Preliminary benchmark results on an Itanium 2 reference platform

Aad J. van der Steen
High Performance Computing Group
Utrecht University
P.O. Box 80195
3508 TD Utrecht
The Netherlands
steen@phys.uu.nl
www.phys.uu.nl/~steen

Abstract

In this preliminary report we give an overview of the benchmark results on an Itanium 2 reference platform of 4 quad SMP nodes connected by Myrinet 2000. Single-CPU, OpenMP, and MPI tests with the EuroBen benchmark are reported. We compare the performance results with the same results obtained at an SGI Origin3800 with 500 MHz MIPS R14000 processors. In addition, in a later stage the same single-CPU and OpenMP were performed on an Intel Madison-based quad SMP server and an SGI Altix 3000 with Madison processors. Where relevant also these results are reported.

Contents

Table 1.1: *Hardware and software characteristics of the systems used.*

Hardware	**Intel**	**Intel**	**SGI**	**SGI**
Name	Itanium 2 cluster	Madison server	Altix 3000	Origin3800
No. of Nodes	4	1	—	—
No. of CPUs	16	4	64	512
CPU Type	1 GHz Itanium 2	1.26 GHz Madison	1.3 GHz Madison	500 MHz R14K
Theor. Peak	4.0 Gflop/s	5.0 Gflop/s	5.2 Gflop/s	1.0 Gflop/s
Memory	4/8 GB/node	5 GB/node	4 GB/brick	4 GB/brick
System bus BW	6 GB/s	6 GB/s	10.2 GB/s	3.2 GB/s
Network type	Myrinet 2000	—	Numalink 3/4	Numalink 3
Peak BW	240 MB/s	—	1.6/3.2 GB/s	1.6 GB/s
Software				
OS	RedHat Linux 7.2	RedHat Linux 7.2	RedHat Linux 7.2	IRIX 6.5
Fortran	Intel Fortran 7.0	Intel Fortran 7.1	Intel Fortran 7.1	MIPSPro f90 7.3
Compiler flags	-O3 (-openmp)	-O3 (-openmp)	-O3 (-openmp)	-O3 (-mp)
MPI	MPICH-GM 1.2.5	—	SGI MPI	SGI MPI

1 Introduction

This is a first impression of the performance on a cluster of 1 GHz Itanium 2 processors. The tests were performed on Intel's reference platform at their research centre in München, Germany. We used the EuroBen Benchmark Version 4.0 for single CPU and shared memory parallel (OpenMP) performance while the EuroBen-DM Benchmark Version 1.0 was used for distributed-memory (MPI) performance[10, 2].

To relate the performances found to the levels currently obtained on the TERAS system, a MIPS R14000-based Origin 3800 with 512 processors, we compare the results of both systems where relevant. The testing circumstances were as shown in Table 1.1: Note that in this table the nominal hardware characteristics are given. Effective speeds may be significant lower: e.g., the MPI bandwidth between two Origin processors is about 1/6th of the hardware bandwidth of 1.6 GB/s.

One has to keep in mind that the results presented in this report are provisional: no special optimisation is done, nor platform specific libraries were used to boost the performance. These tests simply reflect the performance of straightforwardly compiled code with whichever flaws or special features the respective compilers may contain. Of course the MIPSPro `f90` compiler is much more mature than the Intel compiler where the latter is developed for a processor architecture that is quite recent and where many code generation and optimisation techniques that are appropriate for RISC processors do not apply. Therefore, even more than in other situations, this report shows a single-point measurement that may be invalidated in a short time because of both hardware and software developments.

In a later stage we had a Madison-based quad SMP reference platform at our disposition. On this system we performed the same single-CPU and OpenMP tests as on the other platforms but not the MPI tests. The system was atypical in that the clock frequency was 1.26 GHz while the L3 cache was 6 MB. It turned out that the results of this system scaled in almost all cases exactly with the clock frequency for compute intensive programs and showed similar performance for data intensive programs because the frontside bus in the Itanium 2 and the Madison systems is identical. Where relevant, we also comment on these results.
In a few cases we were able to use an Altix 3000 system fitted with 1.3 MHz Madison processors for our MPI tests. This is useful because the network structure of the Origin 3800 and the Altix 3000 are very similar be it that within Compute bricks the faster Numalink 4 instead of the Numalink 3 is employed. The latter

network components are used in the Origin 3800 everywhere.

The structure of this report is as follows: in section 2 we present the results for the single CPU benchmark, EuroBen V4.0, configured as the sequential version. In section 3 the OpenMP results are discussed, again stemming from EuroBen V4.0 but configured for multi-processing. Section 4 contains the distributed memory MPI performances using the EuroBen-DM benchmark V1.0. The algorithms in the shared-memory and the distributed-memory benchmark are largely the same. This enables the assessment of the two parallelisation paradigms. Finally we draw some conclusions in section 5.

2 Single CPU results

For the single CPU performance we used both for the SGI and the Intel platforms `-O3` as optimisation level for the respective compilers. This is done in a configuration stage after which the same optimisation flags apply for all programs in the benchmark, unless compiler errors are met that only can be removed by recompiling specific parts of the code with different compiler flags. For the single CPU version of the EuroBen V4.0 codes this nowhere was necessary.

The EuroBen Benchmark contains parts, called modules, of increasing complexity. Module 1 contains programs that test the performance of basic operations in order to be able to draw conclusions about machine characteristics, whether they stem from a hardware or a software source. Module 2 contains basic algorithms that build upon the basic operations and therefore should expose whether one pays a performance penalty and if so, how much, for the increased complexity of the context in which the operations are embedded. Module 3 contains more complex algorithms, mostly combining some basic algorithms in some cases combined with significant I/O. For clear presentation we only discuss a subset of the programs in order to get a first idea of the machine capabilities, without delving too deep in all kinds of derived machine behaviour details.

2.1 Module 1

The single-CPU version of module 1 tests four items:

1. Program `mod1ac`: The speed of important "basic operations". Here basic means anything from a simple array copy or dyadic operations like a floating-point multiply to the evaluation of a second difference expression or a 9-order polynomial, 14 operations in all. Moreover, to assess the influence of (cache) memory bank conflicts and indirect addressing a subset of the 14 operations are performed with strides of 3 and 4 and using a randomly permuted index array.

2. Program `mod1d`: Detailed assessment of (cache) memory bank conflicts. Both for reading and writing memory.

3. Program `mod1e`: Accuracy of intrinsic mathematical functions. Although this strictly is not a performance issue, it has been shown that the implementation of the intrinsic function accuracy-wise leaves to be desired and even with new compiler release for the same platform can sometimes show unacceptable flaws in the accuracy. We therefore test the most important intrinsics in the relevant ranges where often different evaluation algorithms are used.

4. Program `mod1f`: The speed of the same intrinsic functions as a function of the vector length. From this also an estimate of the $n_{1/2}$ value is made (see [3] for this parameter).

2.1.1 `mod1ac`

Both the Itanium 2 and the MIPS processor should exhibit a speed of about $1/6^{\text{th}}$ of their respective peak speeds, ideally 666 and 166 Mflop/s. From Figure 2.1 it is clear that both systems come fairly close to this ideal performance. The highest observed speed for the Itanium 2 655 Mflop/s, 98% of its theoretical peak, the MIPS processor attains 164 Mflop/s giving exactly the same efficiency of 98% of the peak performance. For these simple dyadic operations both processors are able to execute the optimal code.

Various other effects can be noticed for this simple dyadic operation. For one, the $n_{1/2}$ i.e., the vector length for which half of the peak speed is attained, is rather high in comparison to that of most RISC processors. For the multiply operation it is about 85–90 where it is in the order of 5–10 on most RISC processors (it is

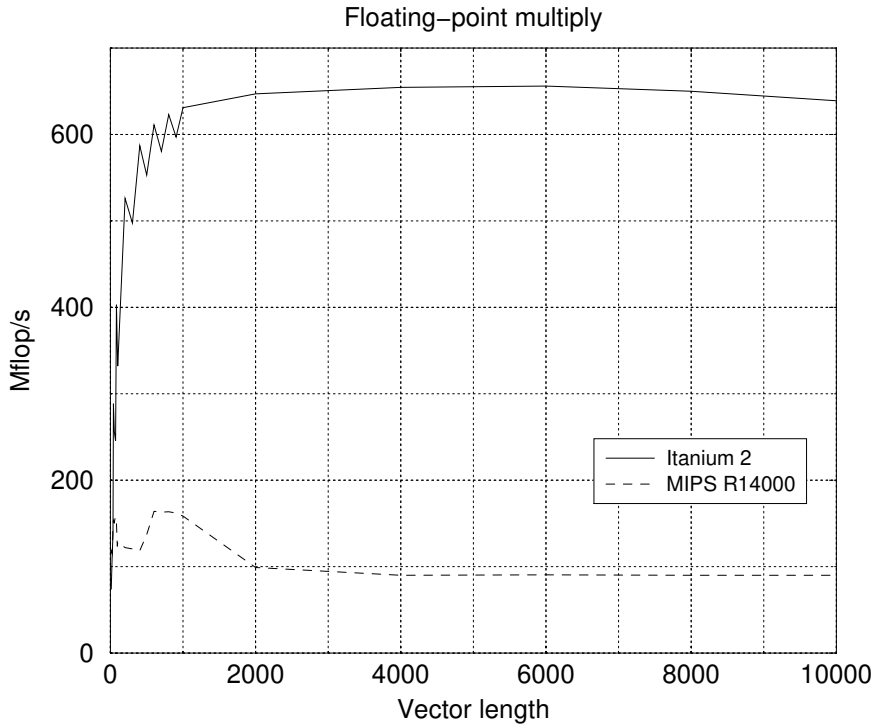

Figure 2.1: *Performance of the floating-point multiply as a function of the vector length*

7–8 on the MIPS R14000 processor). The Itanium's EPIC architecture behaves more like a vector processor because of its static instruction scheduling in this respect. This also shows from the sawtooth pattern of the performance for small vector lengths. There where the vector lengths are not exact multiples of the cache line length, e.g. at $n = 30$, 60, and 70, extra loads are required, giving a somewhat lower performance. Of course, for higher vector lengths this effect is not noticeable anymore.

For the vector lengths considered here (10–10,000) no performance degradation is observed for the Itanium 2 when the elements are accessed consecutively, while on the MIPS processor a maximum speed is observed around a vector length $n = 1000$ which drops off to a value which is roughly half the highest observed one due to primary cache misses. For larger strides, 3 or 4, also the Itanium processor shows similar behaviour where 2 vectors of about ≥ 8000 elements are involved, consistent with the size of the primary cache.

With the Itanium 2 there is a noticeable difference in speed when accessing the vector elements with a stride of 3 or or 4: with a stride of 3 there is a speed plateau of ≈ 320 Mflop/s in the length range of 300–2000, with a stride of 4 this plateau is at ≈ 275 Mflop/s. The most straightforward explanation is a cache memory bank conflict at a stride of four. In the MIPS processor there is no significant change in performance between stride-3 and stride-4 access and there is a slight speed degradation ($\approx 10\%$) with respect to consecutive, stride 1 access.

In Figure 2.2 the maximum performances, r_{max}, of all stride-1 kernels is shown for both systems. The figure shows that speeds for the Itanium processor generally are about 4 times higher, consistent with the number of floating-point functional units and the clock cycles of the respective processors. On the dotproduct with a computational intensity $f = 1$ (see [4] for this concept) one expects the speed to be 50% of the theoretical peak performance in view of the bandwidth from the L1 cache to the processor. This is indeed observed for the R14000 but it is 40% on the Itanium 2, showing that the generated code is not yet optimal. On the other hand the vector update operation $x = x + \alpha y$ with a computational intensity of $f = 2/3$ shows on both processors the theoretical efficiency of 33%. On the 2-D rotation operation the Itanium processor does relatively a better job because of its two store units. The second difference operation with a computational intensity $f = 1$ gives the expected 50% of the peak performance on the Itanium processor. On the MIPS

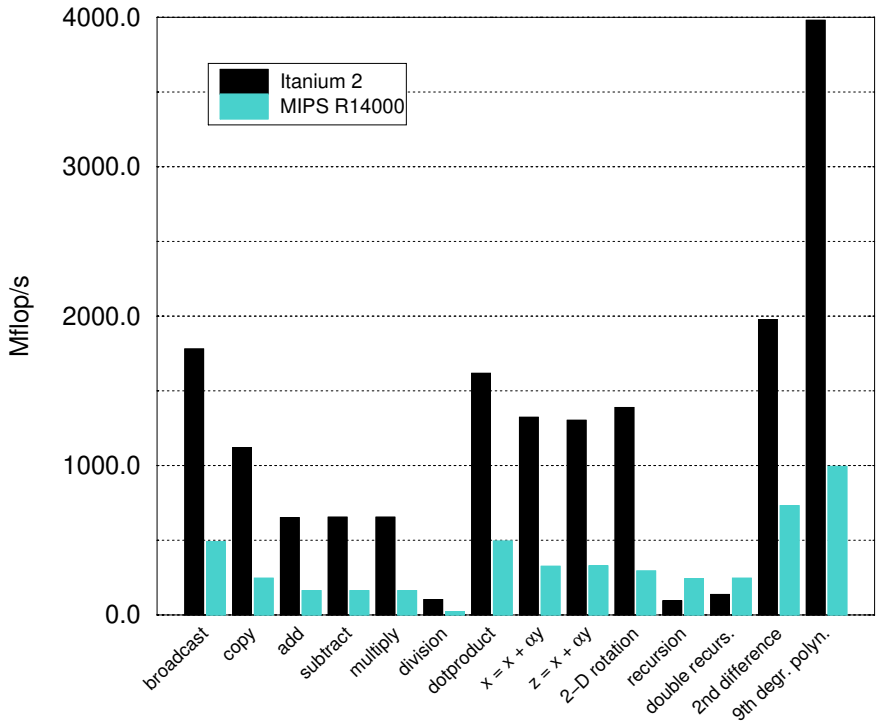

Figure 2.2: *Maximum observed performance, r_{max}, for the stride-1 basic operation kernels.*

processor, however, the efficiency is much higher, almost 75%. This is due to the fact that operands in the registers from the expression $x_{i+1} - 2x_i + x_{i-1}$ are reused in a next iteration. The operations that are atypical with respect to performance are the first and second order recursion operations where the MIPS processor attains an efficiency of about 20% and the Itanium shows an efficiency of 2–3%.

