

# A Technical Note About Fortran 90 Optimization on RISC Processors

Carlo Nardone

Geophysics Area  
CRS4

Via Nazario Sauro, 10  
I-09123 Cagliari, Italy

E-mail: [cmn@crs4.it](mailto:cmn@crs4.it)

July 27, 1998



# 1 Background

This short note has been sparked by some misconceptions about performance optimization of Fortran scientific codes on RISC machines here at CRS4. In what follows I will take for granted that Fortran scientific codes make heavy use of operations on large arrays. This fact opens the way to vector and parallel microprogramming strategies which are fully exploited by today RISC processors.

Common wisdom maintains that pure Fortran 77 (F77) codes are compiled into faster executables than corresponding Fortran 90 (F90) [5] ones in general. While it is true that many new features introduced by F90 are more difficult for the compiler to optimize, some other features may even help the optimizer, for example array syntax statements. Operations on POINTERS to arrays and *assumed shape* arrays are examples of the former case. Notice that the ubiquitous use of pointers in C is the main reason of poorer performances with respect to F77.

When suitable programming practices are followed, however, no practical performance degradation is observed for most native F90 compilers on the RISC workstations available at CRS4.

The main causes of possible performance degradation involving F90 arrays are the following:

1. dynamical allocation;
2. non-contiguous memory (e.g. operations with `A(1:10:3)`);
3. aliasing, i.e. overlap of memory extent.

Point 1. affects the place where the array is placed in RAM. The user memory is organized according to the operating system. In particular, two subareas can be distinguished according to the allocation policy: the *stack* area, handled as a LIFO queue, and the *heap* area, which is not a queue at all. Automatic arrays and compiler-generated temporaries are usually placed in the stack while allocatable and pointer objects are usually placed in the heap. Clearly it is implementation dependent and affects user memory fragmentation, for example, but not optimization *per se*.

Point 2. A stride different from 1 implies more address operations and possible inefficiencies in cache usage, therefore less efficiency.

Point 3. It refers to the fact that different variable names refer to the same memory location, in practice different arrays may overlap. This possibility prevents many optimizations.

It has been argued that a clever enough compiler can counteract point 2. and 3. effectively. For example, in the case of the SGI f90 compiler (v7.2), a member of the compiler development group states that [6]

the latest SGI compiler can get the same performance on an ALLOCATABLE, assumed-shape, explicit shape and automatic array, provided that sufficient information is available. (For example, write array syntax with explicit upper and lower bounds (`A(1:N)`) instead of `A(:)`).

This is actually confirmed by the simple benchmark in the next section.

Other compilers may need a special option to be guaranteed by the programmer that no aliasing occurs (point 3.). See Table 1 for the machines at CRS4 (see also next section).

vendor	\$ARCH	f90 option
DEC	axp	-assume nodummy_aliases
IBM	r6k	-qalias=noaryovrlp:std
SGI	sgi4d	-OPT:alias=restrict
HP	hppa	?
Sun	sun4	?

Table 1: Compiler options to express no aliasing

## 2 A simple benchmark

We choose a simple algebraic operation with arrays for benchmarking F77- vs F90-style, standardized as BLAS [1] level 1 operation SAXPY (for single precision) and DAXPY (for double precision). It is simply  $y = y + \alpha x$  where  $x$  and  $y$  are vectors (1D-arrays) with  $n$  elements each and  $\alpha$  is a scalar. The completion of one SAXPY amounts to  $2n$  floating point operations (Flop) vs 3 memory operations. Most Unix workstations and technical servers come with machine-optimized BLAS libraries, at least level 1. Therefore the benchmark includes a call to such library as well.

The Mflop rate (Mflop/s i.e.  $10^6$  Mflop per second) obtained depends heavily from the number of elements in the arrays. The optimum rate is invariably a fraction of the infamous peak Mflop/s rate, which for RISC processors is a integer multiple of the clock speed in MHz. The peak processor speed has the only meaning of “speed of light” limit, that is an unattainable upper bound. The peak rate for the machines tested is reported in Table 2.

