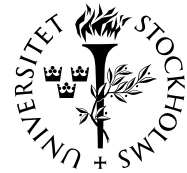




KUNGL
TEKNISKA
HÖGSKOLAN



Department of Numerical Analysis and Computer Science
TRITA-PDC-2002:1 • ISRN KTH/PDC/R--02/1--SE • ISSN 1401-2731

Yee_bench – A PDC benchmark code

Ulf Andersson, PDC, KTH

Center for Parallel Computers

Ulf Andersson, PDC, KTH
Yee_bench – A PDC benchmark code

Report number: TRITA-PDC-2002:1, ISRN KTH/PDC/R--02/1--SE
Publication date: November 2002
E-mail of author: ulfa@pdc.kth.se

Department of Numerical Analysis and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm
SWEDEN
Fax: +46 8 247 784

Abstract

We will describe the code `Yee_bench`. This serial benchmark code has been developed at PDC and the Parallel and Scientific Computing Institute (PSCI). `Yee_bench` is a memory bound code.

Results for `Yee_bench` on some of the computers available at PDC are presented. We demonstrate that the performance of `Yee_bench` can be predicted using the results from the DAXPY part of the `stream2` memory bandwidth benchmark code.

TRITA-PDC-2002:1 • ISRN KTH/PDC/R--02/1--SE • ISSN 1401-2731

1 Introduction

The `Yee_bench` code is a benchmarking code developed at PDC and the Parallel and Scientific Computing Institute (PSCI¹). It implements the kernel of the finite-difference time-domain (FDTD) method [5] in Computational Electromagnetics (CEM). FDTD is the method used on the structured part of the grid in hybrid time-domain code developed within the PSCI project General ElectroMagnetic Solvers (GEMS). The author worked on the GEMS project during his Ph.D. studies [1].

The FDTD method was introduced by Yee [6] in 1966, and is thus sometimes referred to as the Yee scheme. It is based on central differences on staggered Cartesian grids and is second-order accurate in both space and time.

I will use the following definitions: 1 kbyte = 1024 bytes, 1 Mbyte = 1024 kbytes, and 1 Gbyte = 1024 Mbytes. When I mean 1 000 000 bytes, I will simply write one million bytes.

2 The FDTD method

In this section, we will give a brief description of the FDTD method for the Maxwell equations. A more detailed description may be found in the books about FDTD [5, 3, 4]. In CEM, the acronym FDTD refers to one particular finite-difference method, namely the leap-frog method on staggered Cartesian grids.

The FDTD method is derived by applying central differences on Ampère's and Faraday's laws:

$$\left\{ \begin{array}{l} \epsilon \frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma E_x, \\ \epsilon \frac{\partial E_y}{\partial t} = \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma E_y, \\ \epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma E_z, \\ \mu \frac{\partial H_x}{\partial t} = \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y}, \\ \mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}, \\ \mu \frac{\partial H_z}{\partial t} = \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x}. \end{array} \right. \quad (1)$$

Consider a computational domain of size (l_x, l_y, l_z) , *i.e.*, a box. We divide this domain into (N_x, N_y, N_z) equal sized cells. The size of these cells is $(\Delta x, \Delta y, \Delta z) =$

¹<http://www.psci.kth.se/>

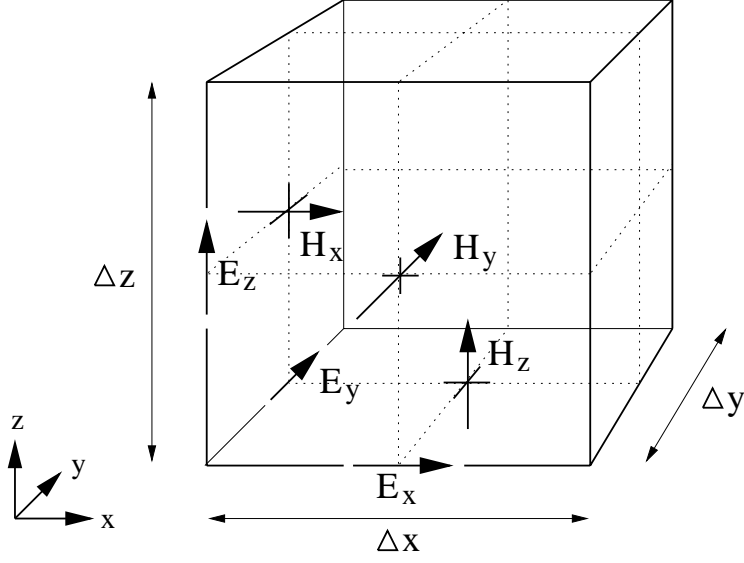


Figure 1. Positions of the electric and magnetic field components in a Yee cell. (Figure courtesy of Erik Abenius, NADA, KTH.)

$(l_x/N_x, l_y/N_y, l_z/N_z)$. Our computational grid is defined by:

$$\begin{aligned}
E_x|_{i+\frac{1}{2},j,k}^n, & \quad i = 1, \dots, N_x, \quad j = 1, \dots, N_y + 1, \quad k = 1, \dots, N_z + 1, \\
E_y|_{i,j+\frac{1}{2},k}^n, & \quad i = 1, \dots, N_x + 1, \quad j = 1, \dots, N_y, \quad k = 1, \dots, N_z + 1, \\
E_z|_{i,j,k+\frac{1}{2}}^n, & \quad i = 1, \dots, N_x + 1, \quad j = 1, \dots, N_y + 1, \quad k = 1, \dots, N_z, \\
H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n-\frac{1}{2}}, & \quad i = 1, \dots, N_x + 1, \quad j = 1, \dots, N_y, \quad k = 1, \dots, N_z, \\
H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n-\frac{1}{2}}, & \quad i = 1, \dots, N_x, \quad j = 1, \dots, N_y + 1, \quad k = 1, \dots, N_z, \\
H_z|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n-\frac{1}{2}}, & \quad i = 1, \dots, N_x, \quad j = 1, \dots, N_y, \quad k = 1, \dots, N_z + 1,
\end{aligned} \tag{2}$$

where $n = 0, \dots, N_t$ for all six components, and N_t is the number of time steps. One cell of this grid is displayed in Figure 1. Note that all the electromagnetic field components are defined at different locations, *i.e.*, the grid is staggered.