As remarked before, for the same operations but with a stride $\neq 1$ the performance roughly halves on the Itanium 2. Doing the operations via indirect addressing on the Intel platform these operations are approximately a factor of 3 slower than the stride 1 operations and the MIPS performances degrade by roughly 25% compared to the stride 1 operations. It seems that the MIPS processor can take some advantage of its dynamic scheduling capabilities here.

2.1.2 `mod1d`

Program `mod1d` tests the performance degradation that may take place when either in the source or in the target array of a vector operation a stride $\neq 1$ occurs. This degradation may be caused by the limiting effect of the bank cycle time, i.e. the time that a memory bank needs before it can accept a new memory access. Also, in a cache memory the strides may cause trashing because elements are mapped on identical cache lines in an address set. Furthermore, the behaviour for reading from or writing to a memory location is generally different. Typically, conflicts occur with strides that are a power of 2. Therefore in this tests mostly strides of this type are used. However, also some strides that are a products of prime numbers near a power of 2 are used to check whether degradation is mainly a result of a 2^n stride access or that a large stride as such has the same detrimental effect. In Figure 2.3 the effect for both processors is shown.

It is clear that for strides of the type 2^n , $6 \leq n < 17$ the performance deteriorates severely on the Itanium processor. For $n \geq 17$ the reading or writing of the 8-byte operands is done from/to the L3 cache and because of its size no interference occurs anymore which results in a partly restoration of the speed. At a vector lengths of 8199 and 140,001 no conflicts occur which results in higher performances both in the read conflict and write conflict case. It is also evident from Figure 2.3 that the performance level for write conflicts, i.e. where the target vector is written back with strides, is significantly lower than for read conflicts where the

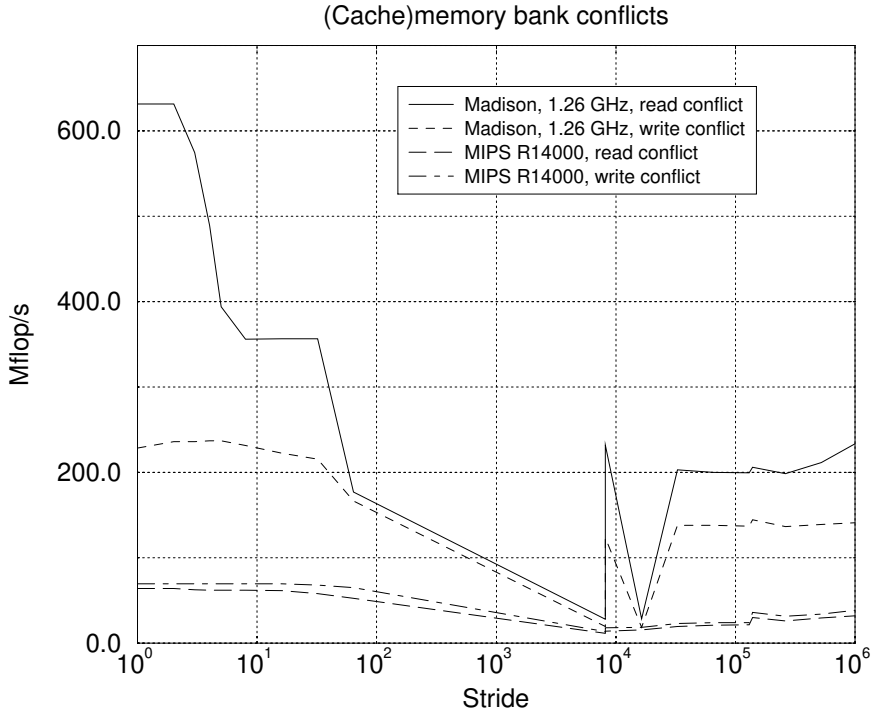

Figure 2.3: *Memory bank conflicts caused by strides in the access of array elements.*

source vectors are accessed with strides.

For the MIPS R14000 processor the initial performance of about 65 Mflop/s drops in the L2 cache to approximately 15 Mflop/s but no significant influence of the stride is noticeable here. However, when accessing from memory at a vector length 140,001 there is a slight performance improvement of about 10% over the speed of 30 Mflop/s that can be attained from memory.

2.1.3 `modle`

Although the accuracy of mathematical intrinsic functions strictly spoken should not be considered in a benchmark designed to assess the speed of operations/algorithms, it has often happened that the accuracy of some functions at least in certain argument ranges were of questionable quality. As there is little point in computing an incorrect result very fast a test for the quality of the intrinsics has been incorporated long ago and it has proved to be quite useful time and again. In Table 2.1 the results of the accuracy tests for both platforms are displayed.

Table 2.1 shows that the intrinsics are of high quality on the Itanium 2. A deviation of ≤ 1.00 decimal digit is normally acceptable and this condition is met on all intrinsic functions in all argument ranges that were tested. By contrast, the R14000 shows for the `Exp`, `Log`, and `Log10` functions deviations that are too large. Especially the `Log` function in the range (0.9999924, 1.000008) exhibits a huge deviation at 1.000000 of 7.59 decimal digits. In fact, when taking `Log(Log(x))` in this range it leaves hardly a significant digit intact which is totally unacceptable. Likewise, the `Exp` function in the range (-3.465736,-668.639) has a deviation that is on average about 1.4 decimal digits and a largest deviation that is almost 2 times more. This is definitely of a lower numerical quality than one may expect from a mature intrinsics library. For the other functions the Itanium processor is mostly slightly more accurate than the MIPS processor whenever the deviation is > 0.

Table 2.1: *Maximum deviations in decimal digits for the most important mathematical intrinsic functions*

Function	Intel Itanium 2 Dev. (dec. digits)	MIPS R14000 Dev. (dec. digits)
$x^{1.0}$	0.00	0.00
$(x^2)^{1.5}$	0.24	0.00
x^y	0.00	0.00
$\sqrt{x}$	0.00	0.00
e^x	0.30	2.71
$\log x$	0.67	7.59
$\log_{10} x$	0.71	1.07
$\sin x$	0.30	0.56
$\cos x$	0.30	0.46
$\tan x$	0.60	0.73
$\cot x$	0.63	0.75
$\arcsin x$	0.30	0.34
$\arccos x$	0.30	0.58
$\arctan x$	0.30	0.58
$\sinh x$	0.48	0.50
$\cosh x$	0.48	0.50
$\tanh x$	0.40	0.59

2.1.4 `mod1f`

Program `mod1f` tests the speed of the same intrinsic functions of which the accuracy was assessed in program `mod1e`. The resulting maximum speeds are shown in Figure 2.4.

The performance is here expressed as the number of Mega(function)call/s as the number of floating-point operations per function call and per processor is not fixed: it may differ depending on the argument value and the approximation algorithm used. The figure shows that the `Sqrt`, `Exp`, `Log`, and `Log10` functions the Itanium processor is significantly faster. This also shows in x^y where both the `Exp` and `Log` functions are involved. On the other hand, for the standard trigonometric functions the MIPS processor is 30–40% faster while for the inverse trigonometric functions and the hyperbolic functions the Itanium processor again is 25–30% faster.

For the functions tested also an estimate is done of the $n_{1/2}$ value. This should give an impression of the "start up time" for the respective functions. It turns out that these $n_{1/2}$ values are about equal for the Itanium processor and the MIPS processor. The absolute values for both are quite small: $n_{1/2}$ is mostly in the range of 4–8.

2.2 Module 2

Module 2 contains basic algorithms that either are part of a range of other important algorithms or themselves are the basis of the computational work as turns up in all kinds of applications. We discuss a subset of these algorithms for as far as they are of interest in understanding the behaviour of the processor/compiler combination. Many of these algorithms represent a linear algebra operation of some sort. Furthermore, FFT and Wavelet Transforms are addressed and one non-numerical algorithm, Quicksort is considered because it is so often a constituent in all kinds of applications, including scientific and technical ones. We discuss here a part of these algorithms for which the performance can be related rather naturally to the way the code is executed.

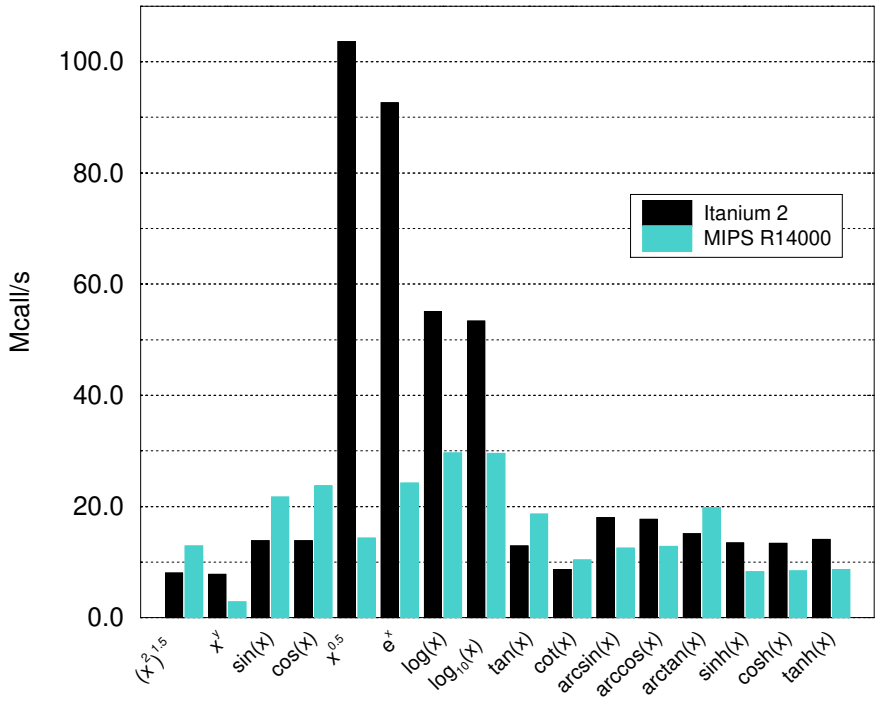

Figure 2.4: *Maximum observed speeds in Mcall/s for the mathematical intrinsic functions.*

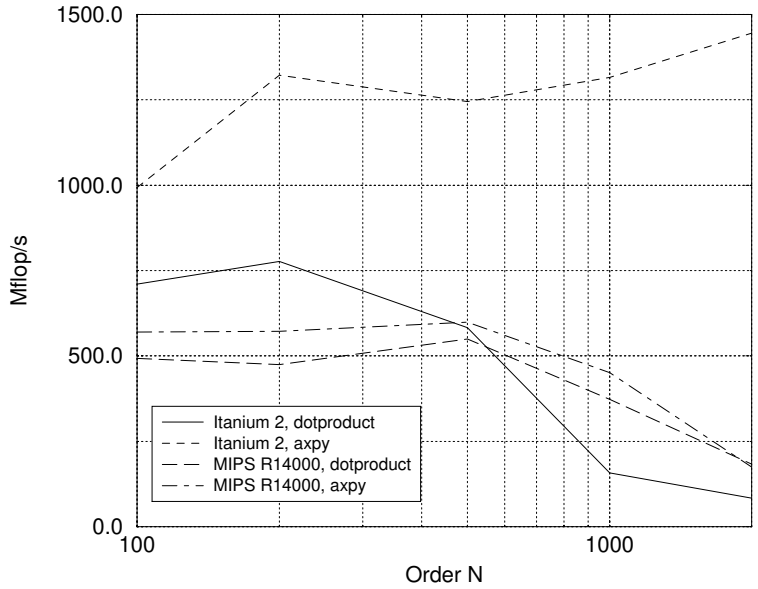

Figure 2.5: *Maximum observed speeds in Mflop/s for two implementations of the dense matrix-vector product* $Ab = c$.

2.2.1 `mod2a(s)`

Program `mod2a` looks at two different implementations of a dense matrix-vector multiply. The first one is a dotproduct implementation in which the computational intensity $f = 1$ is reasonable but the matrix elements are accessed with a stride $\neq 1$ and a vector update (`axpy`) implementation in which the computational intensity is only $f = 2/3$ but the matrix elements can be accessed with stride 1. For the Itanium 2 and the MIPS processor the behaviour for the two implementations is very different as can be seen from Figure 2.5.

