Memory performance of dual-processor nodes: comparison of Intel Xeon and AMD Opteron memory subsystem architectures

Chona S. Guiang, Kent F. Milfeld, Avijit Purkayastha, and John R. Boisseau

Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX , USA 78712

Abstract. Dual-processor (2P) systems are the preferred building blocks in commodity Linux clusters because of their greater peak performance-to-price ratio relative to single-processor (1P) systems. However, running memory-intensive applications on 2P systems can lead to performance degradation if the memory subsystem cannot fulfill the increased demand from the second processor. Efficient utilization of the second processor depends on the scalability of the memory subsystem, especially for memory bandwidth-bound codes. This paper examines the performance of kernels and applications on two dual-processor systems with different memory subsystems: a shared bus architecture (Intel Xeon); and a NUMA-like architecture where each processor has an independent path to "local" memory, and a channel for coherence and inter-processor memory access (AMD Opteron).

Introduction

Computational systems that provide high memory bandwidth and scalability provide direct performance benefits for most HPC applications. Modest improvements in memory technology relative to burgeoning clock rates have made memory performance a crucial factor for improving application performance in an increasing percentage of scientific codes. Although vector architectures with custom memory subsystems (*e.g.*, NEC SX-6) provide the best performance in terms of memory bandwidth, the high cost associated with their development restricts their availability to only those centers with the deepest pockets. More economical systems based on commodity technologies use slower memory relative to the vector machines but employ caches to attempt to offset the imbalance between slow memory and fast processors. This works with varying degrees of success, but even the most cache-friendly codes cannot sustain streaming rates from memory as provided by traditional vector architectures. Although a wide variety of HPC benchmarks exist to indicate application performance, most do not give sufficient information to help analyze the key aspects that govern memory performance.

While cache sizes and memory hierarchy bandwidths are important, they are by no means the only useful metrics of memory performance. The latency incurred in accessing different levels of memory is crucial in cache-unfriendly applications involving gather/scatter operations, which are prevalent in sparse matrix calculations. Hardware support for data prefetching is available in most platforms as a means of hiding memory latency. The detection and implementation of data prefetch streams varies with the platform and compiler; nevertheless, every effort should be used to take advantage of any potential increase in memory bandwidth, and offset latency.

In this study, we compare the memory architectures of two different dual-processor commodity systems. All kernels and applications were run on a Dell 2650 node with dual Intel Xeon processors and on a Newisys dual-processor system based on the new AMD Opteron. In section I, we discuss the differences between the two systems, focusing on details of the memory subsystem. Measurements and results are presented in section II, and we conclude with a summary of our observations and recommendations in section III.

I. Memory architectures of dual-processor systems

Shared bus microarchitecture

The architecture of commodity multiprocessor (MP) PC systems uses an off-chip Northbridge to interconnect the processors to memory, as illustrated in Figure 1. The Northbridge acts as a hub to graphic devices, to the memory DIMMs, and to I/O devices via a Southbridge. A common server configuration for systems based on Intel IA-32 processors incorporates two processors on a single motherboard that shares a common Northbridge and memory DIMMS. Both processors must share the 400 MHz front-side bus (FSB), which provides 3.2 GB/s bandwidth, in our example system in Figure 1.

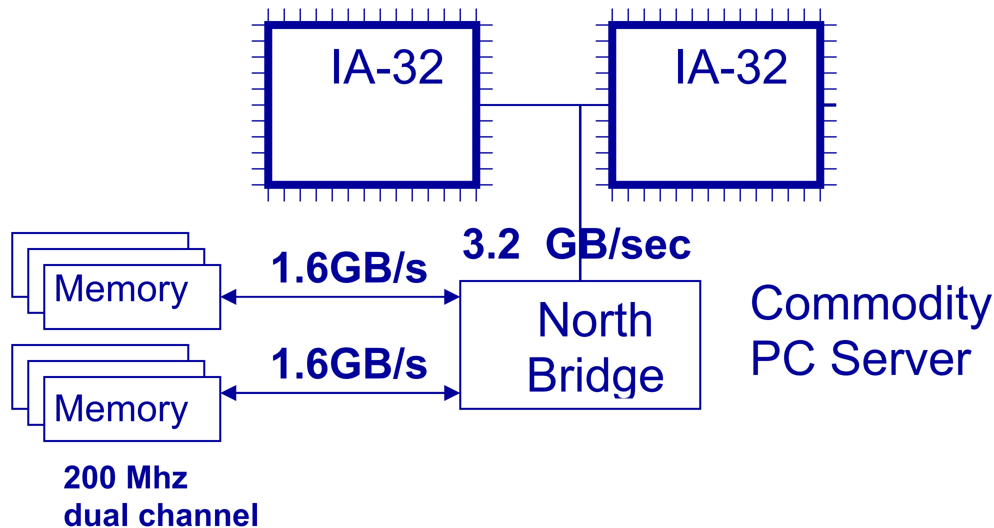

Fig. 1. Memory subsystem, dual-processor PC Server (400 MHz FSB and dual channel 200 MHz memory).

Scalable memory interconnect

AMD has developed a processor [1] and chipset architecture [2] that scales in memory bandwidth and I/O support as more processors are configured in the system. The latter can be important for high-speed interprocessor interconnects that use multiple adapters to increase bandwidths to the switch. Memory access is optimized in two ways. First, controllers are built into the chips, as shown in Figure 2, to create a direct path to each processor's locally attached memory. Hence, the aggregate memory bandwidth is additive, as long as data is accessed locally. Scaling in the memory coherence is achieved through a virtual chipset, interconnected via a high speed transport layer, called HyperTransport [3]. Each processor unit can share its local memory and devices with other processors through HyperTransport links; but each processor retains a local, direct path to its attached memory. This system is NUMA-like in architecture because of the difference in latencies when accessing local versus remote memory. A broadcast cache coherence protocol is used to avoid directory-based serialization delays and to allow snooping to overlap DRAM access. Figure 3 provides the basic characteristics of HyperTransport connections.