In homogeneous materials with $\sigma = 0$, the following formulas comprise the FDTD updating stencils for the electromagnetic field components:

$$\begin{aligned}
H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} = H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n-\frac{1}{2}} & + \frac{\Delta t}{\mu\Delta z} \left[E_y|_{i,j+\frac{1}{2},k+1}^n - E_y|_{i,j+\frac{1}{2},k}^n \right] \\
& - \frac{\Delta t}{\mu\Delta y} \left[E_z|_{i,j+1,k+\frac{1}{2}}^n - E_z|_{i,j,k+\frac{1}{2}}^n \right]
\end{aligned} \tag{3}$$

$$\begin{aligned}
H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}} &= H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n-\frac{1}{2}} + \frac{\Delta t}{\mu\Delta x} \left[E_z|_{i+1,j,k+\frac{1}{2}}^n - E_z|_{i,j,k+\frac{1}{2}}^n \right] \\
&\quad - \frac{\Delta t}{\mu\Delta z} \left[E_x|_{i+\frac{1}{2},j,k+1}^n - E_x|_{i+\frac{1}{2},j,k}^n \right]
\end{aligned} \tag{4}$$

$$\begin{aligned}
H_z|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} &= H_z|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n-\frac{1}{2}} + \frac{\Delta t}{\mu\Delta y} \left[E_x|_{i+\frac{1}{2},j+1,k}^n - E_x|_{i+\frac{1}{2},j,k}^n \right] \\
&\quad - \frac{\Delta t}{\mu\Delta x} \left[E_y|_{i+1,j+\frac{1}{2},k}^n - E_y|_{i,j+\frac{1}{2},k}^n \right]
\end{aligned} \tag{5}$$

$$\begin{aligned}
E_x|_{i+\frac{1}{2},j,k}^{n+1} &= E_x|_{i+\frac{1}{2},j,k}^n - \frac{\Delta t}{\epsilon\Delta z} \left[H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}} - H_y|_{i+\frac{1}{2},j,k-\frac{1}{2}}^{n+\frac{1}{2}} \right] \\
&\quad + \frac{\Delta t}{\epsilon\Delta y} \left[H_z|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} - H_z|_{i+\frac{1}{2},j-\frac{1}{2},k}^{n+\frac{1}{2}} \right]
\end{aligned} \tag{6}$$

$$\begin{aligned}
E_y|_{i,j+\frac{1}{2},k}^{n+1} &= E_y|_{i,j+\frac{1}{2},k}^n - \frac{\Delta t}{\epsilon\Delta x} \left[H_z|_{i+\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} - H_z|_{i-\frac{1}{2},j+\frac{1}{2},k}^{n+\frac{1}{2}} \right] \\
&\quad + \frac{\Delta t}{\epsilon\Delta z} \left[H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} - H_x|_{i,j+\frac{1}{2},k-\frac{1}{2}}^{n+\frac{1}{2}} \right]
\end{aligned} \tag{7}$$

$$\begin{aligned}
E_z|_{i,j,k+\frac{1}{2}}^{n+1} &= E_z|_{i,j,k+\frac{1}{2}}^n - \frac{\Delta t}{\epsilon\Delta y} \left[H_x|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} - H_x|_{i,j-\frac{1}{2},k+\frac{1}{2}}^{n+\frac{1}{2}} \right] \\
&\quad + \frac{\Delta t}{\epsilon\Delta x} \left[H_y|_{i+\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}} - H_y|_{i-\frac{1}{2},j,k+\frac{1}{2}}^{n+\frac{1}{2}} \right]
\end{aligned} \tag{8}$$

These formulas are used for all interior field components. Note that the resulting method is explicit, *i.e.*, any value on a particular time level only depends on values on earlier time levels. The FDTD method is second-order accurate in both time and space.

Formulas (3)–(8) are constructed under the assumption of a homogeneous media. In this case, the Maxwell equations can be solved analytically. However, formulas (3)–(8) can easily be adapted to inhomogeneous materials. This can be achieved by allowing ϵ and μ to be space dependent. How to choose the discrete values of ϵ and μ are discussed in detail in Chapter 8 of [1].

A special, idealized, case of an inhomogeneous medium is the perfect electric conductor (PEC). A PEC object is characterized by a vanishing tangential electric field at its surface. In the FDTD method, a PEC object is modeled by first updating all electric field components using (6)–(8) and then zeroing all electric field components on the surface of the PEC object. This means that we must remodel the PEC object to fit into the Cartesian grid. This need for staircasing of smoothed objects is one of the major drawbacks of the FDTD method.

We also need initial conditions, boundary conditions and excitation. The choices of these for `Yee_bench` is described in Section 3.2.

3 Yee_bench

3.1 Model problem

We use a cubic computational domain, *i.e.*, $N_x = N_y = N_z \equiv N$.

The electromagnetic field is excited with a point source (a dipole) in the center of the computational domain. A Dirichlet boundary condition is used at the outer boundary where a Perfect Electric Conductor (PEC) is assumed. This means that the (tangential) electric field components at the outer boundary are set to zero. The choice of outer boundary condition and excitation is based on a desire to minimize the extra work in each time step.

We also need initial conditions for the electromagnetic fields. We simply set all initial fields to zero, *i.e.*, we set $\mathbf{E}(t=0) = 0$ and $\mathbf{H}(t=-\Delta t/2) = 0$ and then use (3)–(8) to compute the fields for $t = \Delta t/2, \Delta t, 3\Delta t/2, \dots, N_t\Delta t$.

3.2 Code structure

The complete `Yee_bench` code exists in one Fortran 90 version and one C version. We will here concentrate on the Fortran 90 version. This code is written in standard-conforming Fortran 90 to facilitate portability.

`Yee_bench` implements the FDTD method for a cubic domain with varying number of cells, N . For each N , N_t time steps are taken and an average time per time step is computed. The first time step is excluded from these timings in order to avoid initialization overhead. As the problem size grows N_t is automatically decreased so that each problem size N takes about twenty seconds to complete.

For each problem size, it is possible to repeat the timestepping several times. The fastest time is then reported. The value of N_t for the first value of N is given by the user in the input file, which must be called `yee.dat`. Start and stop value for N , and the number of repetitions (`number_of_runs`) for every N value is also set in this file.

Appendix A contains a shortened version of the code. In Appendix A, we have written the field variable updates using array syntax. Unless otherwise stated, all results presented herein, have in fact been implemented as fused do-loops (see Appendix B) in order to maximize register reuse. This choice is controlled by a parameter (`calcmet`) in the input file. The different options for (`calcmet`) are summarized in Table 1.

Table 1. Three different implementations of the leap-frog update.

<code>calcmet</code>	Method
3	array syntax
4	three separate loops
5	fused do loops

Note that the updates are performed in a separate module. This is illustrated with the following call graph generated by FORESYS:

```
+yee_bench:
|  update_mod%update_init
|  timer
|  update_mod%updateh_homo
|  update_mod%updatee_homo
```

3.3 Memory requirements

The memory usage with 64-bit precision is $24N^3 + 24(N+1)^3$ bytes (*cf.* with the allocation in Appendix A). Note that no padding has been used when allocating the fields. However, padding may be added by altering the value of variables `pad[xyz]` [EH] in the module `parameter_mod`. Without padding we will typically experience dips in performance when N or $N + 1$ is a power-of-two. We refrain from using padding in order to study the performance effects of this case.

Table 2 lists how large cubic problems it is possible to solve for a given memory size. The sizes corresponds to cache and main memory sizes on some of the computers used in Section 4.

Table 2. Maximum problem size vs available memory for `Yee_bench`.

128 kbyte	2 Mbyte	4 Mbyte	1 Gbyte	2 Gbyte	4 Gbyte
$N = 13$	$N = 34$	$N = 43$	$N = 281$	$N = 354$	$N = 446$

3.4 Validation case

To verify that the code produces correct results, we use the case $N = 100$, $N_t = 200$ and $\Delta_x = \Delta_y = \Delta_z = 1$. The sum of the Ez components should equal -0.0692134.