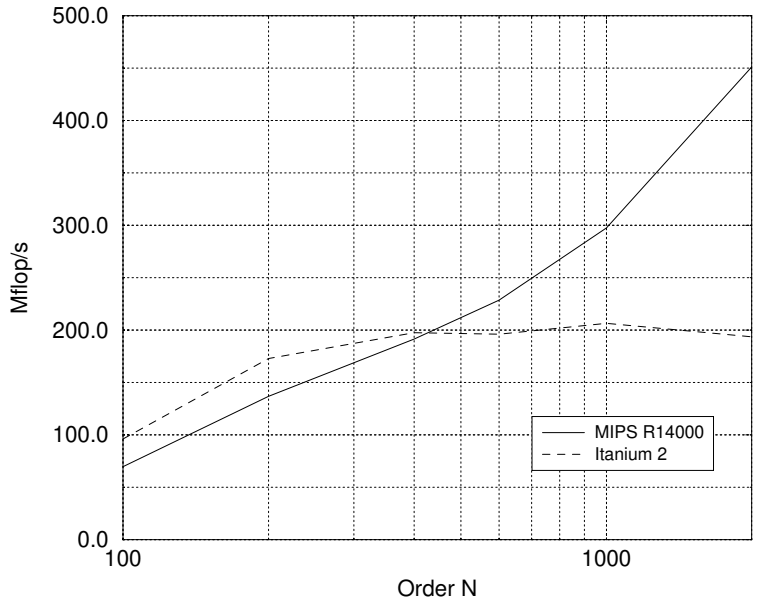

Figure 2.6: *Maximum observed speeds in Mflop/s for the sparse matrix-vector product $Ab = c$ with A in CRS format.*

It turns out that the dotproduct version on the Itanium 2 does not perform well: the stride of N, where N is the order of the matrix and vector, has a detrimental effect on the performance and this effect increases: elements have to be accessed from the L2 cache (at $N = 500$) and the L3 cache (at $N = 1000$). By contrast, when the matrix elements are accessed consecutively as in the `axpy` implementation the speed attains near the theoretical limit for the `axpy` operation, ≈ 1333 Mflop/s, as also found in program `mod1ac`.

For the MIPS processor the doptproduct version behaves as expected as long as the elements reside in cache: about 50% of the peak performance can be achieved.

`mod2as`

Apart from the simple full matrix-vector multiply an algorithm that is at least as important, if not more, is the sparse matrix-vector multiply that occurs in virtually all iterative sparse linear solvers. The implementation we chose for in program `mod2as` is based on the matrix being given in Compressed Row Storage (CRS) format. This means that the non-zero elements are accessed indirectly from a linear array by means of an index array. Effectively, 6 memory references are done of which the 2 indirect ones will access elements that are almost certainly located in memory and not in a cache. So, although the computational intensity is $f = 1/3$, the actual behaviour of the code will be worse because of the memory latency. In practice one therefore will seldom find a performance that exceeds 10–15% of the peak performance. This is confirmed by the results as as shown in Figure 2.6.

Here are again the speeds given as a function of problem size. In this case the amount of non-zero elements is also a parameter that is of interest. In all cases the fraction of non-zero elements is in the range of 3.5–7.5%. The MIPS processor does relatively well in the sense that from $N = 400$ on the performance is about 20% of the Theoretical Peak Performance. Clearly, the Itanium has more difficulties with smaller matrix sizes but it picks up speed for the larger orders. For the largest problem considered, $N = 2000$, the performance lies at 450 Mflop/s, 11.3% of the Theoretical Peak. This can be explained from the linearly increasing average loop length used for the accumulation of the result vector elements. It is to be expected that for the larger loop lengths better instruction schedules can be realised. By contrast the overhead in the smaller problem sizes is such that the performance is lower than that of the MIPS processor.

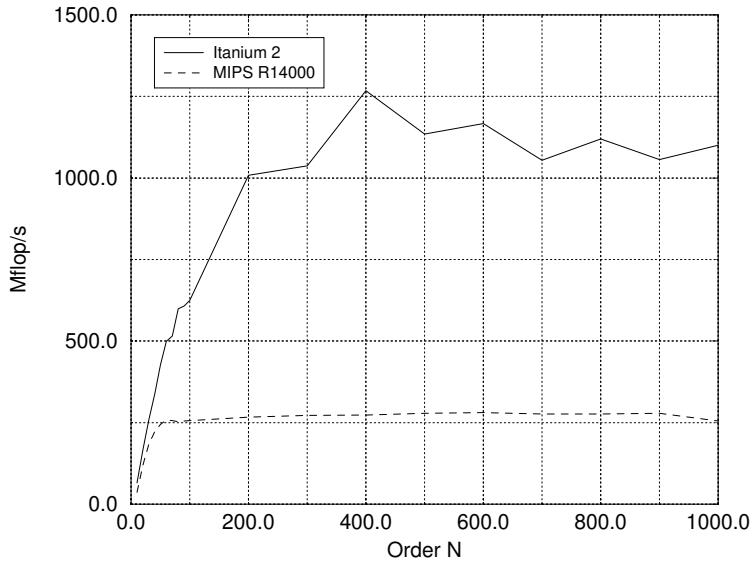

Figure 2.7: *Maximum observed speeds in Mflop/s for the solution of the dense linear system* $Ax = b$.

2.2.2 `mod2b`

Program `mod2b` solves a dense linear system using BLAS 3 and LAPACK routines. Systems up to order $N = 1000$ are considered. This test is identical to Jack Dongarra's optimised LINPACK benchmark as used to establish the TOP500 list where the problem size is increased to N being many hundreds of thousands. As such large dense linear systems do not occur in practice we limited our test to 1000 as an upper bound. The results are shown in Figure 2.7.

Because the BLAS 3 implementation is block-oriented it is using the cache fairly well and the performance does not deteriorate with increasing order of the system. It takes somewhat longer for the Itanium processor to arrive at an approximately constant performance level of around 1100 Mflop/s. The relative performance of both processors is identical: just over 25% of the Theoretical Peak Performance. The Itanium processor is somewhat sensitive to the problem size being an exact divisor of the cache block size being handled, but this a minor effect. We may assume that when on both systems the corresponding optimised linear algebra libraries would have been used, the performance would have been better. For instance, the cache block size has not been optimised for the respective processors but just a "reasonable" block size has been assumed. This reflects the practice in many user applications but could obviously be improved upon.

2.2.3 `mod2d`

Program `mod2d` implements the finding of all eigenvalues of a symmetric positive definite matrix. It uses the standard LAPACK routines that provide block oriented routines for good cache usage. In this respect it is rather similar to `mod2b`, be it that also other basic operations, like 2-D rotations are included in the algorithms. Results of the program for a range of 10–1,000 are shown in Figure 2.8.

It is clear from Figure 2.8 that the MIPS processor is doing relatively quite well: eventually a speed of around 600 Mflop/s is attained, 60% of the peak performance. In absolute speed the Itanium 2 processor is faster everywhere but the efficiency is lower: about 30% at best. This mainly due to the routines `dsyr2` and `dsymv`. Most of the time for the entire algorithm is spent in the matrix-matrix multiply routine `dgemm` which take almost 40% of the time on the MIPS processor and is equally efficient on both processors. The `dsyr2` and `dsymv`, in contrast, do less well on the Itanium 2 processor in the first routine because of the vector accesses for the rank-1 updates required and in the second one because of the matrix element accesses that are seemingly not done optimal. This also shows in the decreasing speed for matrix orders of $N > 500$ which indicates that the cache is not used optimally. This effect is hardly noticeable on the MIPS R14000. It seems

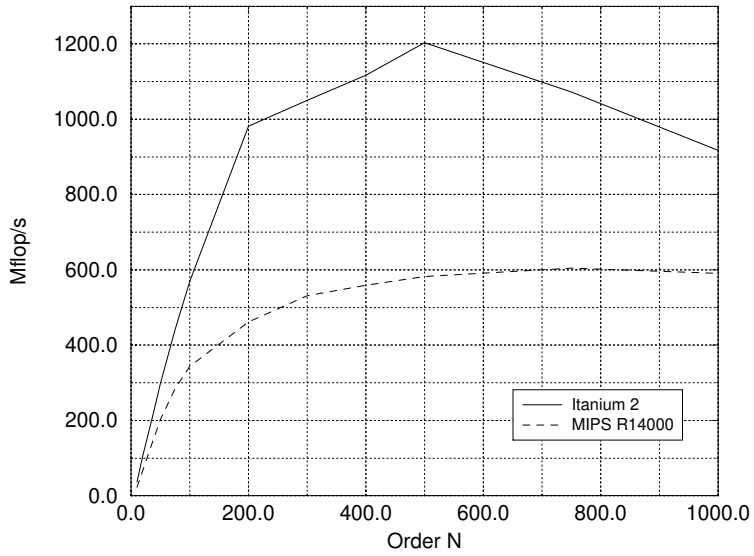

Figure 2.8: *Maximum observed speeds in Mflop/s for finding all eigenvalues of symmetric positive definite matrix A.*

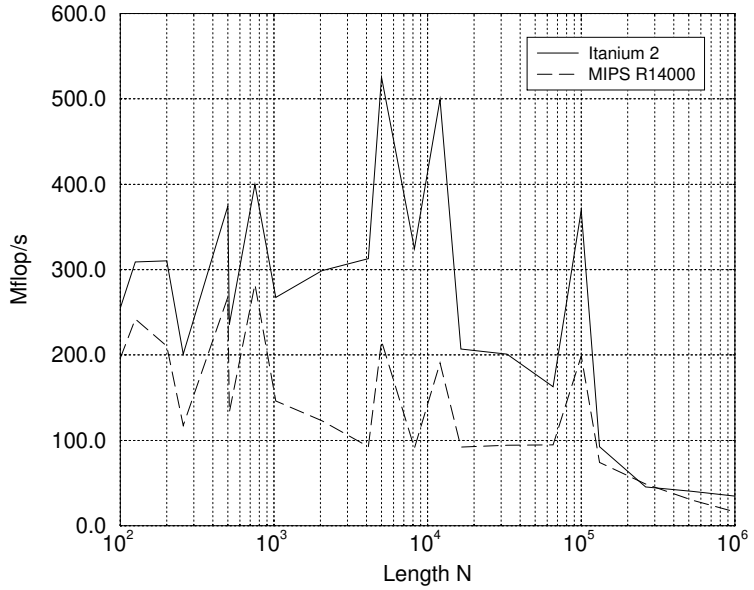

Figure 2.9: *Performance in Mflop/s of a 1-D complex-to-complex FFT in the range of* $N = 81$–$1,048,576$.

that the code on the Itanium needs some further optimisation. As in program `mod2b` we did not use the machine-specific optimised linear algebra libraries but relied on the standard Fortran 77/90 implementation.

2.2.4 `mod2f`

Program `mod2f` performs a set of 1-D complex-to-complex Fast Fourier Transforms ranging in length from $N = 81$–$1,048,576$. The majority of test lengths is of the form 2^n as is usual in many signal processing applications while a number of test lengths have a more general form. The results for this program are shown in Figure 2.9.

The first effect to be noticed is the speed differences for FFTs that have a length of the form 2^n and those

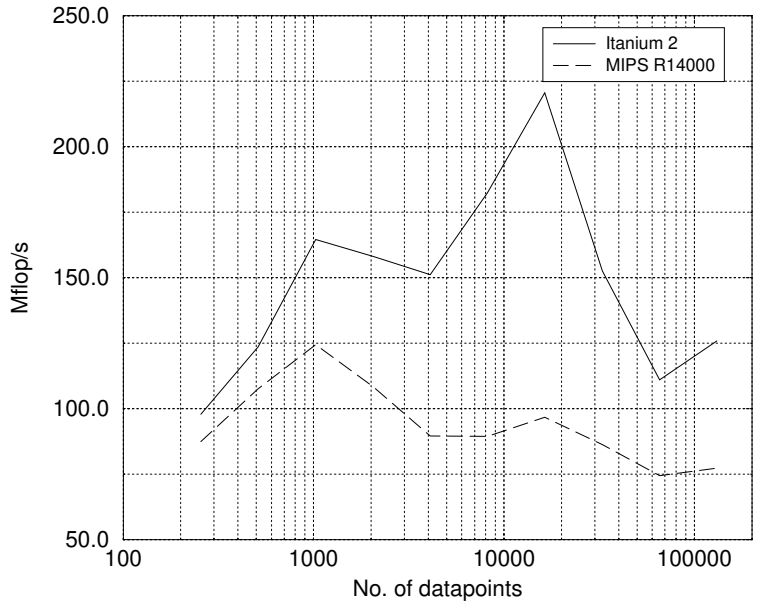

Figure 2.10: *Performance in Mflop/s of a 2-D Haar Wavelet Transform in the range of* $N = 16\times16$–512×256.

that have not: the ones that have a length that is a power of 2 are significantly slower than those for which the length $N \neq 2^n$ on both systems. As already was evident from program `mod1d` the memory access can slow down substantially when the (cache)memory is accessed in strides that are a power of 2. This is exactly what happens for FFTs with power of 2 lengths and the effect is quite clear. Furthermore, the Itanium 2 is often 1.5–2.5 times faster than the MIPS processor. Unfortunately, for many problem lengths the result on the Itanium 2 is incorrect. The severity of the errors have yet to be established. The problems seems to occur with the radix-8 routine within the algorithm but has not been localised exactly yet. This problem is peculiar to the code generated with the current `efc` compiler. Earlier results generated on an Itanium 1 system provided by Compaq did not exhibit this behaviour.