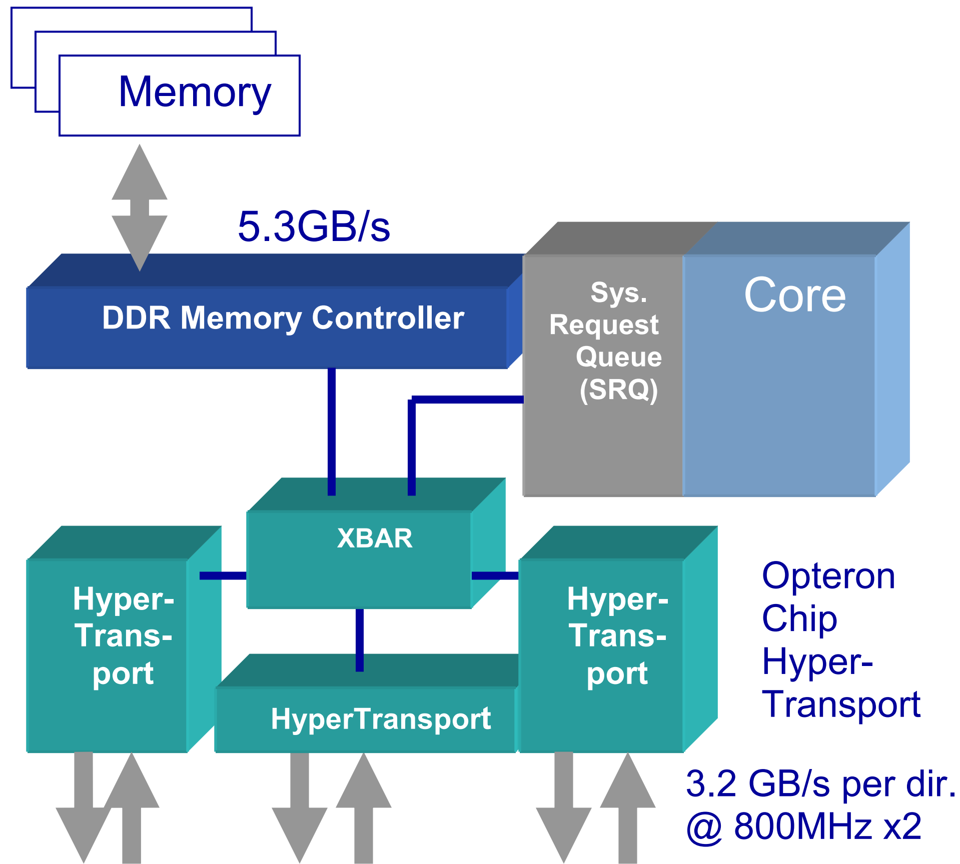

Fig. 2. HyperTransport technology within Opteron microarchitecture

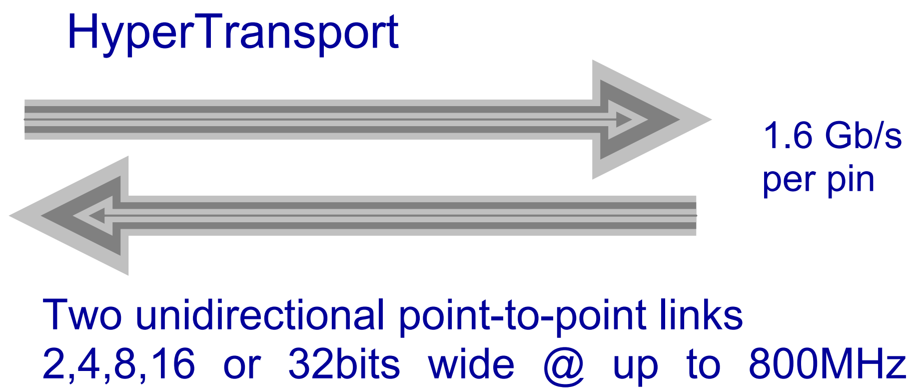

Fig. 3. HyperTransport link widths and speeds [3].

In a dual-processor AMD Opteron system, two of the three available HyperTransport (HT) links are used on one processor: one to connect to external devices (I/O, PCI, etc.) and the other for transaction ordering and coherence with the other processor. The other processor uses a single HT link. In the system illustrated in Figure 4, both processors share access to external devices through the "first" processor; however, an additional link could be used on the second processor to duplicate (scale up) external interfaces.

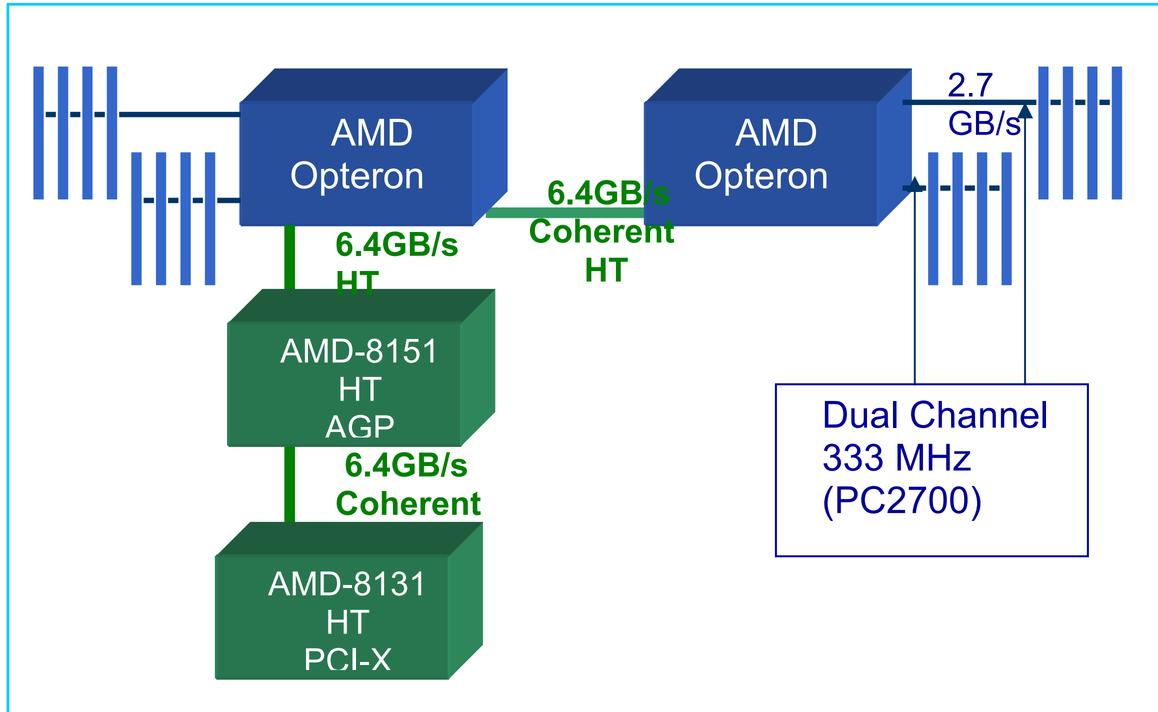

Fig. 4. Dual-processor Opteron System memory layout.

The interprocessor HT interconnects effectively act like a distributed switch, providing coherent, memory (bandwidth) scaling with multiple processors.

II. Measurements and Results

Our numerical measurements consist of memory intensive kernels and simple applications that were run on both Intel and AMD dual-processor systems. Unless otherwise indicated, all data points represent an average of multiple measurements to eliminate system "noise." The Intel system used was a Dell 2650 node [4] with two 2.4 GHz Intel Xeon processors [5] sharing 2 GB of memory. The AMD system was a dual-processor node from Newisys [6] featuring two 1.4 GHz AMD Opteron processors and a total of 2 GB of memory.

Our memory performance kernels carry out latency and bandwidth measurements from all levels of the memory hierarchy. We also determine the scalability of common mathematical calculations in going from one to two application processes per node. Because memory latency is important in applications

which involve non-unit stride, we examine the performance of random and strided memory access, as well as the ability of compilers to detect prefetch data streams in order to hide the latency involved in fetching data from main memory.

Our application benchmarks include matrix-matrix multiplication, molecular dynamics and finite difference codes parallelized using MPI. We also present NAS Parallel Benchmarks [7] results on both the AMD and Intel systems.

Latency from different levels of memory

To measure the latency of accessing data at various levels of memory, we devised the following simple kernel:

```
index=ia(1)
do i=1,n
   num=ia(index)
   index=num
end do
```

The `integer*4` array `ia(n)` contains the integers `1` to `n` in random order, with the restriction that following the array in the order shown above will access all elements of the array. Hence, integer data will be loaded from the cache or memory in a non-sequential (random) order. Because each iteration depends on the previous one, the time to complete one iteration depends on the latency in accessing the element of array `ia`.

To remove the clock speed dependency of the Intel and AMD processors, we report our results in clock periods per iteration. The results of our latency measurements (using only one processor) are shown in Figure 5 (AMD Opteron) and Figure 6 (Intel Xeon) :