3.5 Comments on expected performance

By examining the code in Appendices A and B, we see that 36 floating-point operations (twelve additions, twelve subtractions and twelve multiplications) are performed in each cell every time step. Most modern computers need to have a perfect balance between the number of multiplications and the number of additions/subtraction to be able to reach peak performance. This means that we cannot expect to reach more than 75 % of the peak performance, even if we have unlimited memory bandwidth. In reality the performance is considerably lower than this due to limitations in memory bandwidth.

If we count the number of load, stores, and arithmetic operations, we see that `Yee_bench` is indeed memory bound. Assuming that we use fused do loops, we need twenty loads and six stores per cell in each time step. Appendix B explains this in more detail. We summarize in Table 3.

Table 3. Floating-point and memory operations per cell and time step for `Yee_bench`.

adds/subs	mults	stores	loads
24	12	6	20

If the memory bandwidth, bw , is known (in bytes/s), we can estimate the best possible performance, P , (in Flop/s) with 64-bit precision for large problems with

$$P_l = \frac{24 + 12}{8 * (6 + 20)} bw = \frac{9}{52} bw \approx 0.173bw. \quad (9)$$

Here we have ignored the presence of caches. Hence, (9) is expected to underestimate the performance. If we assume a very fast and rather large cache, we can neglect 8 of the 20 loads. We then get

$$P_h = \frac{24 + 12}{8 * (6 + 12)} bw = \frac{1}{4} bw. \quad (10)$$

We expect the actual performance for large problem sizes to be between P_l and P_h .

The actual memory bandwidth can be measured with the benchmark code `stream2`², which is designed so that cache lines are never reused if the array sizes are large enough. `stream2` contains four different ways to measure the bandwidth:

```
FILL:  a(i) = 0
COPY:  a(i) = b(i)
DAXPY: a(i) = a(i) + q*b(i)
DOT:   sum += a(i) * b(i)
```

These do not give the same result. We use the DAXPY results when computing P_l and P_h , because it has two loads per each store, which is similar to `Yee_bench` which has ten loads per three stores (see Table 3).

3.6 Comments on relevant problem sizes

In many practical computations with the FDTD method, it is memory rather than time that limits the computation. It is often crucial to be able to use all the available memory on a computer. Hence, performance for large problem sizes are much more relevant than performance for small problem sizes, i.e. problems that fit into cache.

On the other hand, FDTD is an explicit timestepping method and therefore rather straightforward to parallelize with MPI. Hence on shared memory nodes, it is not relevant to use the entire memory for one process.

3.7 OpenMP

`Yee_bench` is equipped with OpenMP directives. However, all results in this report are for single thread jobs.

²<http://www.cs.virginia.edu/stream/stream2/>

3.8 Single stream and multiple stream definitions

We use the same definition of *single stream* as in [2]. With single stream performance we mean that only one user program is running when its performance is measured.

A *multiple stream* benchmark means, in this paper, that on a shared memory machine with P CPUs, we (almost) simultaneously start P identical processes. We measure the performance of all these processes until one of them is killed by the OS when memory is exhausted. We then compare the average performance of these processes with the single stream performance. This quotient is usually significantly less than one, because `Yee_bench` is a memory bound code.

3.9 Code versions

There are many different versions of `Yee_bench`. As mentioned earlier, there is a C version. There are also Fortran 77, Matlab, and C++ (under development) versions. Furthermore, there are five more Fortran 90 versions, which use different ways to store the electromagnetic fields. They are discussed in Appendix C.

The C version is the only code version besides the Fortran 90 code that also has looping over problem sizes built into it.

All these codes, except the C++ version, are available on request.

3.10 Parallel FDTD benchmarking codes

There are two related FDTD benchmarking codes used at PDC.

1. `pscyee`: This is a parallel implementation of the FDTD kernel. Parallelization has been done with MPI using point-to-point communication. This code is the suggested solution to the CEM test case of the PDC summer school. It is available upon request.
2. `GemsTD`: This is the hybrid time-domain solver developed within the GEMS project. `GemsTD` is sometimes called `frida`. Presently, all benchmarking cases only involve FDTD and not the finite element-method used on the unstructured part of the grid. This code is NOT available for distribution.

4 Results

We will now present results for some of the computers available at PDC. We have in all cases used 64-bit precision, which is the precision used in `GemsTD`. All results are single stream results, unless otherwise stated. We have started at $N = 10$ and used `number_of_runs=1` in all cases.

4.1 IBM

The hardware and software used in these performance evaluations are listed in Table 4. The pwr4 node was made available through the Nordic Grid Consortium (NGC) cooperation with the Scientific Computing Ltd (CSC³) in Finland. Runs on the pwr4 node were performed around September 17th.

Table 4. Machine characteristics for the used hardware and software.

Processor	pwr4	pwr3	pwr3	pwr2
GHz	1.1	0.375	0.222	0.16
Gbytes	32	16	4	0.25
Peak Gflop/s (per CPU)	4.4	1.5	0.888	0.64
CPU/node	32	16	8	1
Compiler	xlf90_r	xlf90	xlf90	xlf90
Compiler version	8.1.0.1	7.1.1.3	7.1.1.3	6.1.0.3
Compiler options	-q64 ^a	-qhot ^{a b}	-qhot ^{a b}	^a
L1 cache (I+D)	96k	32k+64k	32k+16k	32k+128k
L2 cache	1536k ^c	4096k	4096k	256k
L3 cache	128M ^d	-	-	-
Yee_bench (Gflop/s)	0.36	0.28	0.20	0.18
Yee_bench perf. rel. peak perf.	8.2%	18.7%	22.5%	28.6%
Bandwidth (bw , 10^6 byte/s) ^e	1777	1165	783	1153
$P_l \approx 0.173 bw$ (Gflop/s)	0.31	0.20	0.13	0.20
$P_h = 0.250 bw$ (Gflop/s)	0.44	0.29	0.20	0.29

^a and `-qarch=auto -qtune=auto -O3 -qalias=noaryovrlp`, where the latter option only affects the performance of `calcmets=3`.

^b `-bmaxdata:0x80000000` used when linking

^c Shared by two CPUs.

^d 128 Mbyte per MCM. (MCM=MultiChip Module)

^e Measured by the `stream2` DAXPY benchmark.

The result is displayed in Figure 2. For the pwr4, poor performance occurs whenever N or $N + 1$ contain at least four factors that are two. On the pwr2 and pwr3, poor performance occurs whenever N or $N + 1$ contain at least six factors that are two. These dips in performance are caused by an increase in cache misses. They can easily be alleviated by padding the arrays.

³<http://www.csc.fi/>

The performance numbers presented for `Yee_bench` in Table 4 have been calculated by first taking the mean value for problem sizes larger than one Gbyte ($N > 281$). Second, we exclude all values that are below 75% of the original mean value and calculate a new mean value from the remaining values. (The second step is taken in order to compensate for not using any padding.) In the `pwr2` case we used the limit $N > 140$.