2.2.5 `mod2g`

Program `mod2g` implements a 2-D Haar Wavelet Transform. Because of the choice for the Haar transform, the amount of computation with respect to data access is very low: on average the computational density $f = 5/12$ for the analysis phase and the synthesis phase combined. So, one would expect the bandwidth from the (cache)memory to the CPU to be the dominating factor in the performance. The result is presented in Figure 2.10.

The sawtooth behaviour in the performance is due to an extra array copy operation that is required when the exponent n in the number of data points 2^n is even. Furthermore, the theoretical speeds of about 1680 and 420 Mflop/s for the Itanium 2 and the R14000 will never be reached because also a transposition of the data is required with 2^n data accesses but without any floating-point work. When working from the L1 cache with sufficient operations to schedule the Itanium 2 starts to perform relatively better than the MIPS processor up to a factor of 2.5. Working from the L2 cache this difference decreases again, showing the relatively high efficiency of the MIPS code.

2.2.6 `mod2i`

Program `mod2i` tests an efficient implementation of the Quicksort algorithm in a range 10,000–1,000,000 for 4-byte integers and 8-byte floating-point numbers. For this test the Intel Itanium 2 reference platform, the Altix 3000, with the 1.3 GHz Madison, and the MIPS R14000, 500 MHz were used. The results for the 8-byte floats are shown in Figure 2.11.

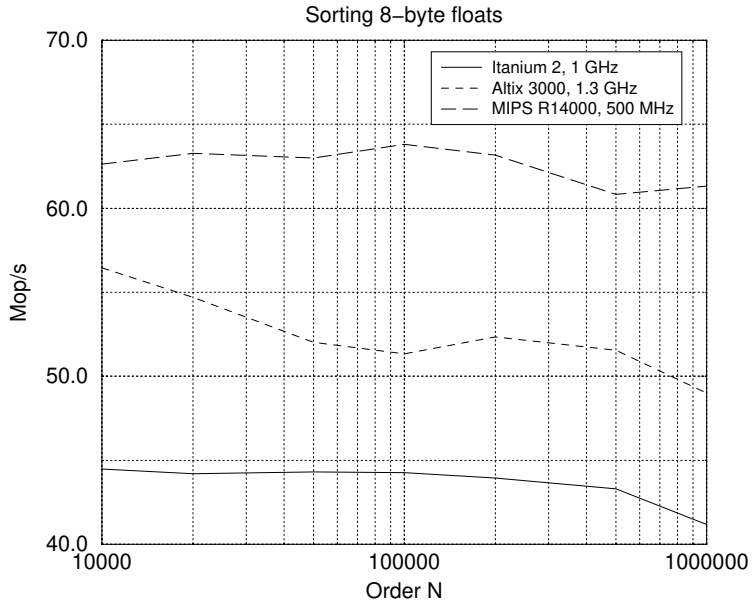

Figure 2.11: *Performance in Mop/s of sorting sequences of 8-byte floating-point numbers with lengths in the range of N = 10,000–1,000,000.*

Sorting is not the most fitting algorithm for the Itanium-based systems as is evident from the figure. On the Itanium 2, 1 GHz, the speed in Mega operation/s is about 44 for most of the range and it decreases slowly for larger problem sizes. On the 1.3 GHz Madison-based Altix 3000 the single-CPU speed scales not quite with the clock frequency (1.2 instead of 1.3, except for $N = 10{,}000$). The MIPS processor does clearly better here with an average of 62 Mop/s notwithstanding its much lower clock frequency. Another notable difference is that for the 4-byte integers the speed on the Itanium-based machines is 2 times higher, while it is only 16% higher on the MIPS processor. So, for integer sorting the Itanium-based systems are faster in all cases. This is at least partly attributable to the large amount of integer units in these processors (6) where the MIPS processor has only two.

2.3 Module 3

From this module only one basic I/O test is included as testing more elaborate compact applications needs a much more extensive analysis.

2.3.1 `mod3a`

Program `mod3a` tests the performance of a very large sparse matrix-vector multiplication $Ab = c$ in which matrix A and vector b reside on disk and the result vector c is written to disk. Both A and b are given as direct access files. In Figure 2.12 the results are given for the Madison-based system and for the MIPS platform.

In this case we show report the time for the Madison machine and not of the Itanium 2 system because on the latter the larger problem lengths could not be performed successfully:

```
Mod3a: Out-of-core Matrix-vector multiplication
-------------------------------------------------------------------------
  Row    | Column | Exec. time   |  Mflop rate |  Read rate  |  Write rate |
  (n)    |   (m)  |   (sec)      |  (Mflop/s)  |   (MB/s)    |   (MB/s)    |
```

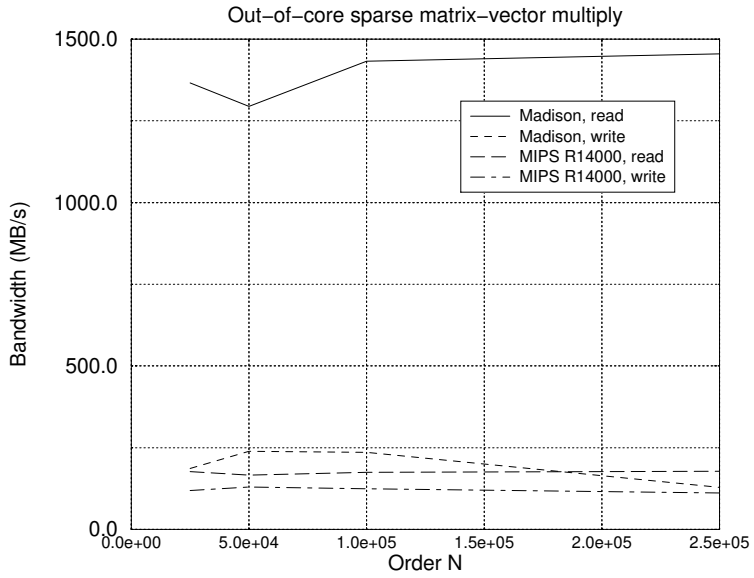

Figure 2.12: *Bandwidth in MB/s for an out-of-core sparse matrix-vector multiply* $Ab = c$.

```
-------------------------------------------------------------------------
  25000 |  20000 |  0.25847E-01|   38.689    |   362.72    |   58.401    |
  50000 |  20000 |  0.49379E-01|   40.503    |   361.89    |   82.768    |


Input/Output Error 173: Write Failure

   In Procedure: genraja
        At Line: 37

      Statement: Unformatted WRITE
           Unit: 1
   Connected To: Work File
           Form: Unformatted
         Access: Sequential
Records Read   : 1466
Records Written: 2445
```

On the Itanium 2 system we used a slightly older version of the `efc` compiler which may explain the disappearance of the problem but as I/O is involved it may also has been an I/O configuration problem.

Note that the read bandwidth for the Madison system is unrealistically high: an order of magnitude larger than one would expect for a normally configured I/O system. We suspect that in this case not I/O bandwidth but the bandwidth for cached data is measured.

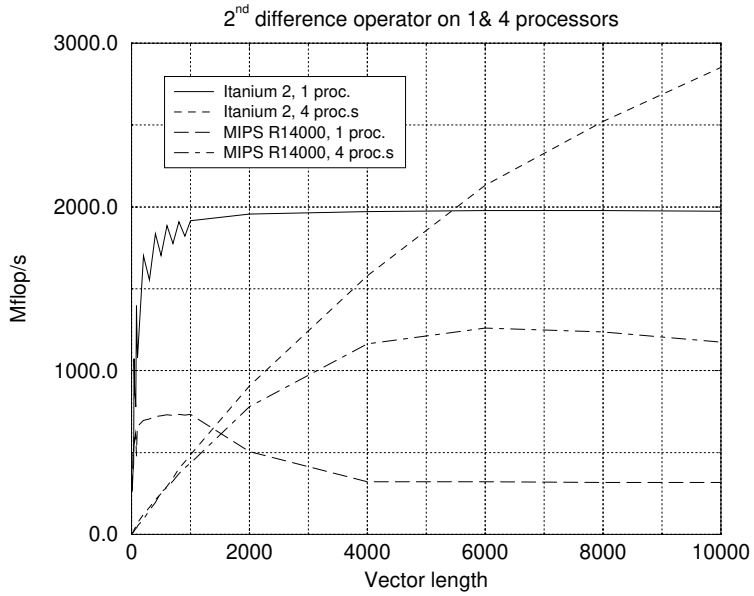

Figure 3.1: *Speed in Mflop/s for 2nd difference operator on a single CPU and with OpenMP on 4 processors.*

3 OpenMP results

In this section we present some results obtained from OpenMP experiments on a subset of the EuroBen V4.0 benchmark (see [9] for OpenMP details). For those programs that are amenable for parallelisation the parallel executables can be produced simply by adjusting the compiler flags (`-openmp` for the Intel platforms and `-mp` for the MIPS platform) and giving a `make omp` command. The OpenMP tests are of particular interest because they give a first indication of the parallelisation overhead incurred when loop bodies or function/subroutine calls are executed in parallel.

As the Itanium 2 cluster and the Madison server contained only 4 processors/node, we will show results obtained on 4 processors and compare them with the single-CPU results. As in the earlier section the programs from module 1, 2, and 3 are discussed in sequence.

3.1 Module 1

The parallelisation of the programs in module 1 is trivial. All tests that are performed in programs `mod1ac`, `mod1d`, and `mod1f` are based on loops that can be split simply over the available processors by a `!$omp parallel do` directive with static scheduling. In all cases this yielded the correct execution of the codes.

3.1.1 `mod1ac`

For loops with a simple body like the basic operations in program `mod1ac` parallelisation with OpenMP is rather disastrous: there is simply not enough work to compensate for the parallelisation overhead. Both on the Intel systems and on the MIPS-based machine this turns out to be quite high. As an example we show the result for the 2nd difference operator in Figure 3.1.

For the 2nd difference operator which has relatively much work to do in the loop body the speed on 4

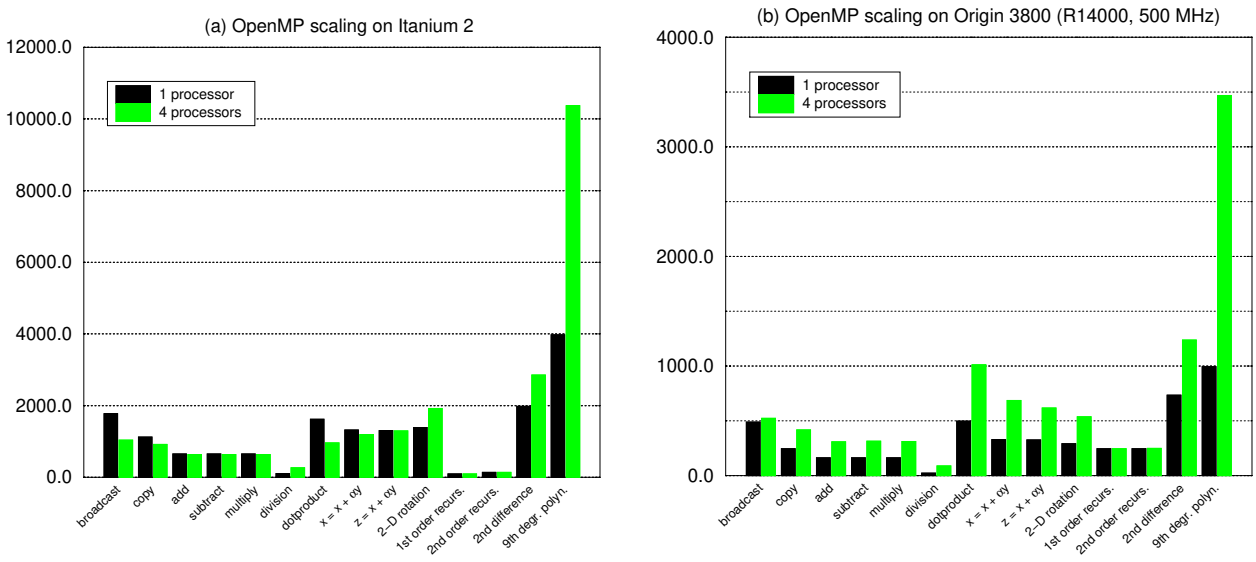

Figure 3.2: *Comparison of speeds for basic operations on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

processors only exceeds that of a single processor at a vector length of 5,500 on the Itanium system and 1,800 on the MIPS system. This is in line with the observed $n_{1/2}$ values ($n_{1/2}$ is the vector length where half of the maximum speed is obtained and, as such, a measure for the parallelisation overhead). These values are 7500 and 5500 for the Itanium and the MIPS processor, respectively. By contrast, the $n_{1/2}$ values on a single CPU, characterising the vector startup time are only 70 and 17 elements for the Itanium and MIPS processor, respectively. It will be clear that for simpler operations like addition or an `axpy` operation hardly anything is to be gained. In fact some operations will be slower as can be seen in Figure 3.2.