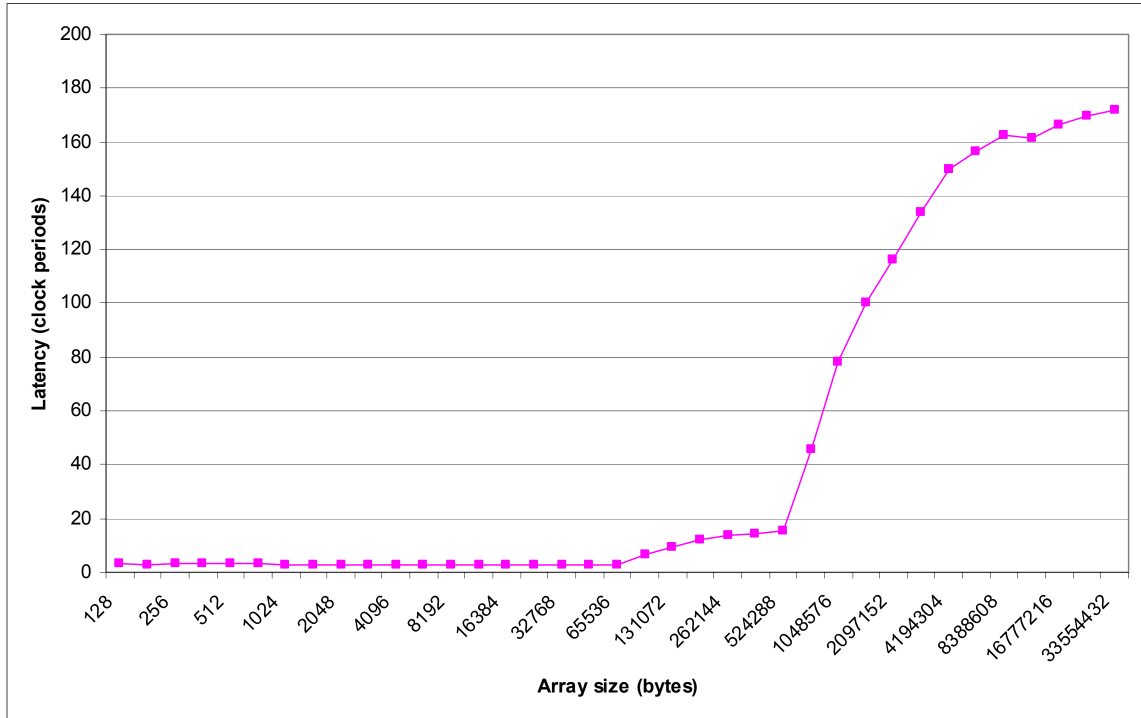

Fig. 5. Plot of latency (measured in clock periods) vs. the extent of the integer array **ia** in bytes for the AMD Opteron system.

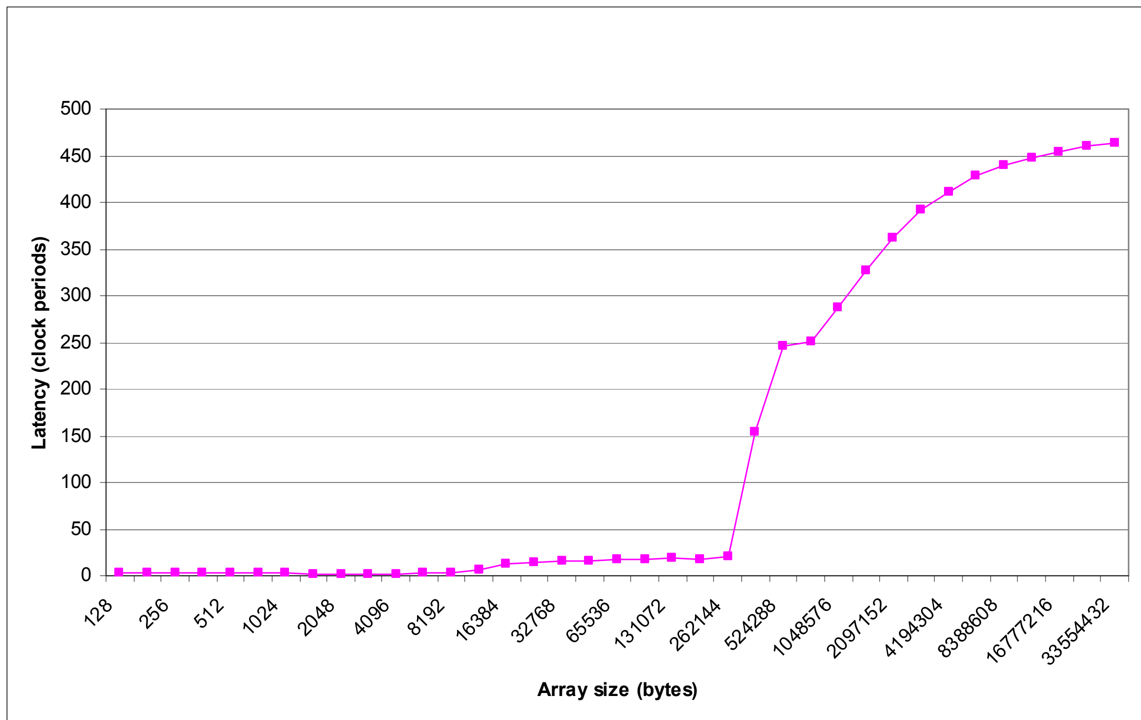

Fig. 6. Plot of latency (measured in clock periods) vs. the extent of the integer array `ia` in bytes for the Intel Xeon system.

Both systems exhibit low latencies when the data fits in cache. Fetching one 4-byte word of data from the L1 data cache takes ~2-3 clock periods for the Intel Xeon and ~3 clock periods for the AMD Opteron. From the two figures, it is easy to see the large penalties associated with accessing data from main memory for both processors and memory architectures. In both systems, the latency jumps to large values when the integer array overflows the 512 KB and 1 MB L2 caches of the Intel Xeon and AMD processors, respectively. However, the latency for the Intel Xeon to access data in main memory is more than twice as many clock periods as for the Opteron.

Bandwidth from different levels of memory

To measure the bandwidth achieved from performing loads from different levels of memory, we ran the following kernel on both the Intel Xeon and AMD Opteron systems:

```
do i=1,n
   s=s+a(i)
   t=t+b(i)
end do
```

Results for serial and dual-processor runs are given in Figures 7 and 8 below. The parallel version involves concurrent execution of two independent instances of the kernel shown above.

For serial execution of the read bandwidth kernel, the bandwidth from cache to the CPUs on the Intel Xeon system is 13 GB/s, while that measured on the AMD Opteron is 11 GB/s. At large vector lengths, both systems deliver 2 GB/s of bandwidth from main memory to a single CPU.

When two parallel read processes are executed, the AMD Opteron bandwidths scale very well even when the data is larger than cache. At large vector lengths, each processor performing a parallel read obtains 2 GB/s of sustained memory bandwidth. The dual-processor reads on the Intel Xeon system however, delivers a memory bandwidth of 1 GB/s to each processor. For the entire range of vector lengths outside of cache, the memory bandwidth delivered to each processor is half of what is obtained in a serial run.