In Figure 2 we see that the `pwr4` performance drops at around $N = 31$ (1.5 Mbyte), when the entire problem no longer fit into L2 cache. Another performance drop can be seen at $N = 140$ (128 Mbyte). This corresponds to the size of the L3 cache on the MCM. Because all other CPUs are idle, `Yee_bench` has exclusive access to the L3 cache. It is not clear to us why this drop in performance seems to start at $N = 120$.

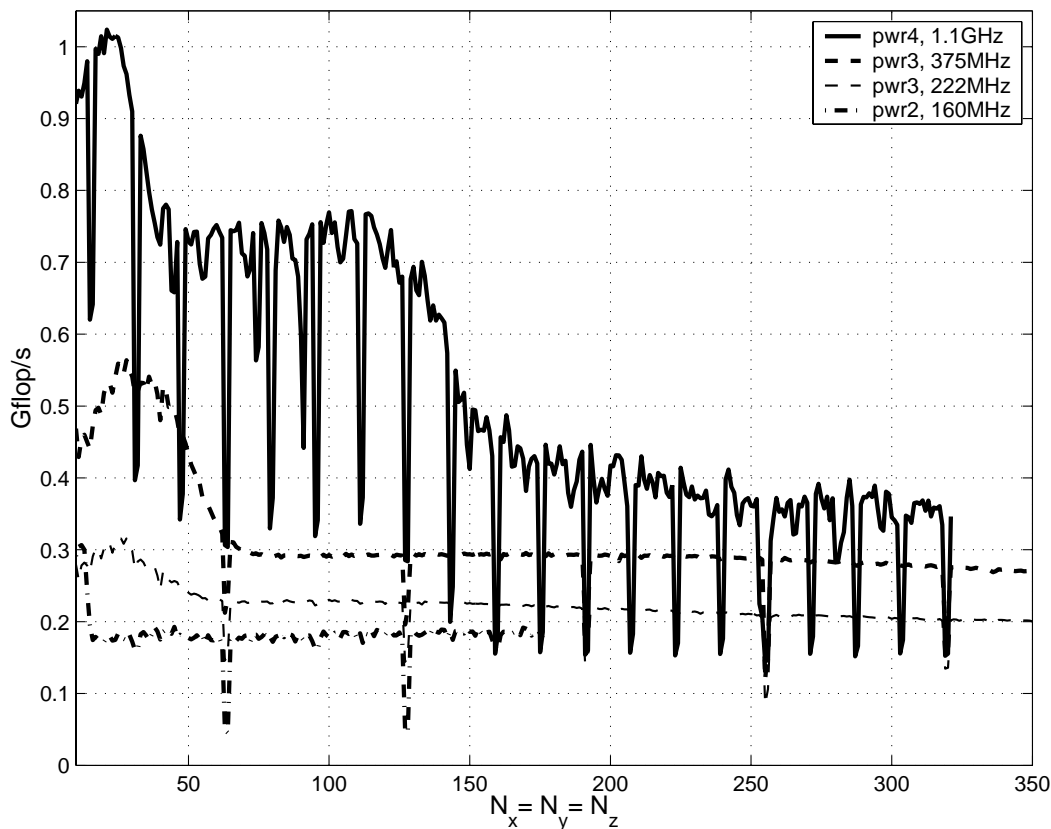


Figure 2. `Yee_bench` performance on different IBM CPUs.

Figures 3 and 4 display multiple stream results. Because `Yee_bench` is memory bound, there will be competition for the memory bandwidth if we start N processes on a shared memory node with N CPUs. Figures 3 and 4 shows the average speed of the N processes as fractions of the speed of a single stream process.

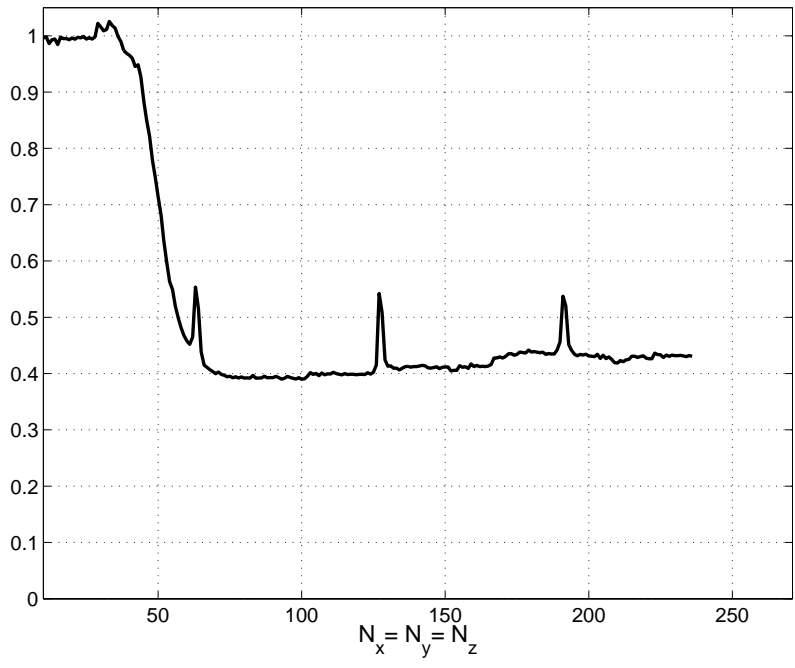


Figure 3. Performance decrease on a pwr3 (375MHz) node when sixteen processes are running.

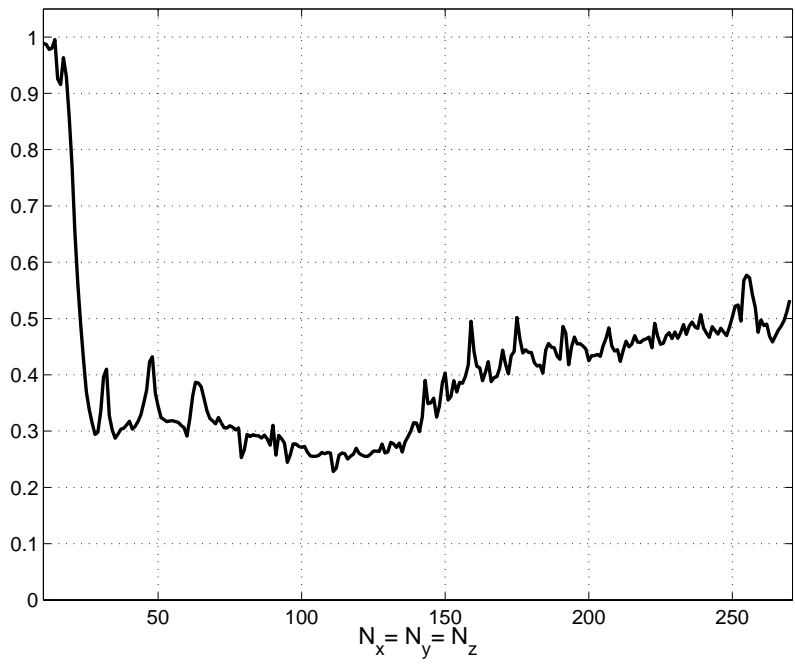


Figure 4. Performance decrease on a pwr4 node when 32 processes are running.