It is clear from the figure that the synchronisation overhead is relatively lower on the MIPS system resulting in modest gain in performance for the more substantial operations like the 2-D rotation, the 2^{nd} difference, and the evaluation of a 9^{th} degree polynomial. The last operation is the only one that has sufficient work in the loop body to cause a reasonable speedup. Of course there is no speedup for the recurrence operations because they are inherently unparallelisable.

3.1.2 `modlf`

We have also looked at the speed of the intrinsic functions with OpenMP to see whether we could get an performance gain. Because the amount of work per function evaluation is usually substantially larger than for the simple operations one would expect a fair speedup in this case. The comparison for the speeds in Mcall/s for 1 and 4 processors is given in Figure 3.3.

Figure 3.3 shows not only that the scaling is generally quite good but even super-linear for some of the more difficult functions like `Tan`, `Arcsin`, and the hyperbolic functions on both machines. This is understandable because of the vector length for which the performance is reported: $N = 20{,}000$. For smaller vector lengths the super-linear speedup disappears because all data fits in the L1 cache. Note that for $\cot(x)$ the speedup is less than linear. This is because it is dominated by the division to be done. This operation is slow on both platforms and does scale less than linear on both machines.

It is also evident from the figure that the speed for most intrinsic functions is about the same on both systems except for the `Sqrt`, `Exp`, and `Log` functions where the Itanium 2 is faster by a factor of 2–6. As already remarked in section 2.1.3, the accuracy of the function evaluations is significantly better on the Intel platform.

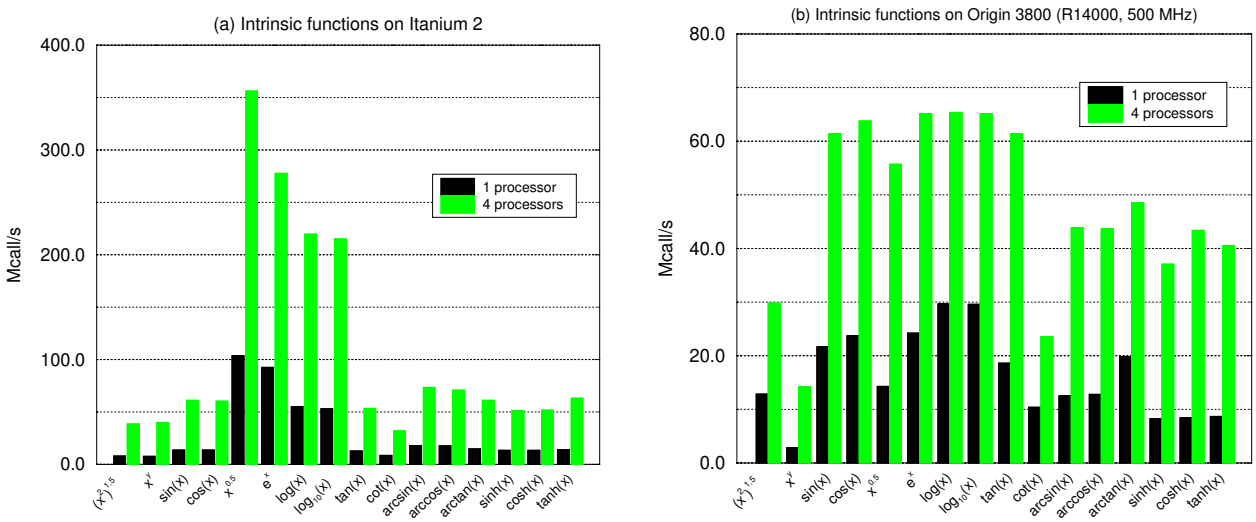

Figure 3.3: *Comparison of speeds of intrinsic functions on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

3.2 Module 2

In this section we compare the scalability of some basic algorithms as also occur in the single-CPU benchmark. We discuss a subset for which OpenMP parallelisation is readily done. As generally more work is done in loops in these algorithms one might hope to find a reasonable speedup for most of the algorithms.

3.2.1 `mod2a(s)`

Program `mod2a` measures the speed of dense matrix-vector multiplies and as such has a quite simple structure both for the dotproduct and the `axpy` variant. However, to obtain correct results while keeping the algorithm reasonably efficient one has to accumulate the result vector c locally before copying it into the global result vector. This involves the computation of start and end addresses of local vectors and extra data copies that noticeably slow down the computation. For that reason the `axpy` variant only begins to pick up speed at the larger matrix orders. The result for both machines is displayed in Figure 3.4.

For the dotproduct form the speedup is generally 2–2.5 with respect to one processor. A notable exception is the case of $N = 500$ on the Itanium processor where on 4 processors the partial data still fit in the L1 cache while on a single processor data have to be accessed from the L2 cache already. The speedup in here almost 4. No explicit loop unrolling has been done in this code. This will probably result in a better behaviour both of the single-CPU and the OpenMP code.

As remarked, to obtain correct results for the `axpy` variant we must make local copies of the the partial data handled in the processors in the parallel region of the algorithm. This, and the synchronsation needed result in a lower sizes. For orders $N > 1000$ the OpenMP version overtakes the other variants because it is still in a favourable cache regime.

In the sparse matrix-vector multiplication, `mod2as`, one can expect that the results with OpenMP will not be very favourable when using the CRS matrix format. This is indeed confirmed in Figure 3.5.

The figure shows that the 4-processor results are significantly slower on both machines. This is due to the low computational intensity ($f = 1/3$) and the fact that almost no assistance from the caches can be expected. For the OpenMP version this situation is compounded by the problem that loops on the different processors must synchronise while the amount of work per loop iteration need not be the same: it is dependent on the number of non-zero row elements. Although the differences over the loop iterations alloted to each of the processors may cancel out, the wait time in the synchronisation is non-negligible. This combined by the

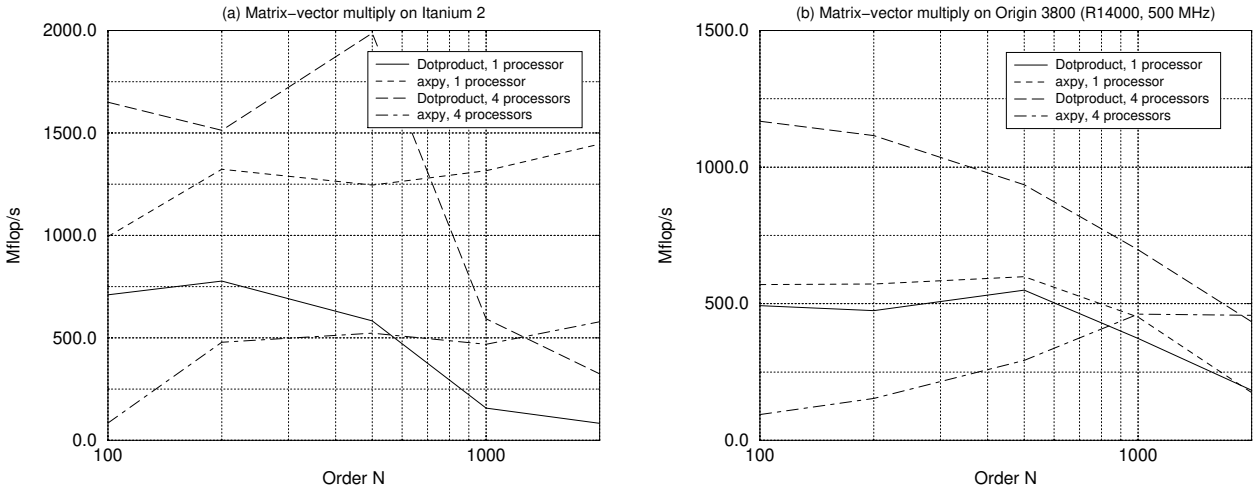

Figure 3.4: *Comparison of speeds of dense matrix-vector multiply on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

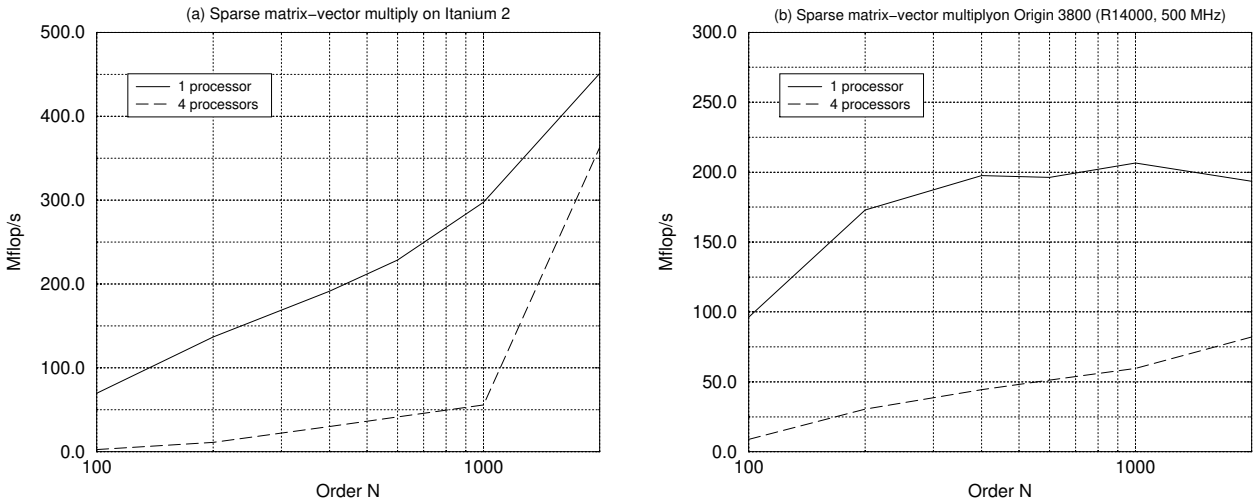

Figure 3.5: *Comparison of speeds of sparse matrix-vector multiply on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

startup time to be amortised over smaller vectors make the sparse matrix-vector multiplication from CRS format at least for order $N \leq 2000$ less than a success. The remarkable recovery on the Itanium 2 and to a lesser extent on the MIPS system for $N > 1000$ lead to the expectation that for very large problem sizes OpenMP can be beneficial. An alternative approach would be to employ a Jagged-Diagonal format for the sparse matrix. Although this needs somewhat more data manipulation to start with, one might be able to get some benefit from cache assistance. Again, this will work better for large problem sizes.

3.2.2 `mod2b`

The solution of dense linear systems, `mod2b` should be reasonably fit to work with OpenMP because of its blocked implementation in LAPACK provided that enough work can be done in the loops of the core routines like `DGEMM`. The result for both machines for solving systems up to an order of $N = 1000$ is shown in Figure 3.6.

When using the `-openmp` flag on the `efc` compiler it was not able to generate the code for routine `DGEMM` with

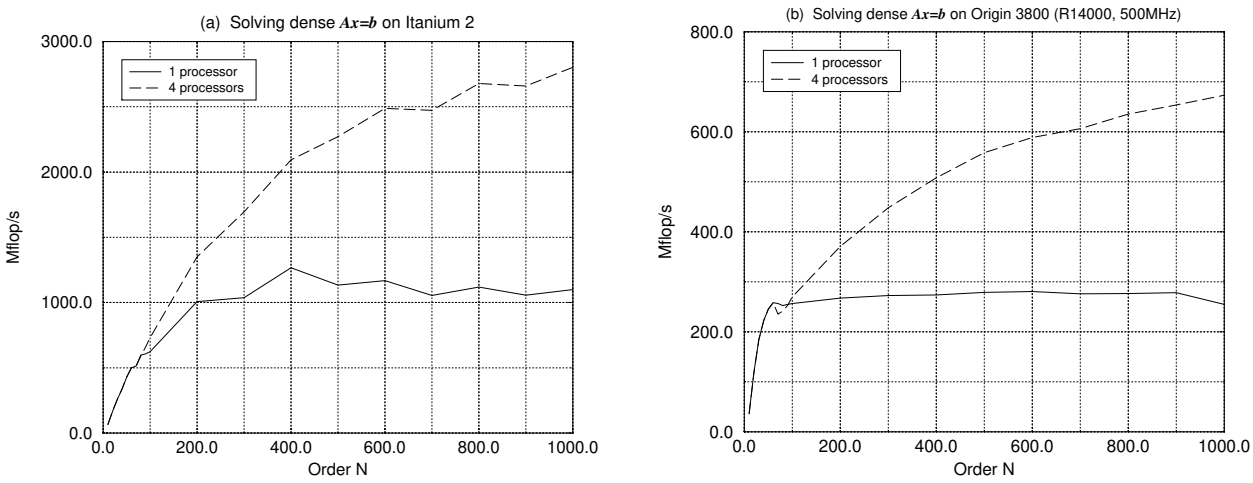

Figure 3.6: *Comparison of speeds of dense linear systems $Ax = b$ on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

all the directives added to the Fortran source. It reported an internal compiler error after parallelising 7 out of 8 loops. However, with the `-parallel` flag for autoparallelisation it was able to produce an object file that linked and executed correctly. With this option it parallelised 4 of the 8 loops and gave the performance as shown in Figure 3.6.

The figure also shows that for sufficiently large problem order both systems reach a speedup of about 2.5 at $N = 1000$ while the absolute speed of the Itanium 2 processor is roughly 3.5 times higher than that of the MIPS processor both in the single CPU case and in parallel.