\$ARCH	machine	$\mu$ proc	peak Mflop/s	cache levels
axp	?	Alpha 21164?	?	3
r6k	sp101	IBM Power2	266	1
sgi4d	coral	MIPS R10000	390 (1CPU)	2
hppa	titanic	HP PA8000	720 ? (1CPU)	1
sun4	paperino	UltraSPARC	?	2

Table 2: Machines,  $\mu$ processors and peak rate

The dependence from the number of elements is striking in Figure 1 (note that the horizontal axis is logarithmic). It can be qualitatively described as follows. When the number of elements is small, hardware pipelining is inefficient and therefore the computational speed starts small. With larger arrays the Mflop rate increases, reaching a plateau for about  $n = 10^3$ . When the arrays do not fit into the cache, a sudden drop in the Mflop rate is observed. The drop can be splitted according to the number of cache levels present in the architecture; two levels are evident in the SGI case. Asymptotically, the Mflop rate is related to the bandwidth between the CPU and the main memory.

Considering the main purpose of this Note, i.e. the evaluation of F90 vs F77, Figure 1 shows a remarkable difference between systems (processor + memory hierarchy + compiler + user). In particular, while the SGI claim of equivalence between F77 and F90 flavours of the same calculation is fulfilled (somewhat less satisfactorily for assumed shape routines), and the same is true for the IBM compiler (with no aliasing option, see later), this is definitely not true for the HP compiler. Even the BLAS implementation is unsatisfactory for the latter system, while, somewhat surprisingly, only for the IBM compiler it has some advantage over the straightforward implementation used for this test (see the appendix).

Single and double precision affects performances in several ways. First of all, every processor considered is equipped by full 64 bit FPU (floating point units). In some cases, single precision is obtained via microcode using such 64 bit hardware, with some performance cost. But doubling the precision slows down the access to memory and fills the cache more easily. Therefore, the single vs double precision comparison gives different results on different systems, as you can judge by yourself comparing 2 with Figure 1.

The aliasing effect is shown by the IBM SP2 thin node data 3. The only difference between alias and no-alias timing data is due to the compiler option described in the previous paragraph. Only the routines affected by this option have been plotted (i.e. `myf90saxpy` (`myf90` in the legend) and `myf90saxpy_shape` (`asf90`), both single and double precision).

Finally, as a warning to quick jumps to conclusions such as “system A is faster than system B”, even for extremely simple algebraic kernels such as BLAS-1, I tested the SDOT routine (dot product of two vectors in single precision) on the same platforms considered before. Figure 4 shows that, in this case, the SGI platform is consistently faster than HP.

### 3 Acknowledgements

I would like to thank Luca Paglieri for useful suggestions and for his efforts dedicated to the spread of high performance computing culture here at CRS4.

## References

- [1] BLAS (Basic Linear Algebra Subprograms) Frequently Asked Questions (FAQ), URL: <http://www.netlib.org/blas/faq.html>
- [2] K. Dowd, *High Performance Computing*, O'Reilly & Ass., 1993.
- [3] comp-fortran-90 mailing list (available by sending request to [comp-fortran-90-request@mailbase.ac.uk](mailto:comp-fortran-90-request@mailbase.ac.uk) or at the URL: <http://www.mailbase.ac.uk/lists-a-e/comp-fortran-90/>)
- [4] *Optimization and Tuning Guide for Fortran, C, and C++*, IBM AIX Version 3.2 for RISC System/6000 Manual SC09-1705-00, 1993.
- [5] M. Metcalf, J. Reid, *Fortran 90/95 Explained*, Oxford Univ. Press, 1995.
- [6] Richard Shapiro ([rshapiro@boston.sgi.com](mailto:rshapiro@boston.sgi.com)), comp-fortran-90 mailing list message, Thu, 24 Jul 1997.