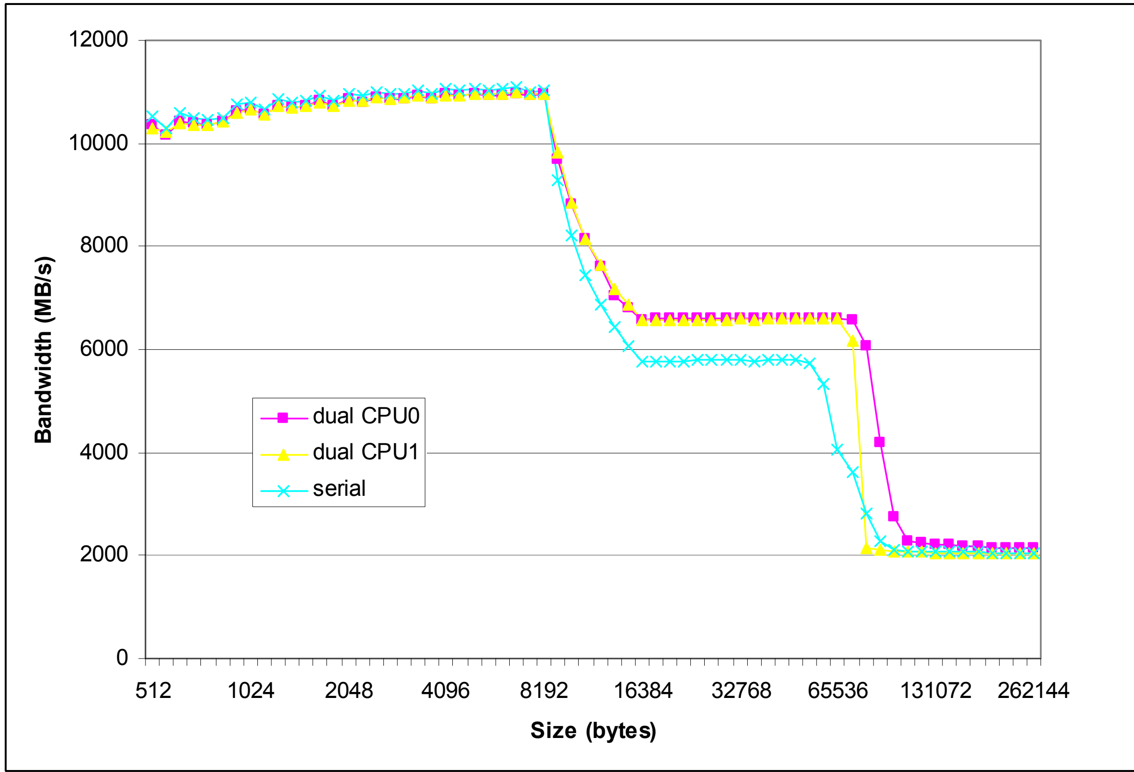

Fig. 7. Read bandwidth on the AMD Opteron system

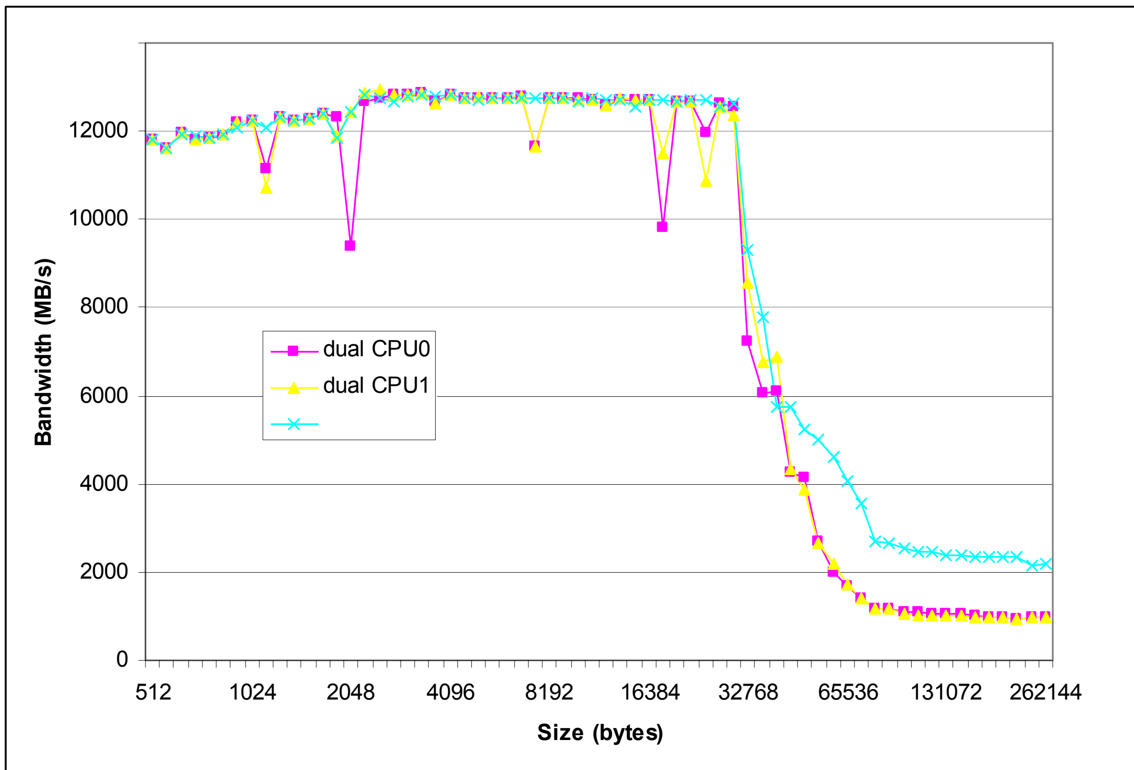

Fig. 8. Read bandwidth on the Intel Xeon system.

We ran the STREAM [8] benchmark on both dual-processor systems to measure sustainable bandwidth from memory for different combinations of memory reads and writes. On each system, different STREAM binaries were generated using different combinations of compiler options. The results below represent the best set of STREAM measurements obtained for each system (bandwidth numbers are in MB/s):

Table 1. STREAM on the Intel Xeon and AMD Opteron systems, serial execution.

Kernel	Intel Xeon	AMD Opteron
Copy	1125	2162
Scale	1117	2093
Add	1261	2341
Triad	1263	2411

Table 2. STREAM on the Intel Xeon and AMD Opteron systems, executed in parallel on two threads.

Kernel	Intel Xeon	AMD Opteron
Copy	1105	3934
Scale	1103	4087
Add	1263	4561
Triad	1282	4529

For both serial and multithreaded execution, the Opteron system shows higher sustained memory bandwidth for all the STREAM kernels. Furthermore, there is excellent scaling of memory bandwidth in this system as the number of execution threads is doubled. On the Intel Xeon node however, the bandwidth is practically unchanged when both processors execute the benchmarks in parallel. This is not unexpected for a shared-bus memory architecture, since the memory demands of the STREAM kernels are large enough to saturate the frontside bus even for the serial case.

The parallel STREAM numbers underscores how much of the available memory bandwidth of both systems is realized when streaming in data from memory. The sustained aggregate memory bandwidths for the Opteron system represents anywhere from 37-42% of the peak, while the numbers for Intel indicate that 35-40% of the peak bandwidth of 3.2 GB/s is obtainable.

Prefetch Streams

Enabling prefetch streaming of data from memory is important because it increases the effective memory bandwidth as well as hides the latency of access to main memory. Prefetch streams are detected and initiated by the hardware after consecutive cache line misses. Although this requires no directives in the code, restructuring memory accesses to expose prefetch opportunities can benefit many calculations. Our prefetch kernel measures the performance of a dot-product by using loop bisection, trisection, etc. to expose multiple data streams to the prefetch engine:

Loop 1: (two data streams)

```
do i=1,n
   s=s+x(i)*y(i)
end do
```

Loop 2: (four data streams)

```
do i=1,n/2
   s0=s0+x(i)*y(i)
   s1=s1+x(i+n/2)*y(i+n/2)
end do
if (mod(n,2).ne.0) s0=s0+x(n)*y(n)    ! clean up code for odd n
s=s0+s1
```

Loop 3: (six data streams)

```
do i=1,n/3
   s0=s0+x(i)*y(i)
   s1=s1+x(i+n/3)*y(i+n/3)
   s2=s2+x(i+2*n/3)*y(i+2*n/3)
end do
do i=3*n/3+1,n      ! clean-up code for mod(n,3)!= 0
   s0=s0+x(i)*y(i)
end do
s=s0+s1+s2
```

Loop 4: (8 data streams)

```
do i=1,n/4
   s0=s0+x(i)*y(i)
   s1=s1+x(i+n/4)*y(i+n/4)
   s2=s2+x(i+2*n/4)*y(i+2*n/4)
   s3=s3+x(i+3*n/4)*y(i+3*n/4)
end do
do i=4*n/4+1,n      ! clean up code for mod(n,4)!= 0
   s0=s0+x(i)*y(i)
end do
```

Measurement of the 2-, 4-, 6-, and 8-stream dot-product memory bandwidths are given below. Note that in all cases, the vector lengths are larger than the cache.