3.2.3 `mod2d`

The solution of a dense symmetric eigenvalue problem as implemented in program `mod2d` should somewhat behave like the algorithm in the former section. However, the diagonalisation takes a significant part of the time and the parallel work to be done on a loop level in this part is relatively low. Again the relevant blocked LAPACK routines are used to ensure good cache behaviour. Because also in this algorithm the BLAS 3 matrix-matrix multiplication routine `DGEMM` is used this routine was again separately compiled with the `-parallel` flag. The performances for both systems are displayed in Figure 3.7.

The figure makes clear that for dense eigensystems for $N \leq 1000$ OpenMP parallelisation does not help. For the Itanium 2 machine at $N = 1000$ the parallel version is slightly faster thanks to the fact that the single CPU version runs out of cache for problem orders of $N > 500$. Although also the MIPS system shows a modest performance increase for the parallel version the initial performance is so low that it stays behind the single CPU version everywhere. To benefit from parallelisation at least some of the routines like `dsyr2` and `dsymv` involved in the diagonalisation process should be rewritten entirely.

3.2.4 `mod2f`

For the program `mod2f`, a mixed radix 1-D complex-to-complex FFT we only report the results for the MIPS-based system as on the Itanium the radix-8 routine was not translated correctly, resulting in faulty outcomes for the single-CPU code, see section 2.2.4. The results for the Origin 3800 are shown in Figure 3.8.

The 1-D FFT does not benefit from parallelisation except very slightly for problem sizes of about $N > 2 \cdot 10^5$. This is due to the fact that in this implementation only the inner loops in the radix-m routines can be parallelised. As for the smaller sizes of N the number of loop iterations is rather limited and per iteration many intermediate private variables are created, the overhead per loop is considerable. This results in the performances as observed and displayed in Figure 3.8.

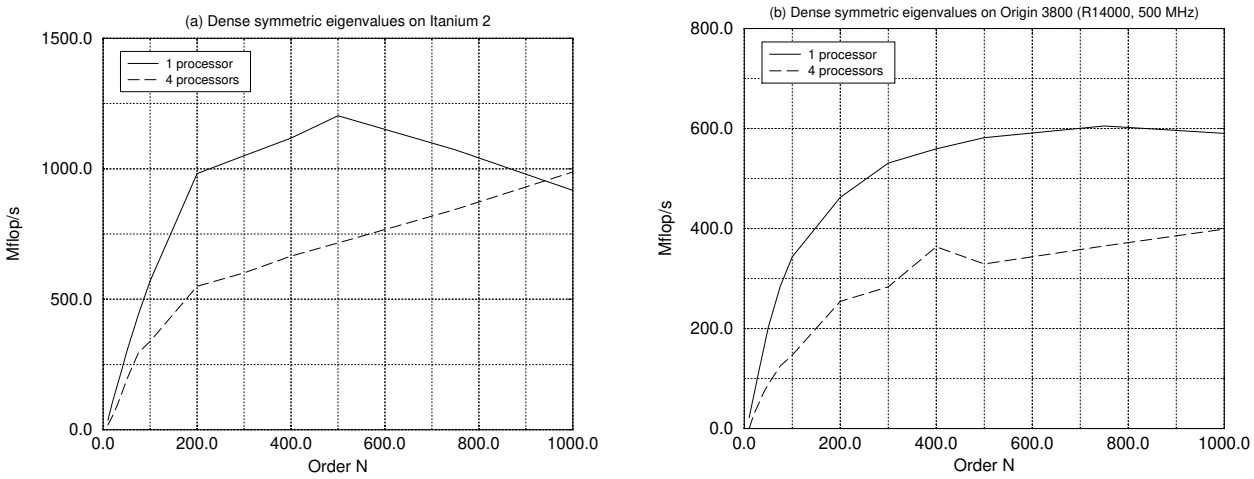

Figure 3.7: *Comparison of speeds of dense symmetric eigenvalue problems $Ax = \lambda x$ on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

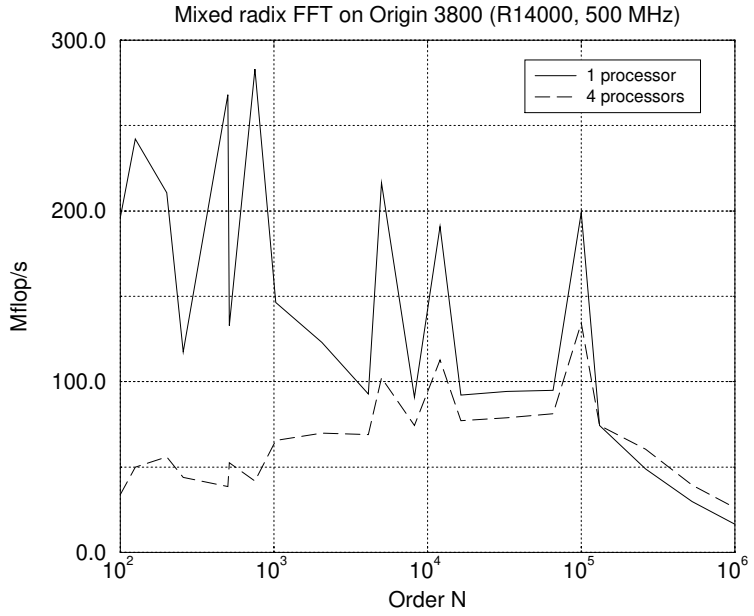

Figure 3.8: *Performance in Mflop/s of a 1-D complex-to-complex FFT in the range of $N = 81$–$1,048,576$ on a single CPU and on 4 processors using OpenMP. Only the results for the MIPS-based Origin 3800 are shown.*

The OpenMP version on the Itanium 2 system compiled and linked without problems although the compilation of the FFT subroutines `fftcc3.f` and `fftccg.f` took an inordinate amount of time. However, when running the program it stalled when entering the FFT-routines without any notification. Because of the limited test time we were unable to pinpoint the cause of the problem. The same experiment with the 1.26 GHz Madison test platform and the 7.1 version of the `efc` compiler *did* run and gave the same errors as in the single-CPU version.

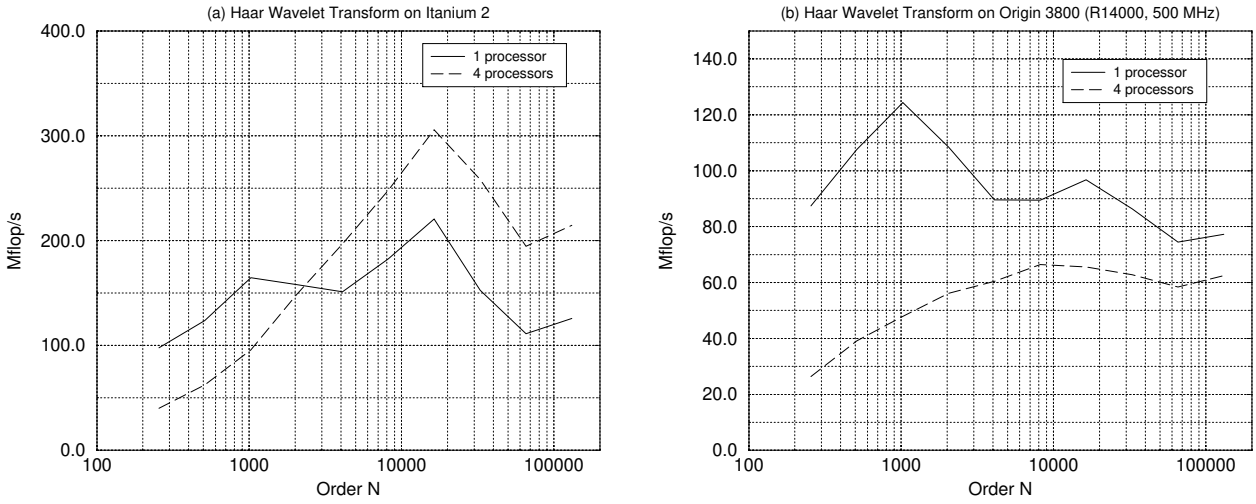

Figure 3.9: *Comparison of speeds of 2-D Haar Wavelet Transform on a single CPU and with OpenMP with 4 processors on the Itanium 2 platform (a) and the MIPS R14000 platform (b).*

3.2.5 `mod2g`

In this 2-D Haar Wavelet Transform, `mod2g`, one can parallelise on a fairly high level: the same 1-D transform can be called in parallel and, although the amount of work within the 1-D transform is quite small, one could expect some benefit. This appeared to work out differently on the Itanium and the MIPS based machines as shown in Figure 3.9.

On the Intel system for $N > 2048$ the parallel version is consistently 1.5–2 times faster while on the MIPS system the parallel version for $N > 10^6$ the parallel version is still slower. Furthermore, for the sequential version the Itanium was faster by a factor of 1.25–2.0 depending on the problem size. In the parallel version this speed difference is obviously increased, ranging from 2–5 times faster. The bad speedup behaviour on the SGI machine is unexpected and needs further investigation.

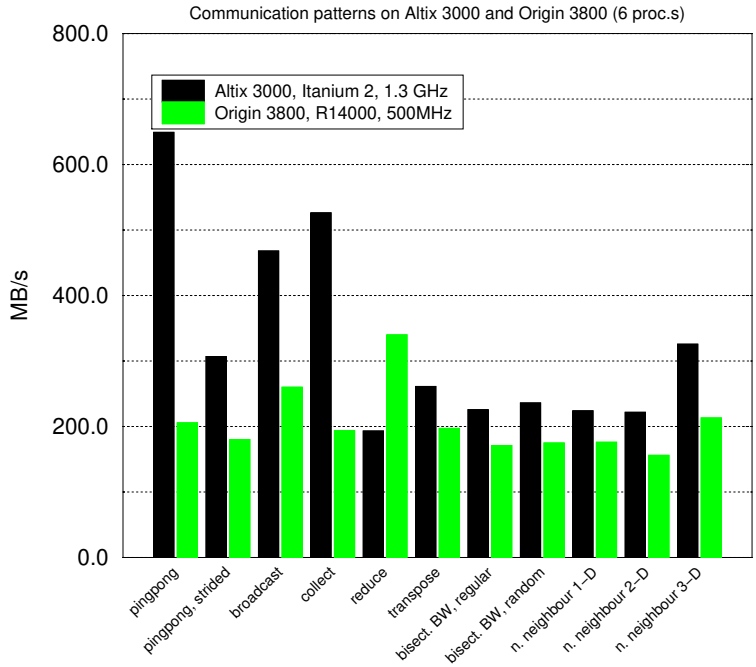

Figure 4.1: *Bandwidth in MB/s for a number of communication patterns in MPI. The Itanium-based results are in this case obtained on an SGI Altix 3000 system also using SGI's MPI implementation.*

4 MPI results

The tests of the Itanium 2 system was concluded by doing some MPI experiments (see [6, 7] for MPI details). For this the distributed memory version of the EuroBen Benchmark was used. It consists of the investigation of some basic communication operations and again considering basic algorithms as also appear in the single-CPU and OpenMP version of the benchmark. This enables comparison of the shared-memory and distributed-memory parallelisation paradigms for these algorithms.

4.1 Module 1

Module 1 of the distributed memory benchmark contains no basic arithmetic operations as in Module 1 of the single-CPU and OpenMP benchmarks. We can safely assume that these operation per processor are the same as in the sequential case, except for the reduction operations like the dotproduct. This is why the distributed dotproduct is tested in this module. Apart from that only communication parameters are measured.

4.1.1 `mod1h`

Program `mod1h` tests a range of important simple and collective MPI primitives. It yields the maximum attainable bandwidth for these operations so that they can be used to estimate communication time in the parallelisation model of an application. Figure 4.1 shows the bandwidth for the operations of interest.

The Intel Itanium 2 reference platform was fitted with a Myrinet 2000 network (see [8] for the network

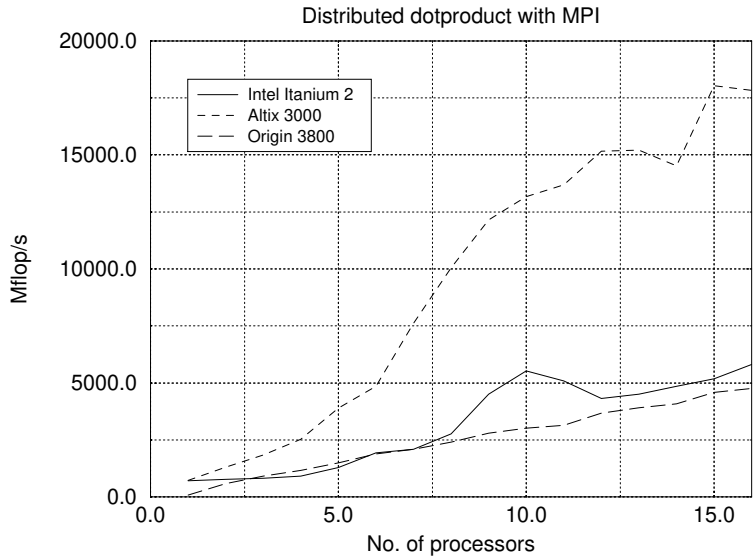

Figure 4.2: *Speed of the distributed dotproduct with MPI. The Itanium-based results are obtained on Intel's Itanium 2 reference platform, on an SGI Altix 3000, and on a MIPS-based SGI Origin 3800.*

specifications) and the public domain MPI version MPICH 1.2.5. Unfortunately, this configuration was not able to perform some of the the communication patterns we were interested in, viz., the 1-D, 2-D, and 3-D nearest neighbour patterns. We therefore show here results obtained with a 1.3 GHz Madison-based SGI Altix system. Computation-wise this makes no difference as only messages are sent around. However, there is some additional interest in these results because the networks of the MIPS-based Origin 3800 and the Altix 3000 have a similar structure. The network in both machines consists essentially of a hypercube structure with a denser connection at the nodes. In the Origin 3000 series the connecting components are so-called Numalink 3 connections, in the Altix 3000 Numalink 4 is used in the compute nodes which have a hardware bandwidth specification that is 2 times higher than that of Numalink 3. Between nodes, however, Numalink 3 is employed. As MPI on the Altix machine should be a one-to-one port of SGI's MPI for the the Origin systems, performance differences should be due purely to the network differences.