## A Code listings

The source code used for benchmarking is available electronically in the directory `/u/cmn/Languages/F90/testBlas`.

### A.1 myf77saxpy.f

```
subroutine myf77saxpy(n,alpha,x,y)
  implicit none
  integer n, i
  real alpha, x(n), y(n)
  do i=1,n
    y(i) = y(i) + alpha * x(i)
  enddo
  return
end
```

### A.2 myf90saxpy.f

```
subroutine myf90saxpy(n,alpha,x,y)
  implicit none
  integer,          intent(in)    :: n
  real,            intent(in)     :: alpha
  real, dimension(n), intent(in)  :: x
  real, dimension(n), intent(inout) :: y
  y = y + alpha * x
end subroutine myf90saxpy
```

### A.3 myf90saxpy\_shape.f

```
subroutine myf90saxpy_shape(alpha,x,y)
  implicit none
  real,          intent(in)    :: alpha
  real, dimension(:), intent(in)  :: x
  real, dimension(:), intent(inout) :: y
  y = y + alpha * x
end subroutine myf90saxpy_shape
```

### A.4 myf77sdot.f

```
function myf77sdot(n,x,y)
  implicit none
  integer n, i
  real x(n), y(n), myf77sdot
```

```

myf77sdot = 0.0
do i=1,n
  myf77sdot = myf77sdot + x(i) * y(i)
enddo
return
end

```

## A.5 myf90sdot.f

```

real function myf90sdot(n,x,y)
implicit none
integer,          intent(in)    :: n
real, dimension(n), intent(in)  :: x
real, dimension(n), intent(inout) :: y
myf90sdot = sum(x*y)
end function myf90sdot

```

## A.6 myf90sdot\_shape.f

```

real function myf90sdot_shape(x,y)
implicit none
real, dimension(:), intent(in)    :: x
real, dimension(:), intent(inout) :: y
myf90sdot_shape = sum(x*y)
end function myf90sdot_shape

```

## A.7 mainf90alloc.f

```

program mainf90alloc

use SystemTimer
implicit none

include 'blas.inc'
include 'myf90saxpy_shape.inc'
include 'myf90daxpy_shape.inc'
include 'myf90sdot_shape.inc'
include 'myf90ddot_shape.inc'

real, dimension(:), allocatable :: x, y
real*8, dimension(:), allocatable :: u, v

```

[...]

```

allocate(x(nmax))
allocate(y(nmax))

x = 1.0
y = 2.0

call clearClock(1,'blas_saxpy')
call startClock(1)
do i=1,niter
  call saxpy(nmax,alpha,x,1,y,1)
enddo
call stopClock (1)
call printClock(1)

call clearClock(2,'myf77saxpy')
call startClock(2)
do i=1,niter
  call myf77saxpy(nmax,alpha,x,y)
enddo
call stopClock (2)
call printClock(2)

[...]

mflop = (2*nmax*1.e-6*niter)
write(*,100) 'blas_saxpy rate= ',mflop/timeAvgClock(1),' n= ',nmax

[...]

end program mainf90alloc