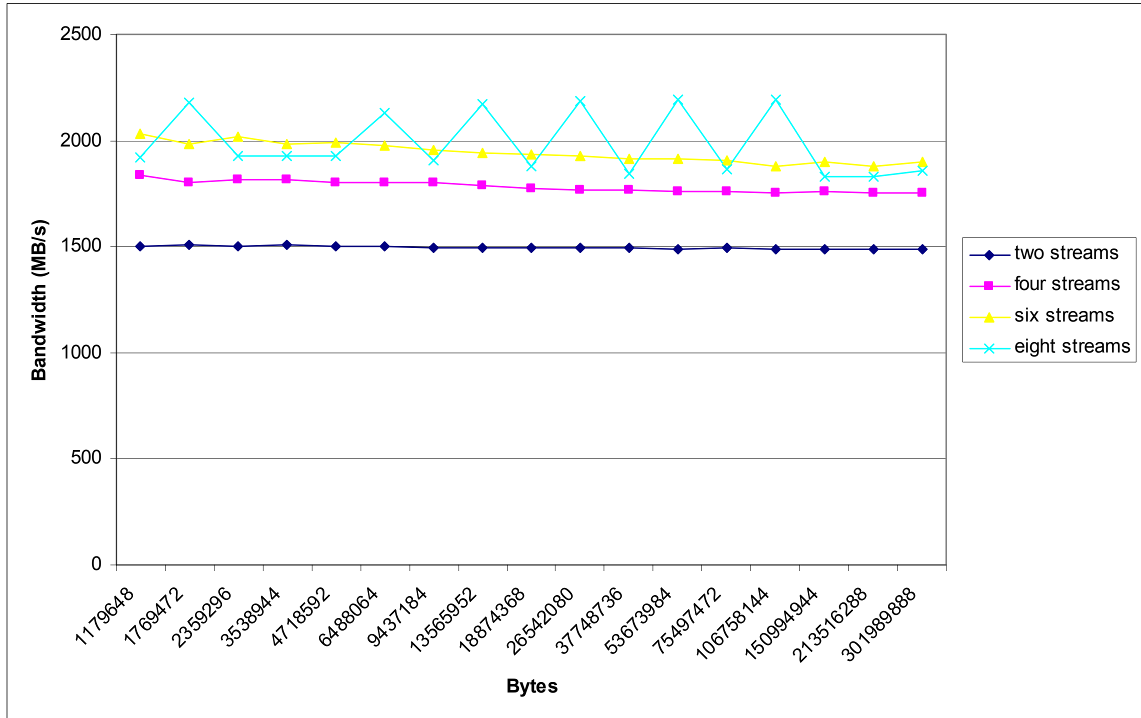

Fig. 9. Dot-product streams bandwidths for the AMD Opteron dual-processor system

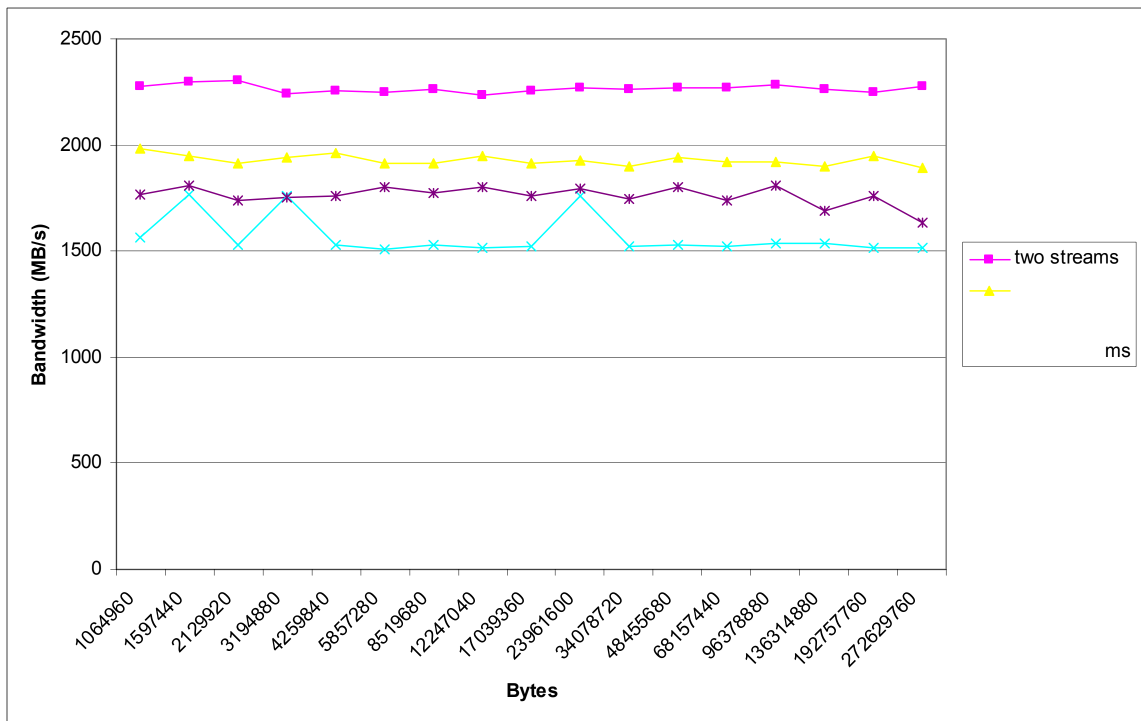

Fig. 10. Dot-product streams bandwidths for the Intel Xeon dual-processor system.

In a shared-bus memory architecture, a serial process can effectively take advantage of the entire memory bandwidth supported by the system bus.

For the Intel Xeon system, the highest bandwidth is obtained with the original dot-product loop. On the other hand, loop bisection, trisection and quadrisection appears to benefit the performance of the dot-product kernel.

Serial and parallel dot-product (dot-product)

Each iteration in a dot-product loop contains two memory loads and two floating point operations. We exclude the scalar sum variable since this remains in a register for the entire duration of the loop. Calculation of the dot product becomes memory bound as the vector length is increased. On the other hand, the dot-product is cache efficient since all the data in a cache line are used, although there is no reuse of data. The results of carrying out the dot-product (sans bisection, etc.) are displayed below. Both graphs contain results for serial and parallel runs executed using two MPI tasks. The parallel version of the code involves simultaneous execution of two MPI tasks, each performing a dot-product on its set of x and y vectors. Again, we normalize the timing results by presenting it in units of clock cycles:

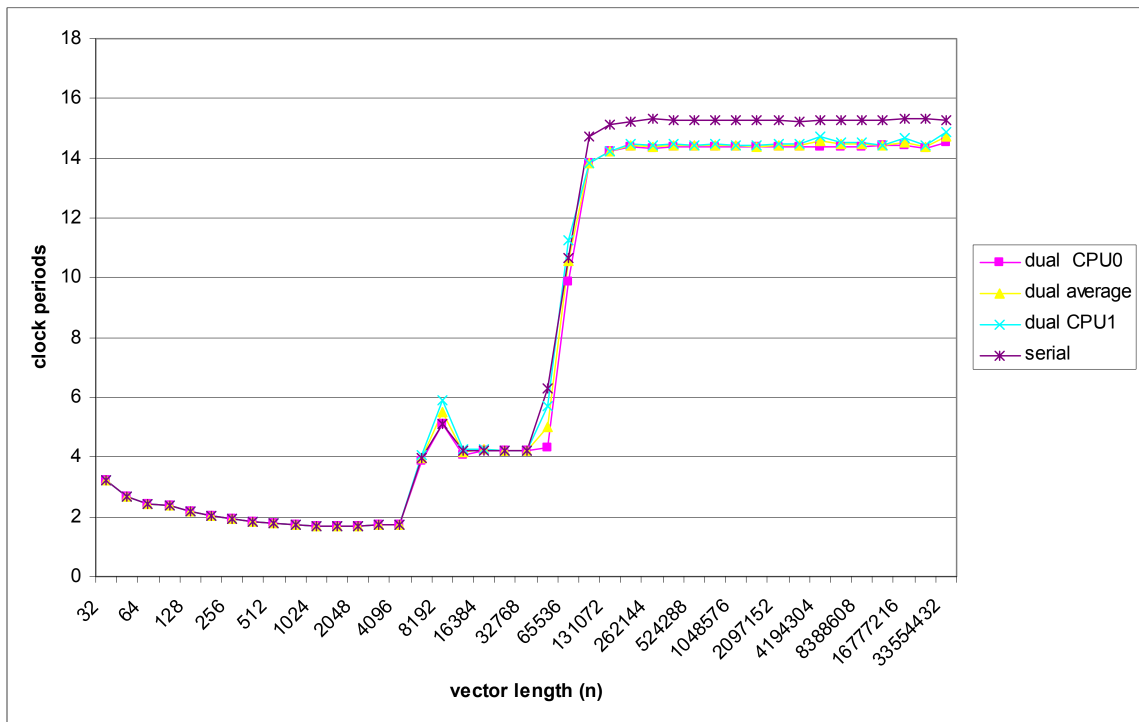

Fig. 11. Execution time per iteration of a dot-product loop performed on the AMD Opteron system.