For this test 6 processors are used to accommodate 3-D nearest neighbour communication. In some MPI implementations this might result is a more favourable configuration of the communication when using the Cartesian grid features of MPI. On can see in Figure 4.1 that for most of the patterns the Altix machine behaves roughly as one would expect: the peak bandwidth is about 15–20% higher. However, the `broadcast` and `collect` operations do somewhat better because of the tree structure employed in these operations in which there is some extra benefit from the faster near-processor communication. For the pingpong experiment this behaviour is extreme: the allocation of processors is done pairwise and all pairs reside in the same node. To assess the inter-node bandwidth the processes should be placed explicitly on different nodes using SGI's `dplace` facility as MPI has no (official) knowledge about the physical location of the processors in the network.

The only operation that not behaves as expected is the `reduction`. Surprisingly the Origin shows a 40% higher bandwidth in this case. We have to investigate this further as it is also clear from other tests in which the `reduce` operation is involved (see 4.1.2) that it behaves better in other circumstances.

4.1.2 `mod1i`

The dotproduct operation is a reduction operation and, as such, must be explicitly measured to establish the speedup with respect to the single-CPU performance. Program `mod1i` measures the the speed of the distributed dotproduct with three different implementations: the first is the "naive" way in that it accumulates the local sums on each processor and then sends these local sums to a root processor that computes

the global sum and sends this global sum to all other processors via a primitive `MPI_Send` operation.
In the second implementation a tree communication structure is explicitly defined in a Fortran subroutine and all simple `MPI_Send` and `MPI_Rcev` operations concerning the collection of the partial sums and the distribution of the global sum are done via this tree.
In the third implementation the collective MPI operations `MPI_Bcast` and `MPI_Reduce` are used. It turned out that in all cases the Fortran implementation was significantly faster than the first, simple implementation and marginally faster than the implementation using the collective MPI calls. In Figure 4.2 the results for up to 16 processors is shown for the Intel reference platform with Myrinet, the SGI Altix 3000, and the Origin 3800 for this fastest variant.

Note that in this test, where a dotproduct of length $N = 1,000,000$ is computed also the processor speed is important. However, our primary interest here is the scalability of the operation, not the absolute speed. A complicating factor is that at some point (on the Itanium processors between 6–8 processors and on the R14000 between 7–9 processors) the partial dotproducts begin to fit in the L2 cache which is noticeable by a more than linear speedup in these processor ranges. When one however extrapolates from the highest number of processors one finds a speed of about 1125 Mflop/s/processor for the Altix system and 297 Mflop/s for the Origin. Both these speeds are consistent with the speeds found for dotproducts of a length of $N = 62,500$, which means that the communication is not a noticeable bottleneck for reduction operations of this size. As the distributed dotproduct using the `MPI_Reduce` operation is only slightly slower on both systems than the results shown in Figure 4.2 this means that this operation is functioning sufficiently well to give linear scaling. This is comforting in the light of the speed found for the `reduce` operation as measured in program `mod1h` on the Altix system. One could draw the conclusion from this that when a modest amount of work is done in relation to the `reduce` operation the communication is not a bottleneck.

On the Intel reference platform the scalability is clearly less than linear. The per processor speed found here is 363 Mflop/s, about half of what could be expected. The network on this platform is far from homogeneous and, in addition, a less optimised public domain MPI version was used here which in part can explain the less than perfect scalability.

4.1.3 `mod1j`

In program `mod1j` a very precise pingpong experiment is done in order to establish both the bandwidth for long messages and the latency for short messages in case of point-to-point communication. For the Intel reference platform and the Origin 3800 the bandwidth results are shown in Figure 4.3.

Note that the bandwidth levels off at message lengths of about 1.5 times L1 cache size because then also the internal processor bandwidth starts to play a role. The same effect could be seen in the bandwidth for the strided pingpong operation as measured in program `mod1h` (see Figure 4.1) were the messages for longer message lengths do not fit in cache anymore. The Origin Numalink network clearly does better than the Myrinet network for messages lengths between 100–50,000. Also the latency for short messages is better: for the Origin it is 3μs while it is 7μs for the Myrinet network.

4.2 Module 2

In this section we discuss some of the basic algorithms that either in the single-CPU case or with OpenMP also were considered. This gives us the opportunity to assess the scalability *per se* and as compared to the OpenMP results.

4.2.1 `mod2a`

We again have looked at the dense matrix-vector multiply `mod2a`. In this program the rows of the matrix are distributed as evenly as possible over the processors to minimise load imbalance. With the problem sizes chosen in this case, orders of $2^n \geq 32$ and 16 processors the load balance should be perfect. Furthermore vector b in $Ab = c$ is present on all processors. This means that in the non-transposed case for A no communication is to be done at all, leaving the appropriate parts of result vector c on each processor. When

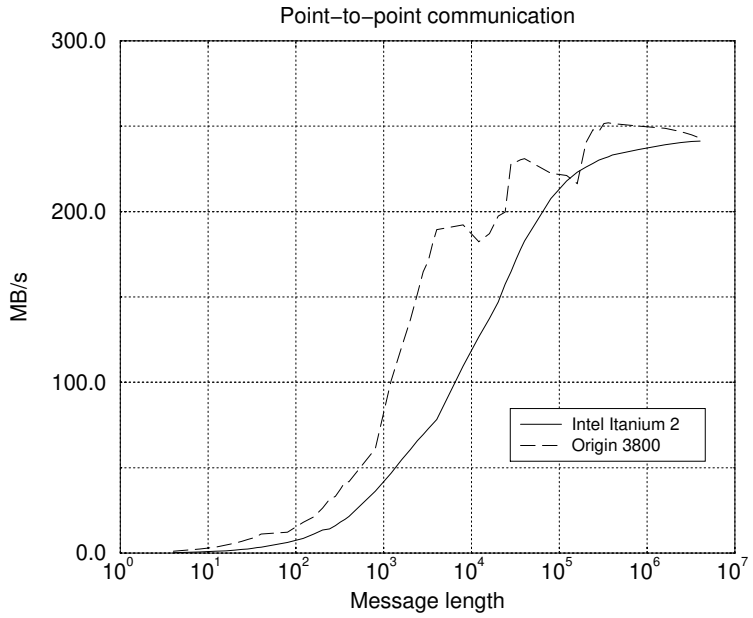

Figure 4.3: *Point-to-point bandwidth as a function of message length with MPI. The results are obtained on Intel's Itanium 2 reference platform and a MIPS-based SGI Origin 3800.*

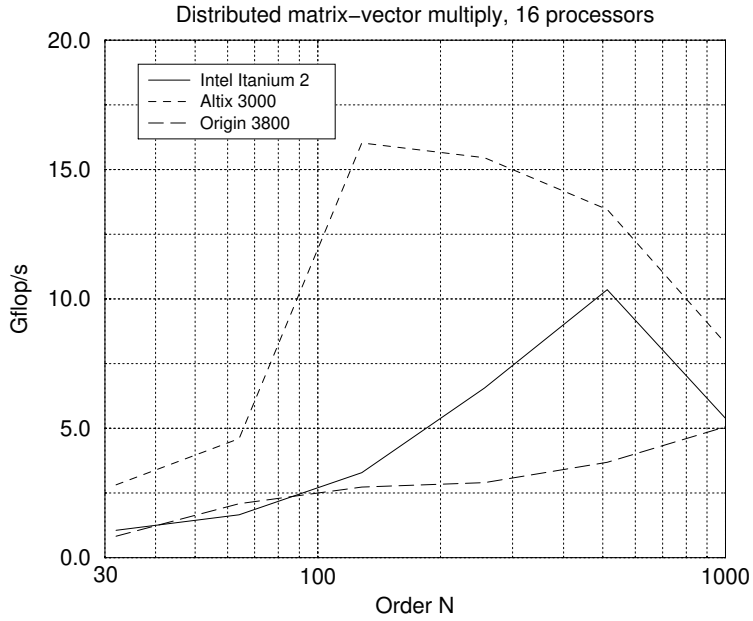

Figure 4.4: *Speed of a distributed dense matrix-vector multiply $Ab = c$ with MPI. The results are obtained on Intel's Itanium 2 reference platform, the SGI Altix 3000, and a MIPS-based SGI Origin 3800.*

operating on the transpose A^{T} partial results have to be combined via a `sum-reduce` operation.

Of the eight variants tested: dotproduct and `axpy` form in simple and 4 times unrolled form with using A and A^{T}, the 4 times unrolled `axpy` version operating on A was clearly the fastest. We show the results in Figure 4.4 for the Intel Itanium 2 reference platform, the Altix 3000, and the Origin 3800.

For the MIPS-based Origin the performance steadily increases over the range of $N = 32$–1024. For the

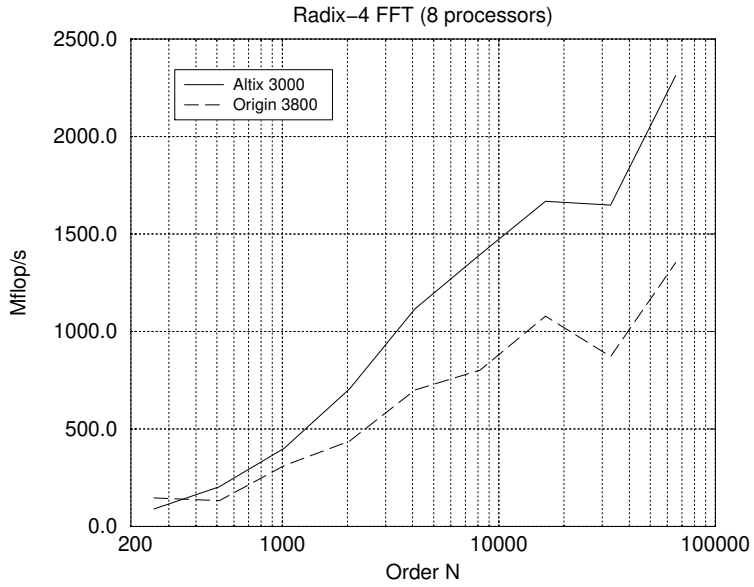

Figure 4.5: *Speed of radix-4 distributed 1-D complex-to-complex FFT with MPI. The results are obtained on the SGI Altix 3000 and a MIPS-based SGI Origin 3800.*

Itanium-based systems, however, there are marked peaks with a highest speed of almost 1 Gflop/s/processor at $N = 128$ for the Altix machine. After these maxima the performance decreases on the Itanium-based machines for no clear reason. Cache utilisation should be high as well as register re-use. As we during the tests had no performance analysing software available it is hard to identify were the problem lies. We will have to look into this in a later stage. In absolute speed the Itanium-based machines are obviously faster in most of the problem range. In processor efficiency the MIPS processor does evidently better for the larger problem sizes, that is, without a further manual optimisation of the code.

4.2.2 `mod2f`

In the distributed-memory version of the 1-D complex-to-complex FFT code a simpler radix-4 algorithm is used that makes it easier to factorise the sequence to be transformed. Furthermore, a transposition and a complex elements-wise matrix multiply has to be done that are additional with respect to the sequential algorithm and add to the flop count thus proportionally diminishing the floprate of the algorithm. So, apart from the necessary communication, which is $\mathcal{O}(n)$, there is also a computational factor that slows down the parallel computation. Furthermore, extra array copies are needed for transform lengths $N \neq 4^{2n}$ which also diminish the overall speed. For FFTs of reasonable lengths however still a good performance should be possible. The results on the Intel Itanium platform, the SGI Altix, and the SGI Origin are given in Figure 4.5.

The sequential performance of the Altix system turns out to be nearly twice that of the Origin machine with speeds of about 200–450 and 100–230 Mflop/s, respectively. The behaviour on 8 processors is largely similar with reasonable scaling for the larger transform lengths with roughly 300 and 150 Mflop/s/processor for large transform lengths, about 2/3 of the sequential speed per processor due to the decelerating factors mentioned above. Note the jumps in performance for $N = 256$, 4096, and 65,536 as no additional array copies have to be made that also interfere with the cache. On the Itanium 2 reference platform we used the 7.0 version of the `efc` compiler. This resulted in code that would not run correctly for some processor counts and not at all for other processor counts. This problem was solved with the 7.1 version of the compiler.