```

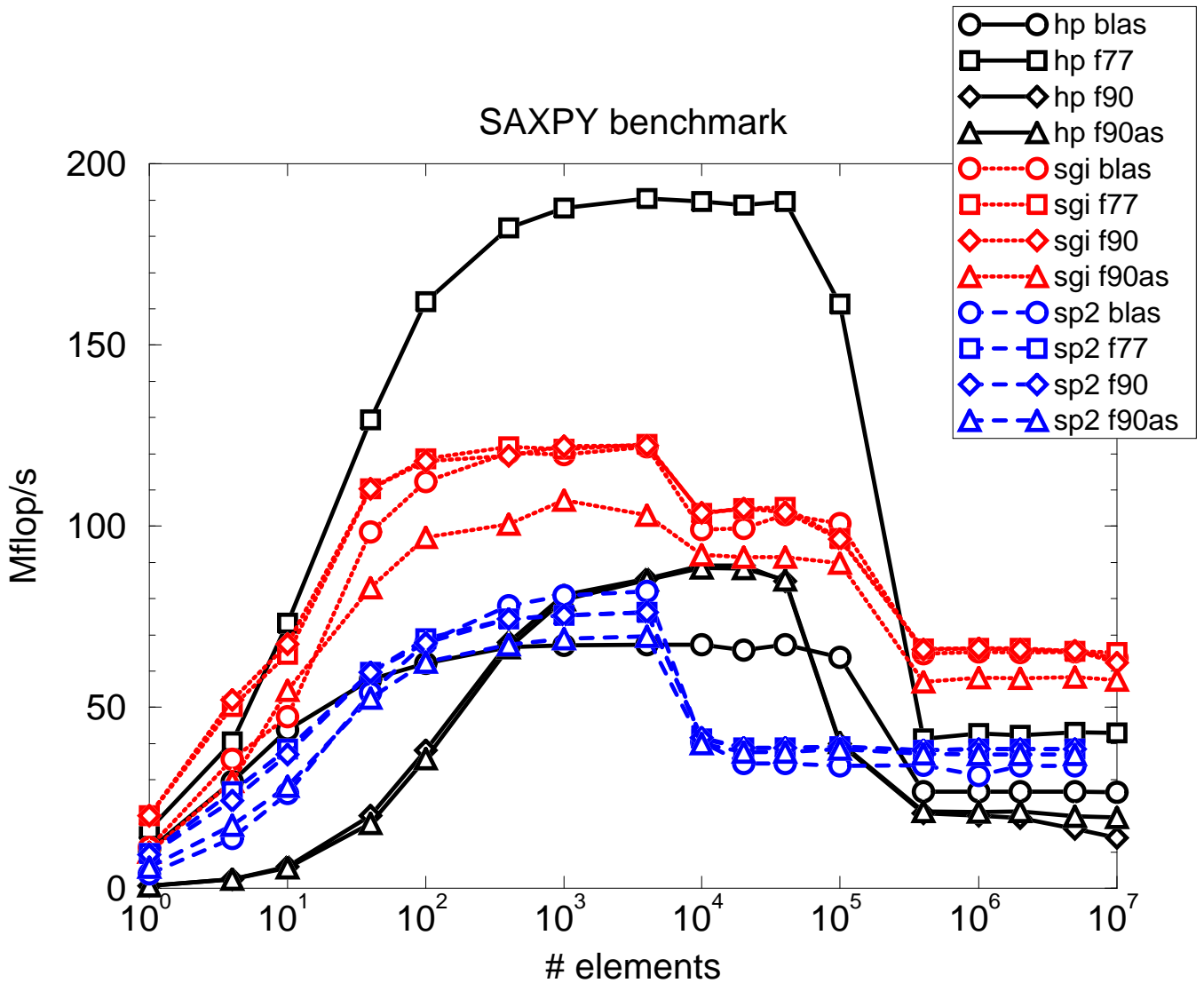


Figure 1: SAXPY benchmark results. The Mflop/s sustained rate is plotted as a function of the number of elements in the vector. The systems considered are an HP, an SGI workstation and a SP2 thin node (see text). The keywords in the legend stand respectively for: blas=optimized, native library call - f77=f77-style routine, parameters as in blas call - f90=f90-style routine, parameters as in blas call - f90as=f90-style routine with assumed shape arrays as parameters.