4.2 SGI

The computer *boye* is a Silicon Graphics (SGI) dual-rack Onyx2 with 12 R10000 CPUs (195 MHz) and four Gbyte memory. It is primarily used for the VR-Cube at PDC (VR=Virtual Reality). It has an L1 instruction cache with 32 kbytes, an L1 data cache with 32 kbytes, and a unified instruction/data L2 cache with 4 Mbytes. We used the f90 MIPSpro Compiler, Version 7.30 and the optimization options `-mips4 -O3`.

The result is displayed in Figure 5. The performance is low. However, the machine is several years old. Again, we see a drop in performance when the entire problem no longer fits into L2 cache ($N = 43$). There is also a severe dip in performance at $N = 31$ and $N = 63$. Smaller dips occur whenever $N + 1$ contains at least four factors of two ($N = 15, 47, 79, 95, 111$).

We also see that the fused do-loop implementation is slightly faster than the other two implementations.

The `stream2` DAXPY performance is 449 million bytes per second. This gives $P_h = 0.11$ Gflop/s and $P_l = 0.078$ Gflop/s. The actual `Yee_bench` performance for `calcmnt=5` is only 0.061 Gflop/s. Theoretical peak performance is 0.400 Gflop/s, *i.e.*, we get 15% of the theoretical peak performance.

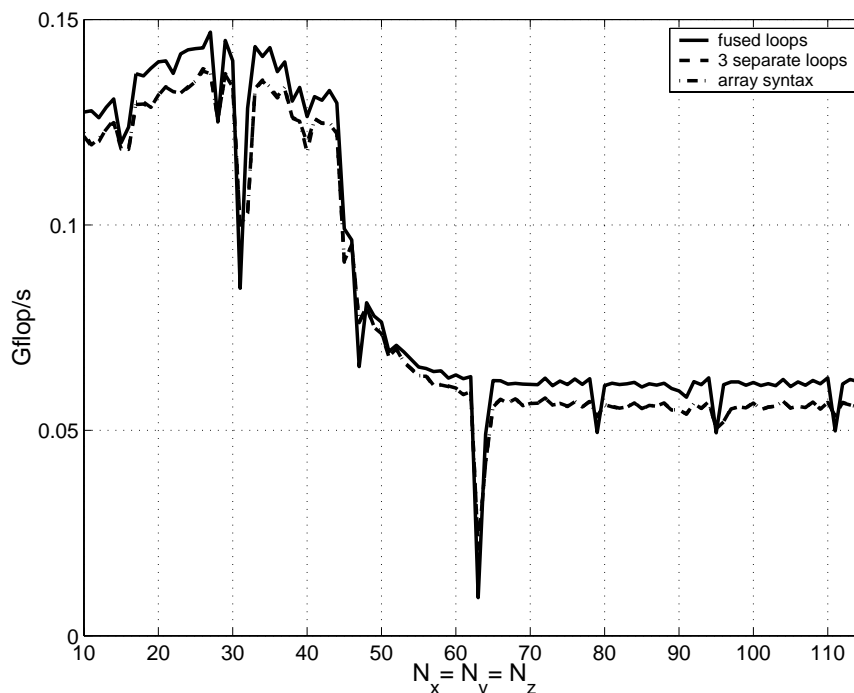


Figure 5. `Yee_bench` performance on *boye*. The curves for 3 separate loops and array syntax are almost identical. Differences are, with one exception ($N = 40$), less than one Mflop/s.

4.3 Fujitsu VX

The computer *Selma* is a Fujitsu VX/3. It has three Processing Elements (PE). The PE consists of a Scalar Unit and a Vector Unit. Each PE has a theoretical peak performance of 2.2 Gflop/s and 2 Gbyte memory.

We used the compiler *f90*, version Fujitsu Fortran90/VP Compiler Driver L00021 (Feb 1 2000 17:27:50), and the options `-Sw -Oe -X9 -Am`. The result is displayed in Figure 6. The behavior here is completely different from previous results. Performance increases with N . This is due to the vectorization of the inner loop. The Fujitsu can take advantage of long vector lengths. Hence, higher N_x -values give better performance.

One way to boost the performance is to use a very elongated domain. Results for such domains are given in Table 5. We see that we reach almost 1.5 Gflop/s. This is 67% of the theoretical peak performance. This is the closest to the maximum of 75% of theoretical peak performance (see Section 3.5) we have seen. Another vector computer, the CRAY J90, previously used at PDC gave around 50% for a similar code (see page 48 in [1]).

The *stream2* DAXPY performance is approximately $12 * 10^3$ million bytes per second. This gives $P_h = 3.0$ Gflop/s and $P_l = 2.1$ Gflop/s. Obviously these estimates are not valid because the memory bandwidth is too high.

Table 5. Performance results on *selma* for elongated computational domains.

N_x	N_y	N_z	Gflop/s
800	100	100	1.36
2 000	100	100	1.45
200 000	10	10	1.46
2 000 000	1	1	1.48

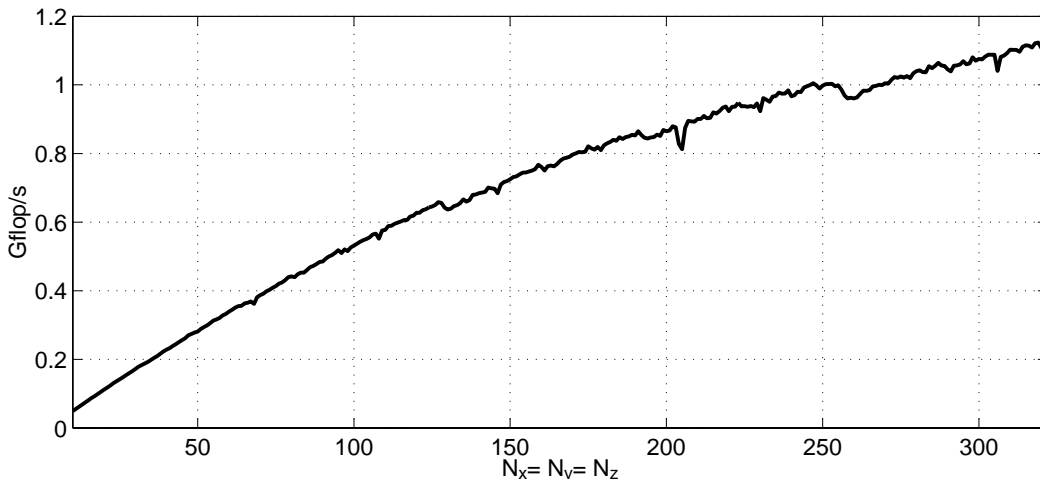


Figure 6. Yee_bench performance on *selma*.

4.4 Linux PC

Figure 7 displays the results for the three different node types used in the SBC PC-cluster at PDC. This cluster is run by PDC for the Stockholm Bioinformatics Center (SBC).

The compiler used was the Intel Fortran compiler (`ifc`), version 7.0 Beta Build 20020908Z. The optimization options used are listed in Table 6, which also lists the `stream2` DAXPY results. The `Yee_bench` results in Table 6 are the mean value of the performance for problem sizes larger than 128 Mbyte ($N > 141$).