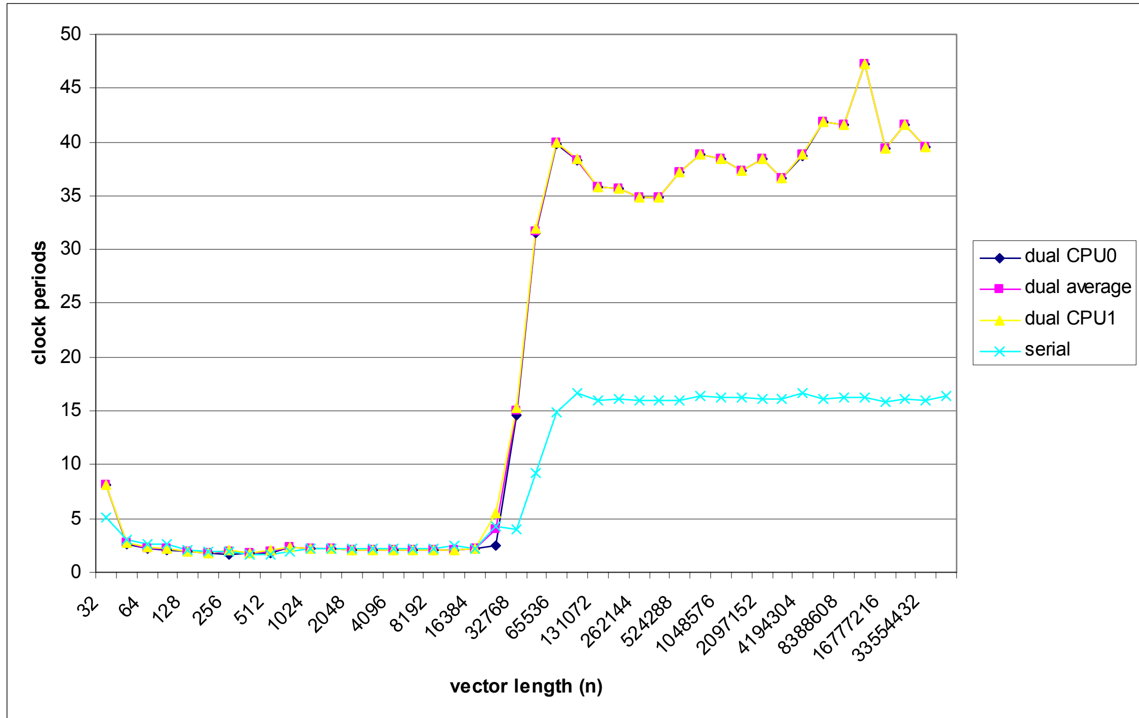

Fig. 12. Execution time per iteration of a dot-product loop performed on the Intel Xeon system.

Calculation of the dot-product takes 2-3 clock cycles on both systems when the `x` and `y` vectors fit in cache. Important differences can be observed in the AMD vs. Intel timing numbers when the vectors spill out of cache, It takes an average of 16 cycles to complete a dot-product on the AMD system once the vectors are in main memory. This execution time does not change as the number of application processes is doubled. This scalability of memory bandwidth makes sense in light of the fact that each processor on the Opteron system has an independent path to memory.

On the Intel system, the completion time for one iteration of a serial run takes about the same time as that on the AMD, but the average completion time more than doubles once two instances of the dot-product loop is run on the node. Because the bandwidth is split among the two running processes, the average execution time of each iteration is doubled.

Matrix-vector multiplication (DAXPY) loop

We implement the general matrix-vector multiply kernel as follows:

```
do j=1,n
   do i=1,n
      y(i)=y(i)+A(i,j)*x(j)
   end do
end do
```

This is to ensure that the elements of vector `y` and the `nxn` matrix `A` are accessed sequentially, while `x(j)` is register-resident in the inner loop. The DAXPY loop was run in serial and parallel modes on both the Intel Xeon and AMD Opteron systems. As before, parallel execution involves two MPI tasks, each running a DAXPY loop. The results of our measurements are shown below:

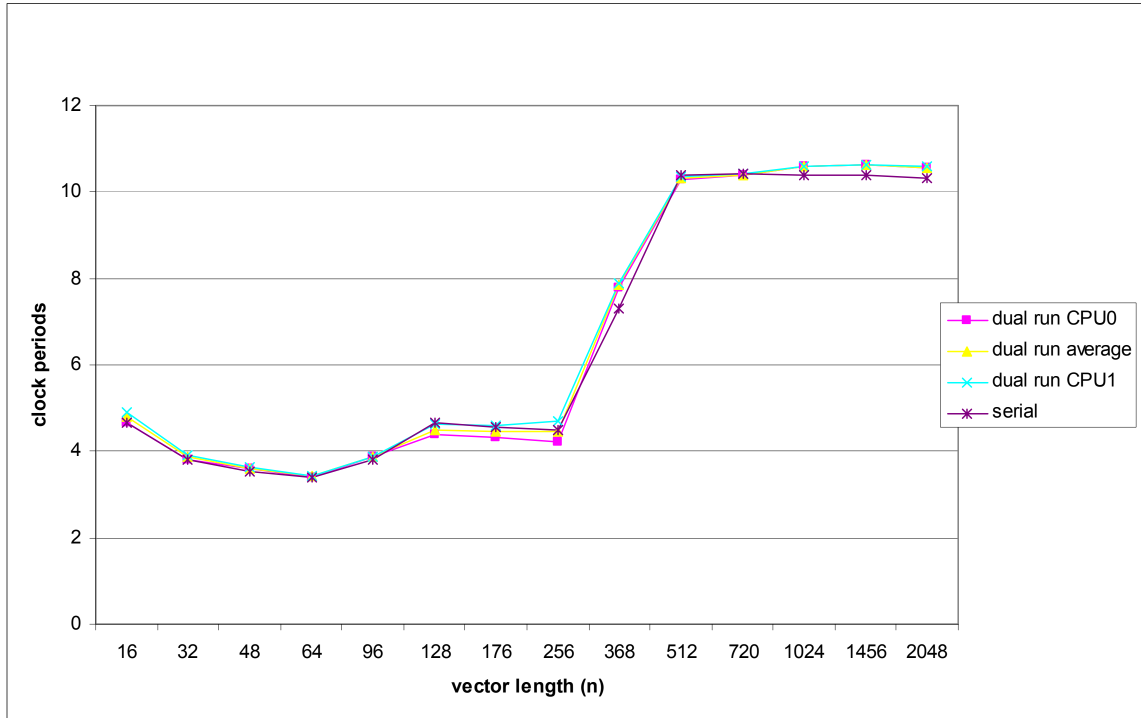

Fig. 13. Execution time per iteration for a DAXPY loop run serially and in parallel (using two MPI tasks) on an AMD Opteron system.