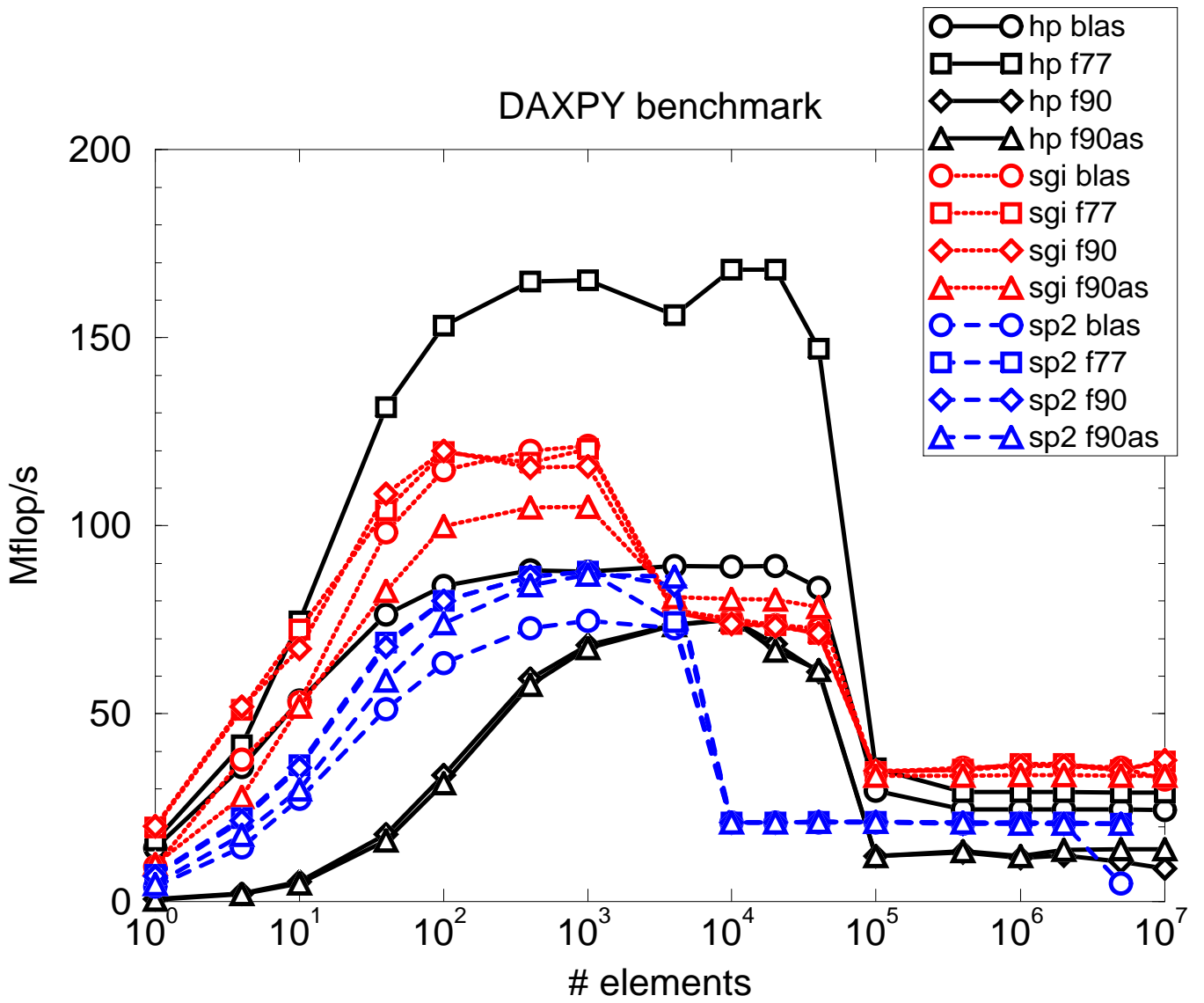


Figure 2: DAXPY benchmark results. Legend as in Figure 1.