We see that the `Yee_bench` performance is between P_l and P_h for all three cases.

Table 6. `stream2` DAXPY and `Yee_bench` results on the SBC cluster.

nickname	R	U	V
CPU type	PIII, 866 MHz	Athlon, 900 MHz	Athlon XP, 1.4 GHz
Compiler options	-O3 -xK	-O3	-O3
<code>stream2</code>	470	399	1040
P_l (Gflop/s)	0.081	0.069	0.18
P_h (Gflop/s)	0.12	0.10	0.26
<code>Yee_bench</code> (Gflop/s)	0.10	0.073	0.21
Peak Gflop/s	1.73	1.8	2.8
<code>Yee_bench</code> rel. perf.	5.8%	4.1%	7.5%

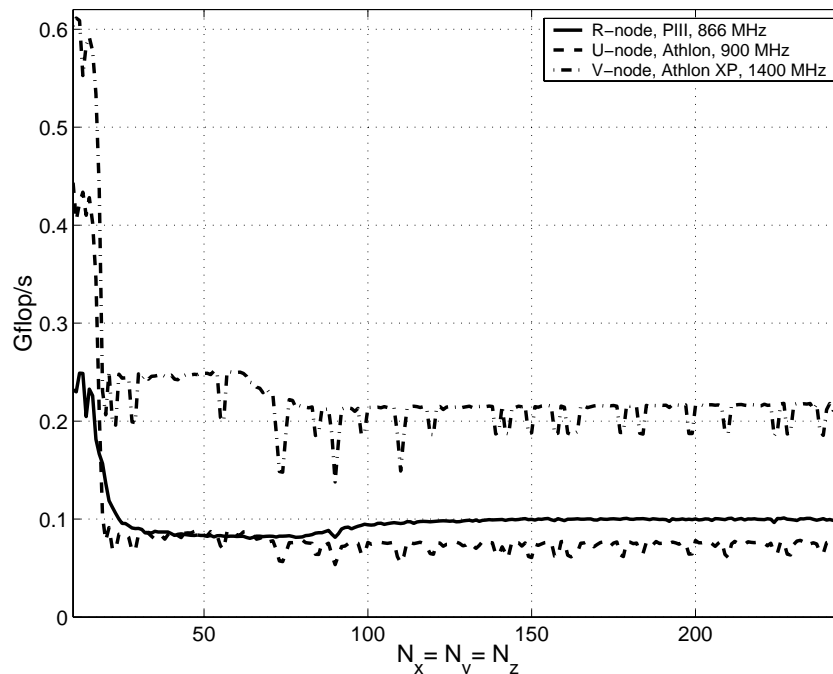


Figure 7. `Yee_bench` performance on nodes in the SBC cluster.

4.5 Itanium 2

Figure 8 shows the single stream result for an HP rx2600 server with two Itanium2, 900 MHz CPUs. The L3 cache size is 1.5 Mbytes. The theoretical peak performance is 3.6 Gflop/s.

The compiler `efc` Version 6.0, Build 20020320 has been used. The optimization option was `-O3`.

The `stream2` DAXPY result for large problems was approximately 5 000 million bytes per second. This gives $P_h = 1.25$ Gflop/s and $P_l = 0.87$ Gflop/s. The average `Yee_bench` performance for $N > 281$ is 0.92 Gflop/s, *i.e.*, 26% of the theoretical peak performance. Again, we see that the performance of `Yee_bench` is between P_l and P_h .

The rx2600 has 4 Gbyte 266 MHz DDR SDRAM memory. The memory bandwidth is 8.6 Gbyte/s and the system bus bandwidth is 6.4 Gbytes/s.

The severe drop in performance for $N = 437$ (3.74 Gbytes) is caused by swapping when there is no more memory available. This effect occurs on all computers. We also see a drop in performance at $N = 28$ (1.06 Mbyte). This is caused by the problem no longer fitting into the L2 cache.

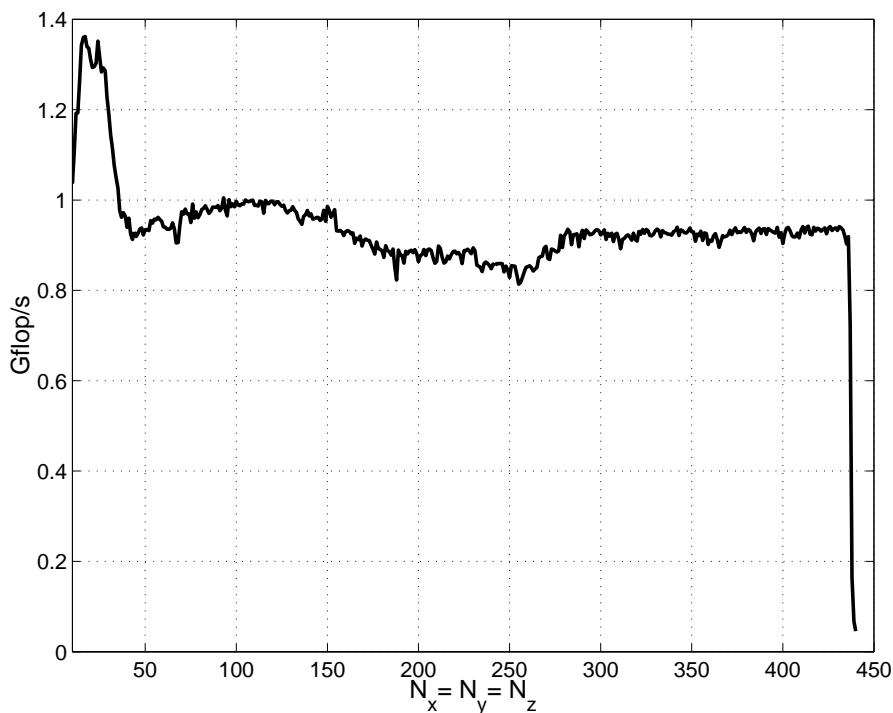


Figure 8. Performance on a 900 MHz Itanium2 CPU (HP rx2600).

Figure 9 shows the multiple stream results. We see that except for problem sizes that fits into cache we get no benefit by using the second CPU.