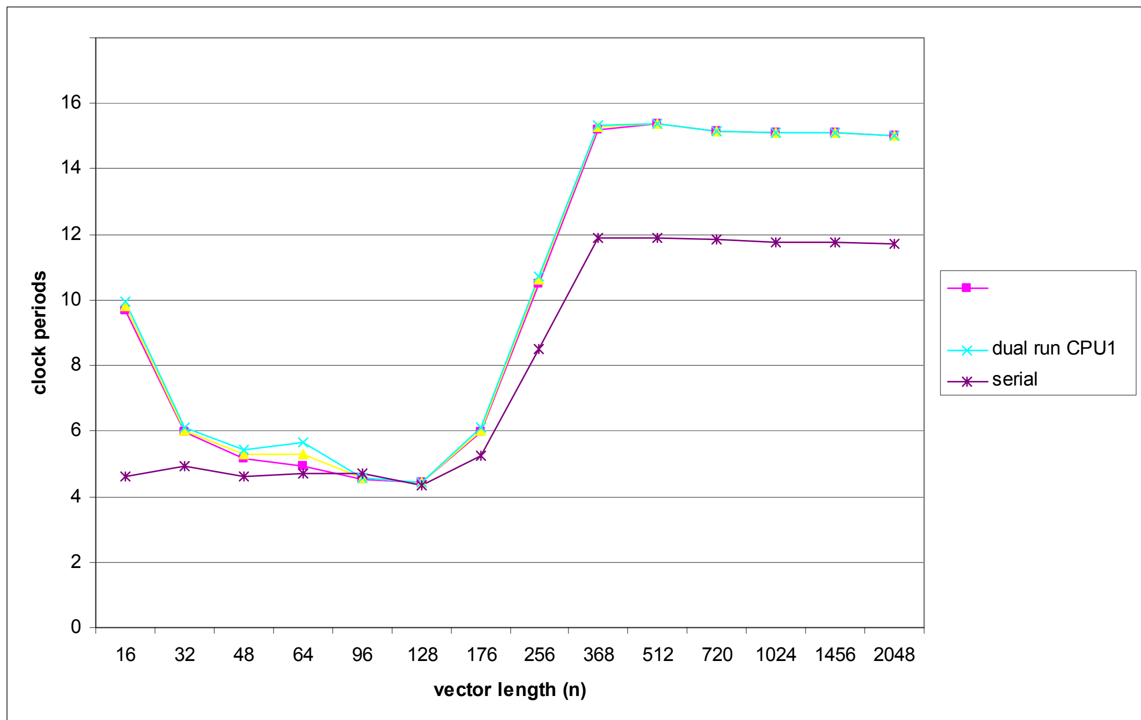

Fig. 14. Execution time per iteration for a DAXPY loop run serially and in parallel (using two MPI tasks) on an Intel Xeon system.

The average number of clock cycles for the completion of one serial DAXPY iteration on the Intel system (~4-5) is fairly close to that observed on the AMD (~4). Of course, this translates to greater throughput on the Intel Xeon processor because its clock speed is almost twice the clock speed of the AMD Opteron.

In going from serial to dual-processor execution, the time to complete each DAXPY iteration on the AMD remains fairly constant. This is consistent with the previous results for the dot-product calculation, and can be explained in terms of independent data streams from memory to each processor on a node. However, this is not the case for the Intel Xeon system where the bandwidth is split among the two processors. DAXPY performance is halved in going from serial to parallel execution. However, this degradation is not as severe as that observed in the dot-product calculation. This is to be expected since DAXPY is more computationally intensive than dot-product; the ratio of memory references to floating point operations is lower in DAXPY (~1/2) than dot-product (1).

Naïve matrix-matrix multiply

Although most users would not implement matrix-matrix multiplication using the simple 'textbook' algorithm, our matrix-matrix multiplication kernel was included to measure the response of the memory system to calculations involving unit and non-unit stride access.

Each iteration in a matrix-matrix multiplication calculation contains three loads and one store, with one of the memory reads being non-sequential, and a total of two floating point operations. The matrix-matrix multiplication kernel was run serially and in parallel, using two MPI tasks performing independent matrix-matrix calculations. The results the serial and parallel runs are shown below:

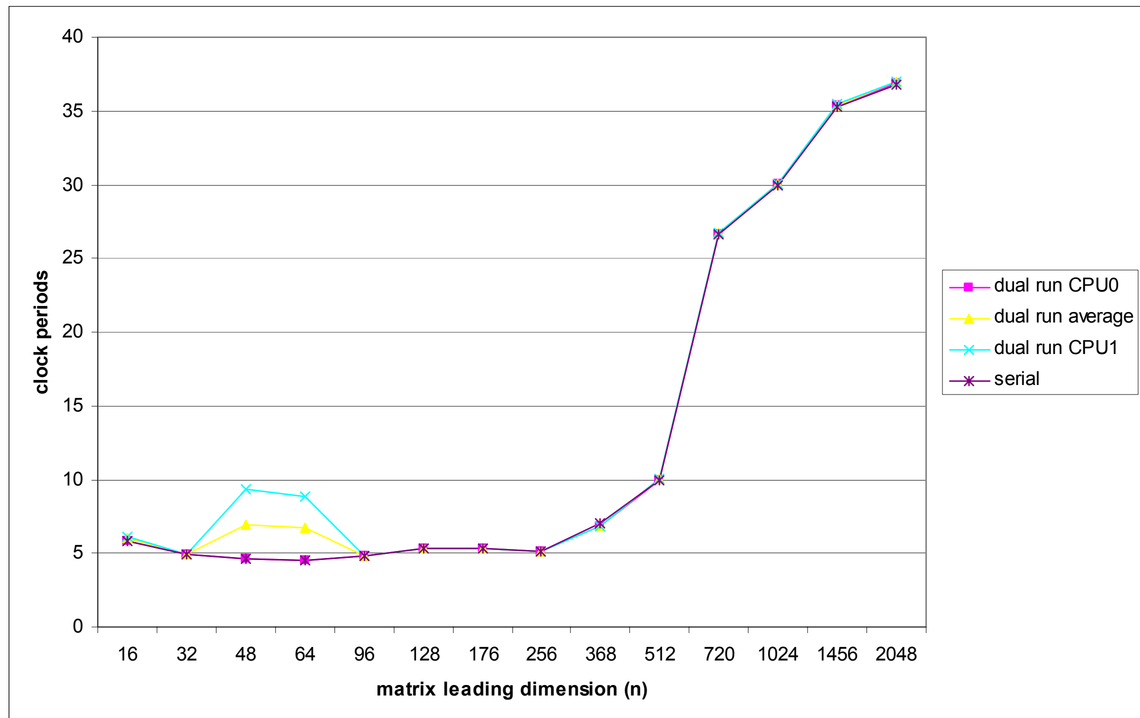

Fig. 15. Execution time per iteration for a matrix-matrix multiplication loop run serially and in parallel (using two MPI tasks) on an AMD Opteron system.

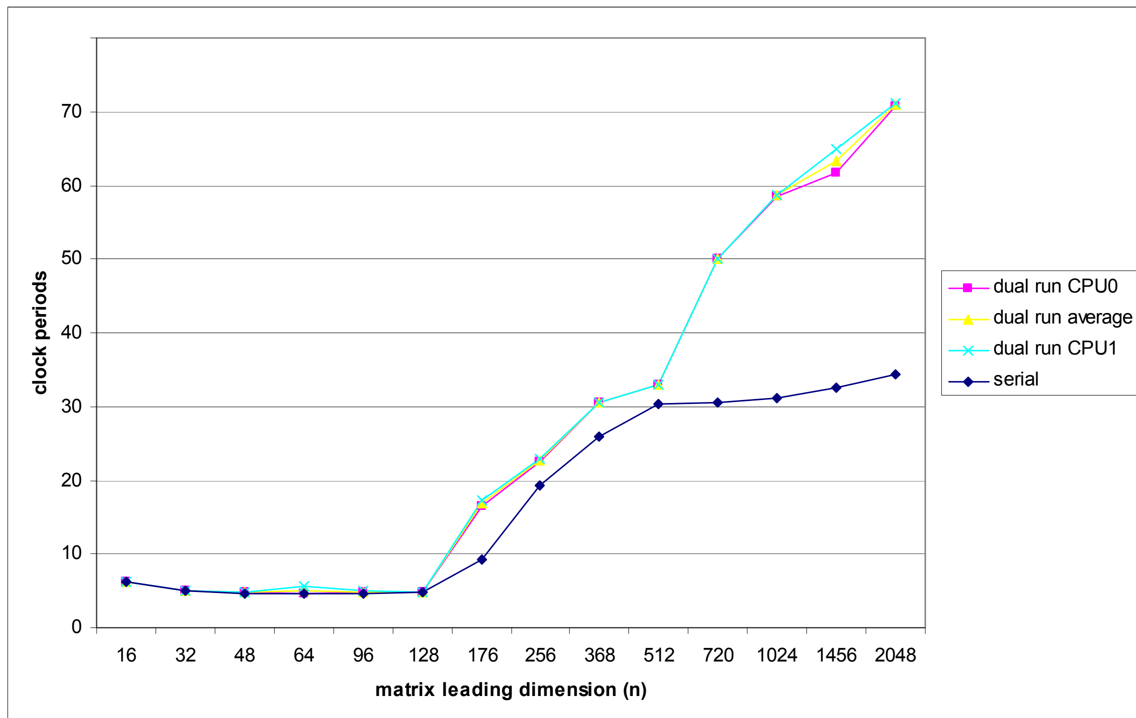

Fig. 16. Execution time per iteration for a matrix-matrix multiplication loop run serially and in parallel (using two MPI tasks) on an Intel Xeon system.