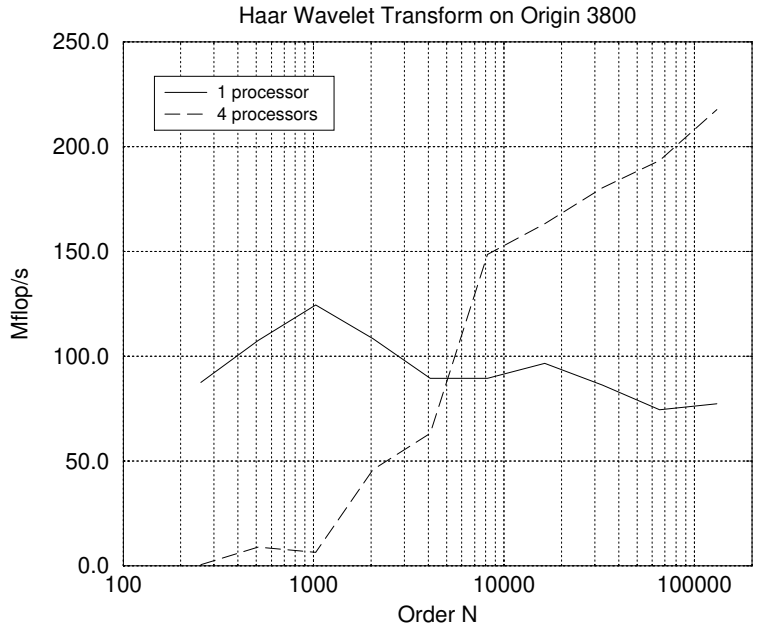

Figure 4.6: *Speed of 2-D Haar Wavelet transform with MPI. The results are obtained on the SGI Origin 3800 sequential and on 8 processors are shown.*

4.2.3 `mod2g`

The 2-D Haar Wavelet Transform is a good candidate for distributed memory parallelisation. The only communication needed is during the matrix transpose phase and this can be done at high speed. On the Itanium platforms compiling with optimisation level `-O3` results in wrong code and consequently in wrong results for some of the problem sizes. Probably, there is a problem in the synthesis routine `synthr.f` which is heavily hand-unrolled. When compiling with `-O2` the results are correct. The speed, however, is quite low in this case. In Figure 4.6 we therefore only show the results for the Origin 3800 with the sequential performance as a reference for the scaling.

The figure shows that for the smaller problem sizes the sequential performance is clearly better. This thanks to the fact that the amount of work in the distributed case is so low that the communication dominating the execution time. For larger problem sizes the situation changes for two reasons: first the computation per processor increases with a factor $2N \log N$ while communication only grows linearly. Second, for the large problem sizes the data cannot be held in cache anymore for the sequential code while the distributed parts of the data still easily fit in the cache in the parallel code. This results in a speed that is almost thrice as high as in the sequential case. The absolute floprate is quite low in both cases due to the low computational intensity of the algorithm.

4.2.4 `mod2i`

In program `mod2i` we considered the performance of a Quicksort algorithm. As a basis the same algorithm as in the sequential single-CPU test was used but in the parallel case first a distribution of subsequences over the processors is done by means of a limited amount of sample values (see [5] for the regular sampling technique). Once this is done the subsequences are sorted and some exchange of elements of subsequences is required followed by merging the non-local elements received to arrive at the sorted sequence distributed over the processors. The dispersal of the elements over the processors grows linear with N. This is the most time consuming communication phase. In the later phases communication is quite limited in comparison to the data manipulation to be done on each processor. One would therefore expect that the scaling behaviour is reasonable, though not perfect, due to the total amount of communication. In Figure 4.7 we show the speeds for the Itanium 2 reference platform and the Origin 3800. As in the sequential case the MIPS-based

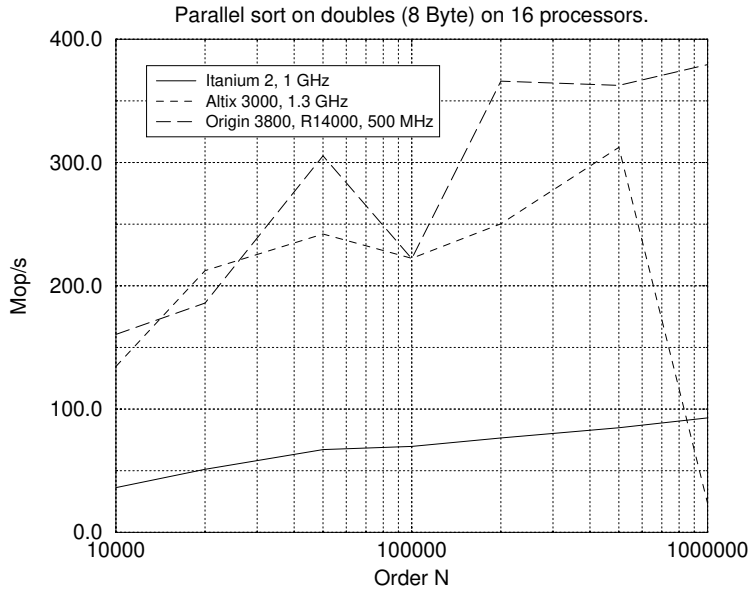

Figure 4.7: *Speed of sorting 8-byte floating-point numbers MPI. The results are obtained on Intel Itanium 2 reference platform and on the SGI Origin 3800 with 16 processors.*

system is clearly faster (see section 2.2.6) when sorting 8-byte floating-point numbers. The scaling behaviour of the three platforms is very different: the speedup on the Intel reference platform is only a factor or 2 at best, while it is 3–6 for both the Altix and the Origin system. The reason is not the available bandwidth, but rather the MPI implementation. On the Altix and Origin systems the SGI proprietary MPI implementation is used which is highly efficient. On the Intel Itanium 2 reference platform the Myrinet MPICH -GM 1.2.5 implementation was used in which some of the collective communication calls like `MPI_Alltoallv` are not implemented optimally.
Note the sharp decrease in performance on the Altix system for N =1,000,000. This is *not* an observational artefact. Repeated experiments show the same behaviour. For N = 1,000,000 integers this effect does not occur and with 2,000,000 8-byte floats the speed is again up to normal (296 Mop/s, not shown in Figure 4.7). This seems a quirk in the code that has to be investigated further.

5 Concluding remarks and summary

In the preceding sections we have looked into the behaviour of the Itanium 2 processor in combination with the Intel `efc` compiler to assess the performance both in a single-CPU setting and in multi-processing mode with OpenMP and MPI. In this report we tried to uncover the variety of performance patterns, the performance profile, that is pertinent to its use in a technical/scientific environment. The EuroBen benchmark was designed for this purpose and can point out specific strong and weak points of a processor platform for technical/scientific computing. The next step is then to identify the causes and, eventually, to come up with a remedy in case of unsatisfactory behaviour (if possible).

In this report we have concentrated on the first stage: getting to know how the processor behaves in various frequently occurring situations. This is one of the reasons this report is titled *"Preliminary benchmark results on an Itanium 2 reference platform"*. Another reason is that no full scale applications are reported here although some limited experiments have been done. Also, no use is made of the SGI SCSL and Intel MKL libraries that provide similar program building blocks as are used in the programs discussed here (e.g., BLAS3 and LAPACK routines). The reason is that many user programs implement the same algorithms but often in a non-standard way. In that case one still wants to know what a processor is typically capable of. It is an easy exercise to repeat some of the dense linear algebra type programs using either the vendor provided libraries or tuned versions from ATLAS [11, 1] or the Goto library.

We chose for this approach because it maps out real and potential problems with algorithms that are sufficiently simple to track down at least a good part of the reasons why programs behave as they do. Which is seldomly the case with full applications without reducing them again to the constituent algorithms. This is especially true for the I/O behaviour which is generally very elusive because of the large amount of configuration options and the interaction with the OS.

For a new platform like the Itanium/`efc` it is instructive to compare and contrast the findings with that of a system that has matured over several generations both with regard to the processor platform and the macro architecture. In this respect the SGI Origin 3800 with MIPS R14000 processors is a good choice. It brings out some of the differences between a dynamically scheduling RISC processor like the R14000 and and EPIC-based Itanium 2. We therefore have tried to provide results of both systems consistently.

5.1 Summary

To summarise the raw results as discussed in the previous sections we can do that in the following points:

1. For basic operations as tested in program `mod1ac` the performance ratio of the MIPS and the Itanium 2 systems is almost exactly the same as for the Theoretical Peak Performance of the respective processors, except for the 1^{st} and 2^{nd} order recurrence which are significantly slower on the Itanium 2 processors.

2. The precision of the mathematical intrinsic functions on the Itanium 2 processors is superior to that on the MIPS processor (and most other known processors, although results are not discussed here, see 2.1.3). The speed of the intrinsics is much higher for the Itanium 2 for the `Exp` and `Log`-related functions and for the `Sqrt` function. The normal goniometric functions are about 40% faster on the MIPS processor while the inverse goniometric functions and the hyperbolic functions are better by about 25% on the Itanium 2 processors (see 2.1.4).

3. For more involved basic algorithms, especially the dense linear algebra ones, the Itanium 2, 1 GHz, is 2–3 times faster than the MIPS processor (see 2.2.1, 2.2.2, and 2.2.3).
   For sparse of irregular data access the difference is usually smaller (a factor of 1.2–2.0, see 2.2.1, 2.2.4,

2.2.5, and 2.2.6). For the Quicksort algorithm `mod2i` also the number of integer units plays a role for integer sorts, while for sorting 8-byte floats the Itanium 2 is markedly slower than the MIPS processor.

4. In the mixed radix FFT program `mod2f` the radix-8 subroutine is not translated correctly by the `efc` compiler, resulting in incorrect outcomes for sequences in which this routine is required.

5. One simple I/O test was performed which determines the read/write bandwidth for direct access files in a sparse matrix-vector multiply setting. With version 7.0 of the `efc` compiler this gave rise to run time I/O errors and abortion of the program. This situation did not occur anymore with the 7.1 version of the compiler(see 2.3.1).

6. For OpenMP parallelism the parallelisation overhead in many cases is prohibitively large and spoils any speedup that is potentially present. This is true for both types of platforms but slightly more so on the Itanium 2 systems (see 3.1.1, 3.2.1).
   For the intrinsic functions the speedup on 4 processors is generally quite good as on the solution of a dense linear system for larger problem sizes (see 3.1.2, 3.2.2). Still, the amount of work per loop iteration is by no means a sufficient criterion for a good speedup as is shown by algorithms with more complicated memory access patterns (see 3.2.4 and 3.2.5). Generally speaking, OpenMP parallelisation is in many cases marginally useful on such a low level. One has to put in more than a few OpenMP directives to have a noticeable benefit of this facility.

7. Measurement of communication patterns on the Origin 3800 and the Altix 3000 show that the internal Numalink4 connections noticeably add to the observed maximum bandwidth, especially for point-to-point intra-node communication and for broadcast and collect operations on a limited number of nodes. For internode point-to-point communication and other communication patterns the Altix shows $\approx 20\%$ higher bandwidth (see 4.1.1).

8. Dense matrix-vector multiplication work with moderate success with MPI on the Itanium 2 and the MIPS systems. The potential on the Itanium 2 systems is undoubtedly higher than what is observed in the tests and should lead to higher efficiencies than the present 0.4–0.45 per processor (see 2.2.1 and 4.2.1).

9. For the MPI version of the 1-D FFT the speed ratio is consistent with that of the sequential version when comparing the Origin and Altix platforms: 1.4–2.0 depending on the problem size (see 4.2.2). For the other complicated memory access algorithms, the Haar Wavelet Transform and the Quicksort algorithm, no comparable result could be obtained on the Itanium platform because of compiler translation problems (see 4.2.3) for the former while the sorting algorithm gave a moderate speedup on the systems using SGI's MPI implementation but bad speedup with the less optimised MPICH-GM on the Intel Itanium 2 reference platform (see 4.2.4).

References

[1] Atlas webpage: `http://math-atlas.sourceforge.net`.

[2] EuroBen webpage: `www.euroben.nl`.

[3] R. W. Hockney, C. R. Jesshope, *Parallel Computers II*, Bristol: Adam Hilger, 1987.

[4] R.W. Hockney, $f_{1/2}$*: A parameter to characterize memory and communication bottlenecks*, Parallel Computing, **10** (1989) 277-286.

[5] X. Li, P. LU, J. Scheaffer, J. Shillington, P.S. Wong, *On the Versatility of Parallel Sorting by Regular Sampling*, Tecnical Report Dept. of Comp. Science, Univ. of Waterloo, Canada.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference Vol. 1, The MPI Core*, MIT Press, Boston, 1998.

[7] W. Gropp, S. Huss-Ledermann, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir *MPI: The Complete Reference, Vol. 2, The MPI Extensions*, MIT Press, Boston, 1998.

[8] Myrinet webpage: `http://www.myrinet.com`

[9] OpenMP Forum, *Fortran Language Specification, version 1.0*, Web page: `www.openmp.org/`, October 1997.

[10] A.J. van der Steen, *The benchmark of the EuroBen Group*, Parallel Computing **17** (1991) 1211–1221.

[11] R.C. Whaley, A. Petitet, J.J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Technical Report, Dept. of Computer Science, Univ. of Tennessee at Knoxville, September 2000.