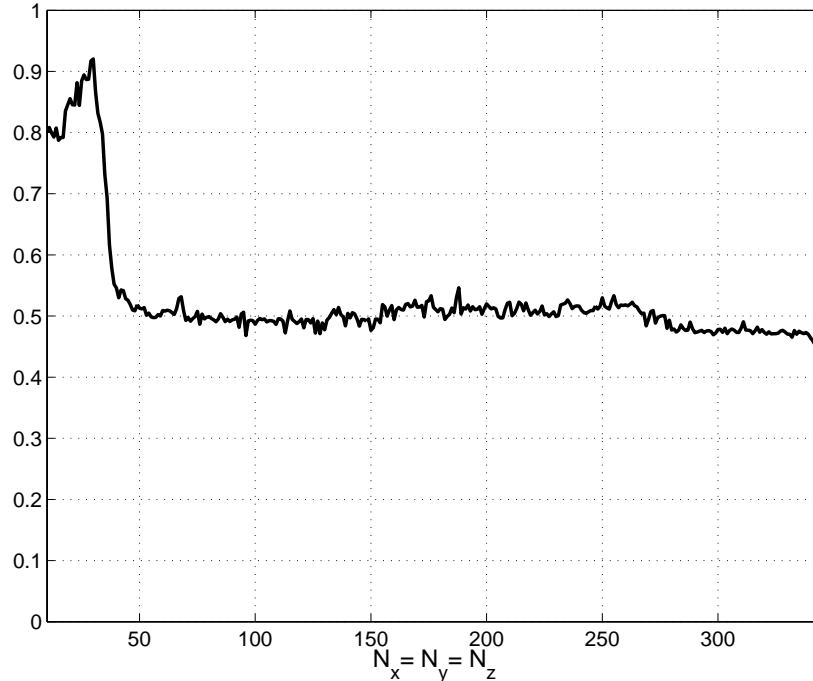


Figure 9. Performance decrease on a dual rx2600 server when two processes are running.

5 Conclusions and future work

With a few exceptions, *selma* (Fujitsu VX), *boye* (SGI), and IBM pwr2, we see that the *stream2* DAXPY results can be used to predict limits for the performance of *Yee_bench* using (9) and (10). This fact strongly supports our claim that *Yee_bench* is a memory bound code. On *boye* and the IBM pwr2, *Yee_bench* performs slightly below the prediction P_l from (9).

The best relative performance of *Yee_bench* occurs on vector computers. On *selma* we get 50% of the peak performance for large cubic problem sizes. On the other computers, *Yee_bench* achieves between 4% and 29% of the peak performance. Formulas (9) and (10) cannot be used to predict the *Yee_bench* performance on vector computers.

Possible future updates of *Yee_bench* include switching from point source excitation to using an initial field. This will insure that the entire domain is excited and remove excitation computations during timestepping.

A Yee_bench skeleton code

```

do nx=n_start,n_stop
  ny = nx ; nz = nx
  <Alloc.: H[xyz](1:nx,1:ny,1:nz);E[xyz](1:nx+1,1:ny+1,1:nz+1)>
  do iii=1,number_of_runs
    Ex=0.0 ; Ey=0.0 ; Ez=0.0 ; Hx=0.0 ; Hy=0.0 ; Hz=0.0
    do ts=1,nts ! timestepping loop
      if (ts==2) then
        <Start (wall-time) clock>
      end if
      !! H-updates are done in a separate SUBROUTINE (updateH)
      Hx(1:nx,1:ny,1:nz) = Hx(1:nx,1:ny,1:nz) +
        ( (Ey(1:nx,1:ny,2:nz+1)-Ey(1:nx,1:ny,1:nz))*Cbdx +&
          (Ez(1:nx,1:ny,1:nz)-Ez(1:nx,2:ny+1,1:nz))*Cbdy )
      Hy(1:nx,1:ny,1:nz) = Hy(1:nx,1:ny,1:nz) +
        ( (Ez(2:nx+1,1:ny,1:nz)-Ez(1:nx,1:ny,1:nz))*Cbdx +&
          (Ex(1:nx,1:ny,1:nz)-Ex(1:nx,1:ny,2:nz+1))*Cbdx )
      Hz(1:nx,1:ny,1:nz) = Hz(1:nx,1:ny,1:nz) +
        ( (Ex(1:nx,2:ny+1,1:nz)-Ex(1:nx,1:ny,1:nz))*Cbdy +&
          (Ey(1:nx,1:ny,1:nz)-Ey(2:nx+1,1:ny,1:nz))*Cbdx )
      !! E-updates are done in a separate SUBROUTINE (updateE)
      Ex(1:nx,2:ny,2:nz) = Ex(1:nx,2:ny,2:nz) +
        ( (Hz(1:nx,2:ny,2:nz)-Hz(1:nx,1:ny-1,2:nz))*Dbdy +&
          (Hy(1:nx,2:ny,1:nz-1)-Hy(1:nx,2:ny,2:nz))*Dbdz )
      Ey(2:nx,1:ny,2:nz) = Ey(2:nx,1:ny,2:nz) +
        ( (Hx(2:nx,1:ny,2:nz)-Hx(2:nx,1:ny,1:nz-1))*Dbdz +&
          (Hz(1:nx-1,1:ny,2:nz)-Hz(2:nx,1:ny,2:nz))*Dbdx )
      Ez(2:nx,2:ny,1:nz) = Ez(2:nx,2:ny,1:nz) +
        ( (Hy(2:nx,2:ny,1:nz)-Hy(1:nx-1,2:ny,1:nz))*Dbdx +&
          (Hx(2:nx,1:ny-1,1:nz)-Hx(2:nx,2:ny,1:nz))*Dbdy )
      !! A point source in the center.
      Ez(nx/2,ny/2,nz/2) = Ez(nx/2,ny/2,nz/2) + f(ts*dt)
      !! PEC outer bc implicitly enforced by E=0 in init.
    end do
    <Stop clock>
  end do
  <Compute best performance for this problem size>
  <Deallocate fields>
  if (shortest_time>20).and.(nts>10)) then
    nts = nts/2
  end if
end do

```

B Fused do-loops

Three different ways to code the magnetic field updates:

```

if (calcmnet==3) then ! Array syntax
  Hx(1:nx,1:ny,1:nz) = Hx(1:nx,1:ny,1:nz) +
    ( (Ey(1:nx,1:ny,2:nz+1)-Ey(1:nx,1:ny,1:nz))*Cbdz +
      (Ez(1:nx,1:ny,1:nz)-Ez(1:nx,2:ny+1,1:nz))*Cbdy )
  Hy(1:nx,1:ny,1:nz) = Hy(1:nx,1:ny,1:nz) +
    ( (Ez(2:nx+1,1:ny,1:nz)-Ez(1:nx,1:ny,1:nz) ) *Cbdx +
      (Ex(1:nx,1:ny,1:nz)-Ex(1:nx,1:ny,2:nz+1))*Cbdz )
  Hz(1:nx,1:ny,1:nz) = Hz(1:nx,1:ny,1:nz) +
    ( (Ex(1:nx,2:ny+1,1:nz)-Ex(1:nx,1:ny,1:nz))*Cbdy +
      (Ey(1:nx,1:ny,1:nz)-Ey(2:nx+1,1:ny,1:nz))*Cbdx )

else if (calcmnet==4) then ! three separate loops
  do k=1,nz
    do j=1,ny
      do i=1,nx
        Hx(i,j,k) = Hx(i,j,k) +
          ( (Ey(i,j,k+1)-Ey(i,j,k))*Cbdz +
            (Ez(i,j,k)-Ez(i,j+1,k))*Cbdy )
      end do
    end do
  end do
  <and similarly for Hy and Hz>

else if (calcmnet==5) then ! fused loops
  do k=1,nz
    do j=1,ny
      do i=1,nx
        Hx(i,j,k) = Hx(i,j,k) +
          ( (Ey(i,j,k+1)-Ey(i,j,k))*Cbdz +
            (Ez(i,j,k)-Ez(i,j+1,k))*Cbdy )
        Hy(i,j,k) = Hy(i,j,k) +
          ( (Ez(i+1,j,k)-Ez(i,j,k) ) *Cbdx +
            (Ex(i,j,k)-Ex(i,j,k+1))*Cbdz )
        Hz(i,j,k) = Hz(i,j,k) +
          ( (Ex(i,j+1,k)-Ex(i,j,k))*Cbdy +
            (Ey(i,j,k)-Ey(i+1,j,k))*Cbdx )
      end do
    end do
  end do

end if

```

Usually, `calcmct==3` and `calcmct==4` give identical results. Most compilers seem to replace the array syntax with three separate loops.