# IBM SP2 thin

alias vs noalias

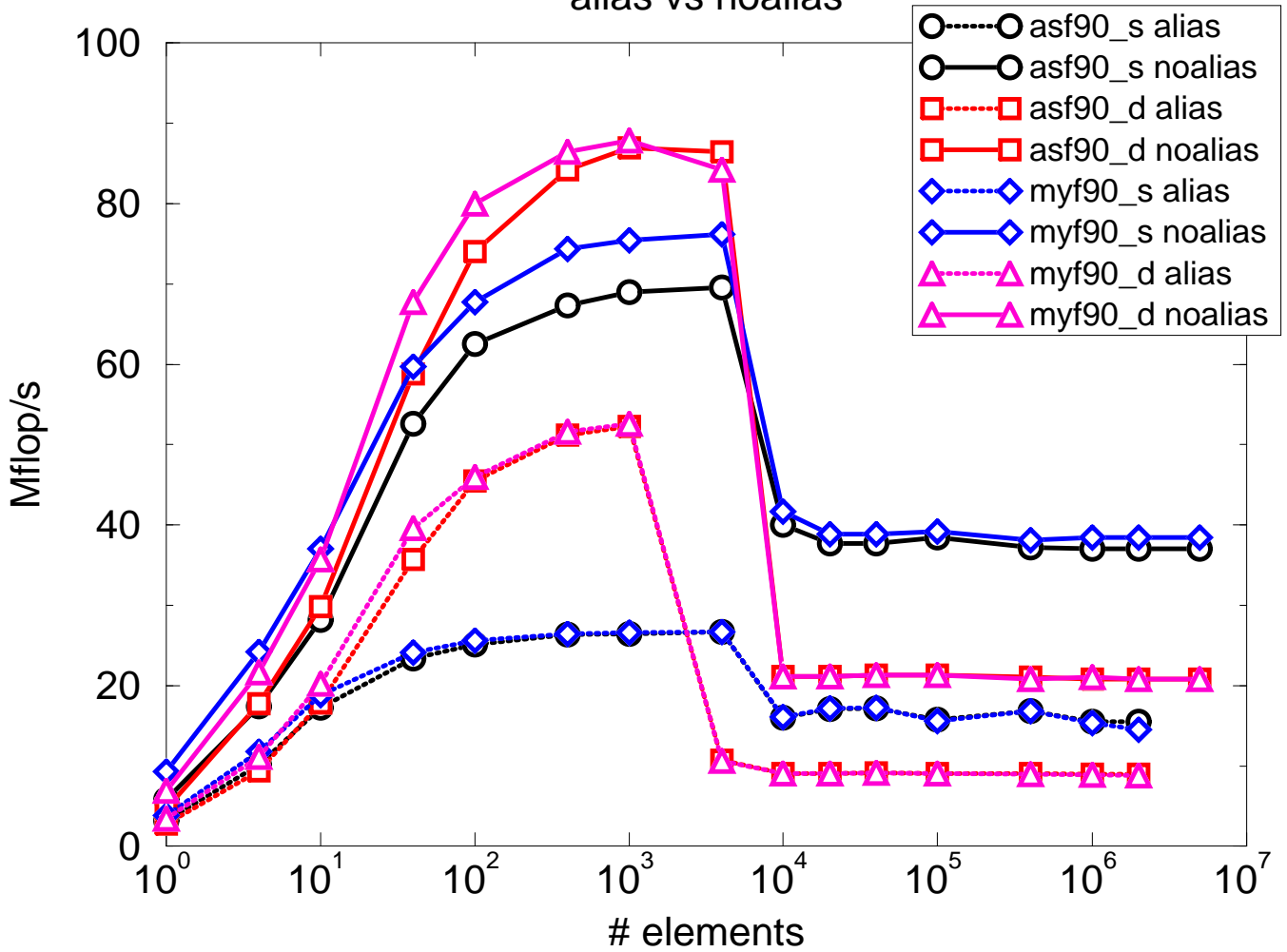


Figure 3: Aliasing effect. The IBM SP2 thin node only has been considered. Results from compilation with noalias enforcing options are consistently faster than default, with possible aliasing between arrays. The keywords in the legend stand respectively for: asf90\_s=assumed shape arrays, single precision - asf90\_d=assumed shape arrays, double precision - myf90\_s=automatic arrays, single precision - myf90\_d=automatic arrays, double precision.

