, double b[3]);

(Dot and vector products) s = a . b: void AutoMT_dot_v_v_s (double a[3], double b[3], double *s); / c = a . b: void AutoMT_times_v_v_v (double a[3], double b[3], double c[3]);

Operations on 2nd-order tensors

(Substitutions) \( X = 0 \): void AutoMT_zero_t (double X[3][3]); void AutoMT_zero_st (double X[6][6]); / \( Y = X \): void AutoMT_assign_t_t (double X[3][3], double Y[3][3]); void AutoMT_assign_st_st (double X[6][6], double Y[6][6]); void AutoMT_assign_t_st (double X[3][3], double Y[6][6]); void AutoMT_assign_st_t (double X[6][6], double Y[3][3]);

(Transpose) \( Y = X^t \): void AutoMT_transpose_t_t (double X[3][3], double Y[3][3]);

(Inverse) \( Y = X^{-1} \): void AutoMT_inv_t_t (double X[3][3], double Y[3][3]); void AutoMT_inv_st_st (double X[6][6], double Y[6][6]);

(Determinant) \( s = \text{det} X \): void AutoMT_det_t_s (double X[3][3], double *s); void AutoMT_det_st_s (double X[6][6], double *s);

(Symmetric part) \( Y = \text{sym} X \): void AutoMT_sym_t_t (double X[3][3], double Y[3][3]); void AutoMT_sym_st_st (double X[6][6], double Y[6][6]);

(Asymmetric part) \( Y = \text{asym} X \): void AutoMT_asym_t_t (double X[3][3], double Y[3][3]);

(Addition and subtraction) \( Z = X + Y \): void AutoMT_add_t_t (double X[3][3], double Y[3][3], double Z[3][3]); void AutoMT_add_st_st (double X[6][6], double Y[6][6], double Z[6][6]); / \( Z = X - Y \): void AutoMT_sub_t_t (double X[3][3], double Y[3][3], double Z[3][3]); void AutoMT_sub_st_st (double X[6][6], double Y[6][6], double Z[6][6]);

(Products of tensor and scalar) \( Y = \sigma X \): void AutoMT_prod_s_t_t (double s, double X[3][3], double Y[3][3]); void AutoMT_prod_s_st_st (double s, double X[6][6], double Y[6][6]); / \( Y = \sigma X \): void AutoMT_prod_t_s_t (double X[3][3], double s, double Y[3][3]); void AutoMT_prod_st_s_st (double X[6][6], double s, double Y[6][6]);

(Products of tensor and vector) \( b = \sigma a \): void AutoMT_prod_t_v_v (double X[3][3], double a[3], double b[3]);

(Products of tensors) \( Z = XY \): void AutoMT_prod_t_t_t_t (double X[3][3], double Y[3][3], double Z[3][3]); void AutoMT_prod_t_t_st_t (double X[6][6], double Y[6][6], double Z[6][6]); / \( Z = XY^T \): void AutoMT_prod_t_t_t_t (double X[3][3], double Y[3][3], double Z[3][3]); void AutoMT_prod_t_t_st_t (double X[6][6], double Y[6][6], double Z[6][6]);

(Products of tensor and scalar) \( Y = \sigma X \): void AutoMT_prod_s_t_t (double s, double X[3][3], double Y[3][3]); void AutoMT_prod_s_st_st (double s, double X[6][6], double Y[6][6]); / \( Y = \sigma X \): void AutoMT_prod_t_s_t (double X[3][3], double s, double Y[3][3]); void AutoMT_prod_st_s_st (double X[6][6], double s, double Y[6][6]);


(Inverse) \( B = A^{-1} \): void AutoMT_inv_nst4_mntst4 (double A[3][3][3][3], double B[3][3][3][3], double C[3][3][3][3], double D[3][3][3][3]);

(Addition and subtraction) \( C = A + B \): void AutoMT_add_t4_t4_t4_t4 (double A[3][3][3][3], double B[3][3][3][3], double C[3][3][3][3], double D[3][3][3][3]); void AutoMT_add_nst4_mntst4 (double A[6][6][6][6], double B[6][6][6][6], double C[6][6][6][6], double D[6][6][6][6]); / \( C = A - B \): void AutoMT_sub_t4_t4_t4_t4 (double A[3][3][3][3], double B[3][3][3][3], double C[3][3][3][3], double D[3][3][3][3]); void AutoMT_sub_mntst4_nst4_mntst4 (double A[6][6][6][6], double B[6][6][6][6], double C[6][6][6][6], double D[6][6][6][6]);

(Products of tensor and scalar) \( B = \sigma A \): void AutoMT_prod_s_t4_t4 (double s, double A[3][3][3][3], double B[3][3][3][3], double C[3][3][3][3], double D[3][3][3][3]); void AutoMT_prod_s_mntst4_mntst4 (double s, double A[6][6][6][6], double B[6][6][6][6], double C[6][6][6][6], double D[6][6][6][6]);

(Products of tensors) \( Y = A X \): void AutoMT_colon_t4_t_t (double A[3][3][3][3], double X[3][3], double Y[3][3]); void AutoMT_colon_mntst4_mntst4 (double A[6][6][6][6], double X[6][6][6][6], double Y[6][6][6][6]); / \( Y = X A \): void AutoMT_colon_t4_t_t (double X[3][3][3][3], double A[3][3][3][3], double B[3][3][3][3], double C[3][3][3][3], double D[3][3][3][3]); void AutoMT_colon_mntst4_mntst4 (double X[6][6][6][6], double A[6][6][6][6], double B[6][6][6][6], double C[6][6][6][6], double D[6][6][6][6]);