Note that in each cell/iteration we seem to need 18 floating-point operations (6 additions, 6 subtractions and 6 multiplications), 3 stores and 15 loads. However, some of the loads can be avoided by storing values in registers. In fact, with `calcmct==4` (or 3) we should only need 13 loads, and the with `calcmct==5` we should only need 10 loads. This reasoning (see next paragraph) depends on the assumption that the compiler realizes this, which usually is the case. It has, for instance, been verified on the IBM pwr3 nodes by using the tool `hpmcount`.

In the fused loops, `Ex(i,j,k)`, `Ey(i,j,k)` and `Ez(i,j,k)` occur twice, which saves us 3 loads. The occurrence of `Ey(i+1,j,k)` and `Ez(i+1,j,k)` saves us 2 loads. These two values were used in the previous iteration. These two saves occur also for `calcmct==4` (or 3).

The constants `Cbd[xyz]` are assumed to be stored in registers.

The update of the electromagnetic field components is very similar. The difference is that the tangential electric field components at the outer boundary is calculated by the outer boundary condition. (In our case, they are kept zero, since we assume PEC at the outer boundary.) In array syntax, the updates are:

```

Ex(1:nx,2:ny,2:nz) = Ex(1:nx,2:ny,2:nz) +           &
  ( (Hz(1:nx,2:ny,2:nz) - Hz(1:nx,1:ny-1,2:nz)) * Dbdy +   &
    (Hy(1:nx,2:ny,1:nz-1) - Hy(1:nx,2:ny,2:nz)) * Dbdz )
Ey(2:nx,1:ny,2:nz) = Ey(2:nx,1:ny,2:nz) +           &
  ( (Hx(2:nx,1:ny,2:nz) - Hx(2:nx,1:ny,1:nz-1)) * Dbdz +   &
    (Hz(1:nx-1,1:ny,2:nz) - Hz(2:nx,1:ny,2:nz)) * Dbdx )
Ez(2:nx,2:ny,1:nz) = Ez(2:nx,2:ny,1:nz) +           &
  ( (Hy(2:nx,2:ny,1:nz) - Hy(1:nx-1,2:ny,1:nz)) * Dbdx +   &
    (Hx(2:nx,1:ny-1,1:nz) - Hx(2:nx,2:ny,1:nz)) * Dbdy )

```

The number of floating point operations per time step is approximately $36N_x N_y N_z = 36N^3$. More exactly it is $18N^3 + 18(N-1)^3 + 18(N-1)^2$. `Yee_bench` uses the exact number when calculating the performance.

C Fortran versions

There are five alternative versions of the Fortran 90 code. They differ in how the electromagnetic field is stored. In all versions, we use allocatable arrays (or pointers). The versions are called:

- **ADT:** A user defined (Abstract) Data Type is used to store the electromagnetic field arrays:

```
type adt_type
real(kind=rfp), dimension(:,:,:), pointer :: Ex,Ey,Ez,Hx,Hy,Hz
end type adt_type
```

Fortran 90 does not allow allocatable arrays in user defined types. Hence, we must construct the electromagnetic field array using pointers.

- **EandH:** Two four-dimensional arrays are used, one for the electric field and one for the magnetic field.
- **EH:** One four-dimensional array, `EH(6,1:nx+1,1:ny+1,1:nz+1)`, is used.
- **SepMod:** The same allocations as in the original code, but the six electromagnetic field arrays are defined in a separate module instead of the main program.
- **Slim:** Similar to the allocations in the original code. Each of the six arrays have different sizes to make the memory usage as small as possible. The sizes are:

```
Hx(2:nx ,1:ny ,1:nz )
Hy(1:nx ,2:ny ,1:nz )
Hz(1:nx ,1:ny ,2:nz )
Ex(1:nx ,1:ny+1,1:nz+1)
Ey(1:nx+1,1:ny ,1:nz+1)
Ez(1:nx+1,1:ny+1,1:nz )
```

Note that this affects the magnetic field update described in Appendix B. We have here eliminated the (normal) magnetic components on (the lower sides of) the boundary, since we do not need them. Hence, we cannot update them. (In the original code, these components are updated, but the result is never used since the (tangential) electric field components at the boundary are set to zero.)

Results for the different versions are given in Table 7. On the Pentium III (Coppermine) processor on the computer `roxette` we used the `pgf90` compiler, version 4.0-1, with the option `-fast -Mvect=sse`. On the IBM CPUs we used the compilers and compiler options listed in Table 4.

Studying Table 7, we see that in all cases there is a performance penalty in using user-defined datatypes.

Putting all fields in one array also gives low performance. This should increase the locality of references. However, the compilers have difficulties with handling this case. They are probably not able to determine that two references like $\text{EH}(i, j, k, 4)$ and $\text{EH}(i, j, k+1, 2)$ do not overlap with one another.

There is also a small performance loss for the `Slim` version. This might be due to a higher cache miss rate. However, we have only used one problem size. This is not enough to draw any conclusions on whether this implementation is less efficient than the original code.

With one exception, `EandH` on `pwr3`, the original code is the fastest implementation. For the exception, the difference is too small to draw any conclusions.

Table 7. Performance in Mflop/s for the different Fortran90 versions for $N = 100$. The other parameters in the input file was set to $N_t = 200$ `calcmets=5`, and `number_of_runs=3`.

	PIII, 866 MHz	pwr2, 160 MHz	pwr3, 222 MHz
Peak performance	866	640	888
Original code	108	179	230
ADT	29	141	112
EandH	76	176	236
EH	67	127	159
SepMod	107	177	81
Slim	107	166	205

Using the Intel compiler (version 6.0, Build 020312Z) on the PIII (`roxette`) and using the original code gave a performance of 107 Mflop/s. We used the optimization options `-O3 -xK`. The `stream2` DAXPY result using `pgf90` was 630 million bytes per second. This gives $P_h = 0.16$ Gflop/s and $P_l = 0.11$ Gflop/s.

References

- [1] U. Andersson. *Time-Domain Methods for the Maxwell Equations*. PhD thesis, Kungl Tekniska Högskolan, Stockholm, Sweden, 2001. Available at <http://media.lib.kth.se:8080/dissfull.asp>.
- [2] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly, second edition, 1998.
- [3] K. S. Kunz and R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*. CRC Press, Boca Raton, FL, 1993.
- [4] A. Taflove, editor. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, 1998.
- [5] A. Taflove. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Boston, MA, second edition, 2000.
- [6] K. S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans. Antennas Propagat.*, 14(3):302–307, March 1966.