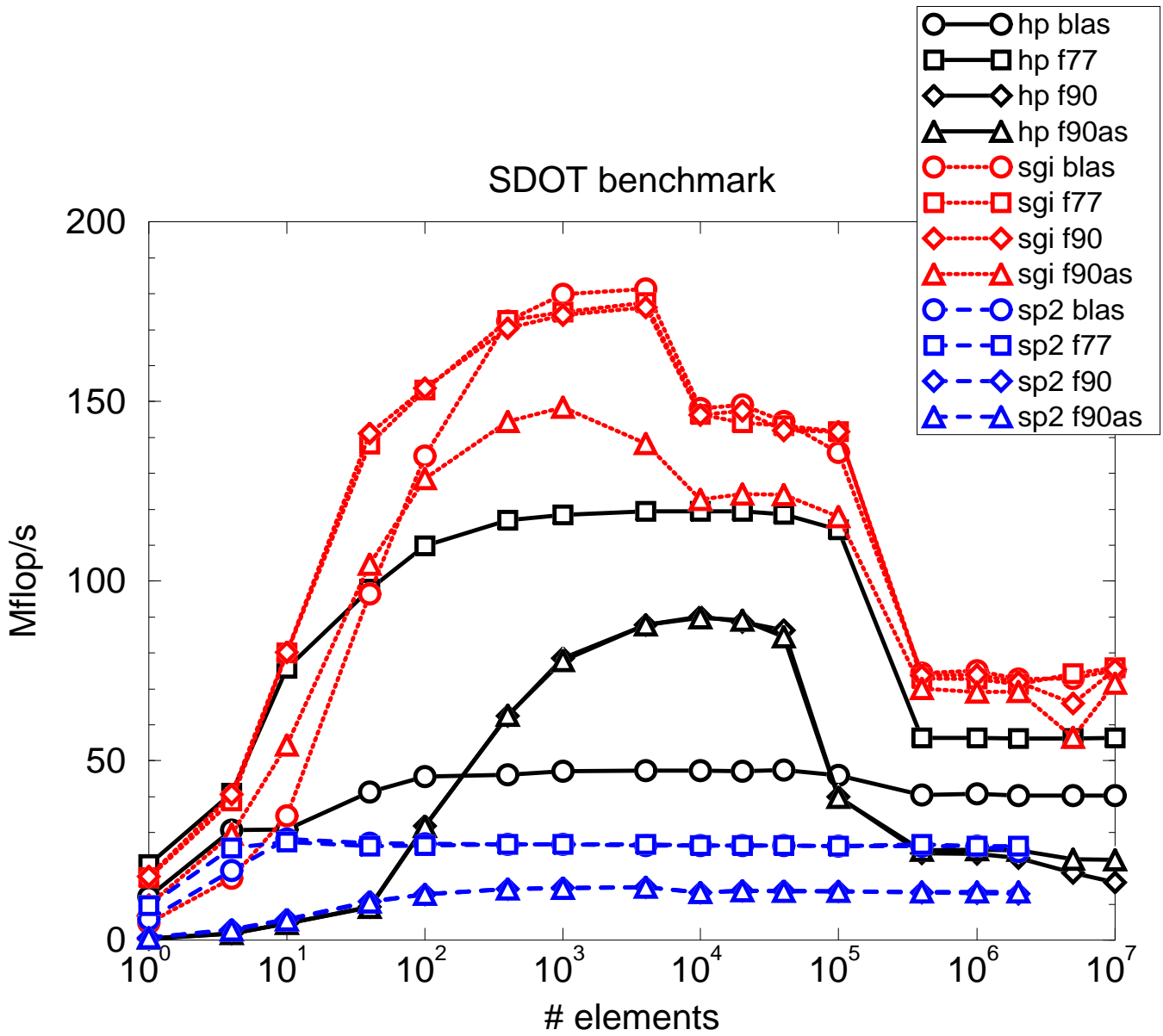


Figure 4: SDOT benchmark results. Legend as in Figure 1.