For serial execution, the number of cycles required to complete one iteration of a matrix-matrix multiply is about the same on the Intel and AMD systems (~5 cycles) when the matrices fit in cache. On both systems, there is a jump to 35 cycles per iteration for serial runs when the data spills out of cache.

Parallel execution of the matrix-matrix multiplication on the Intel Xeon node takes twice as long per iteration as the corresponding serial run. On the AMD Opteron system, the time per iteration for parallel matrix-matrix multiplication is the same as that obtained for the serial run.

Indirect dot-product

In an indirect dot-product, one of the vectors is indirectly addressed using an index array:

```
do i=1,n
   s=s+x(i)*y(ind(i))
end do
```

This kernel forms the basis for sparse matrix-vector multiplication. In our tests, the integer*4 array `ind` is a random re-indexing of the integers `1` to `n`. Our results in Figures 17 and 18 show higher clock cycles for the completion of one iteration relative to dot-product, whether the data is in cache or in main memory. This is not unexpected since there are more memory references per iteration in indirect dot-product than in ordinary dot-product (three memory loads versus two). Furthermore, there are no hindrances to compiler-initiated loop unrolling and vectorization of the dot-product loop, whereas moderate to aggressive vectorization of the indirect dot-product loop is inhibited by the use of indirect addressing. Thirdly, the use of an index array to address the elements of vector `y` does not trigger prefetching of these vector elements, whereas two independent data streams can be activated for a simple dot-product loop.

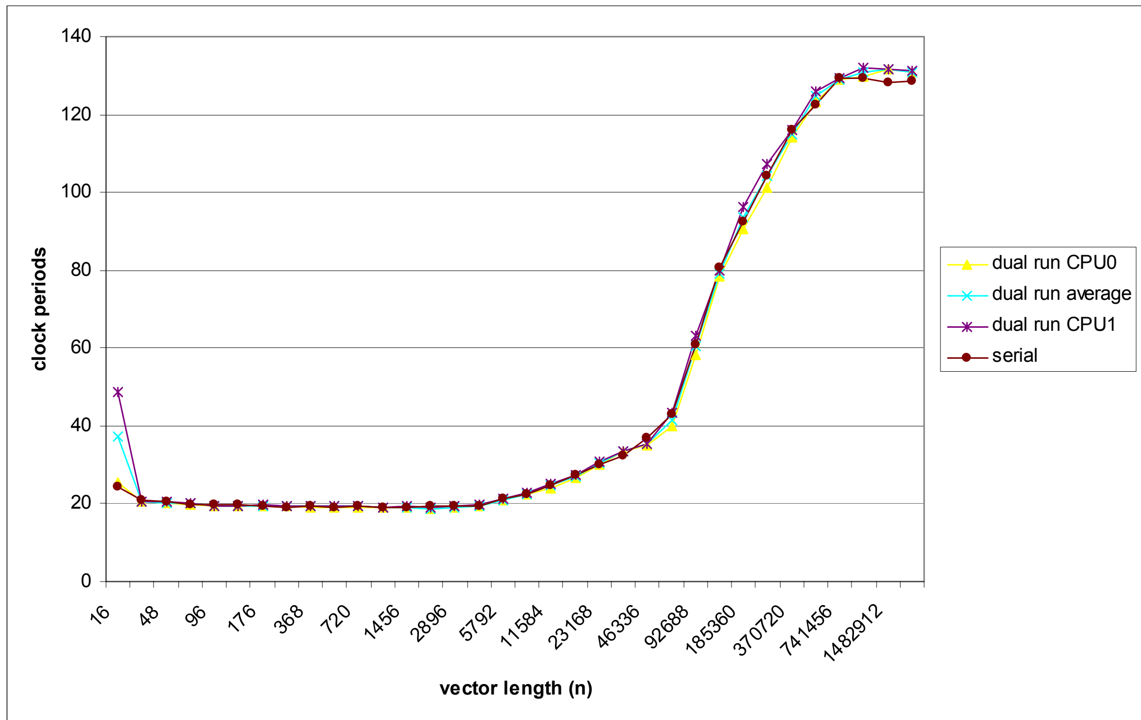

Fig. 17. Time per iteration of an indirect dot-product loop executed serially and in parallel using two processors on the AMD Opteron system.

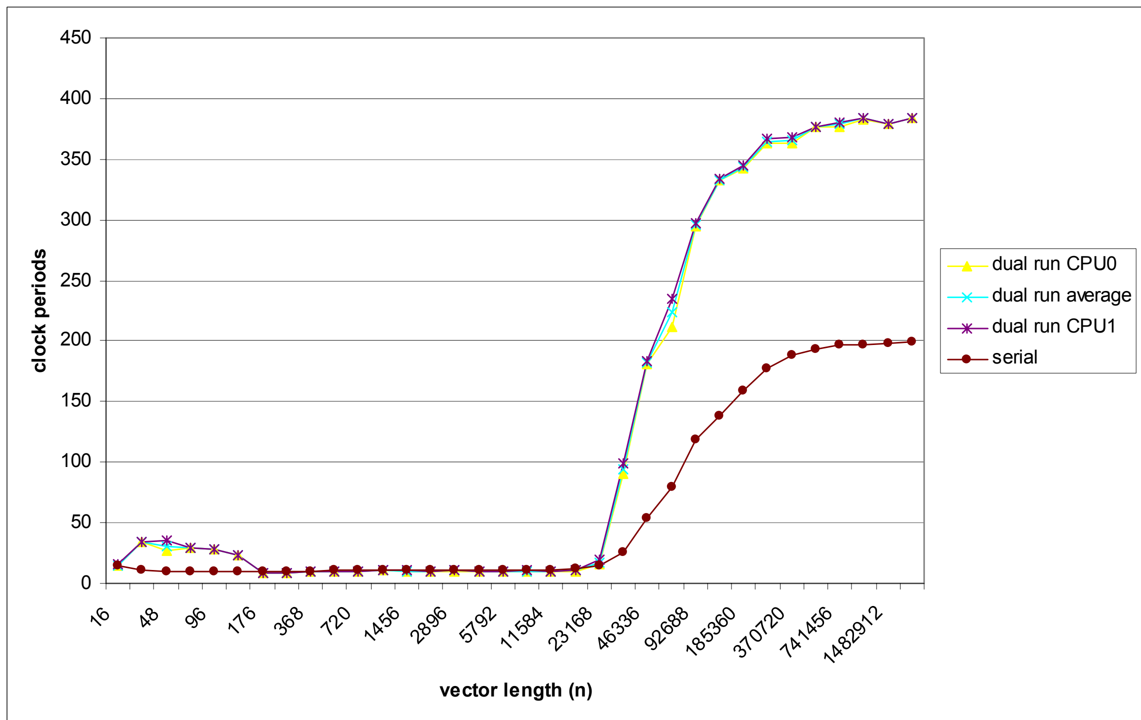

Fig. 18. Time per iteration of an indirect dot-product loop executed serially and in parallel using twop processors on the Intel Xeon system.

When running the indirect dot-product serially, the Intel Xeon system takes less time to complete one loop iteration than the AMD Opteron when the data is cache resident. But the indirect-dot product on the AMD is faster when data is in main memory.

Performing the indirect dot-product in parallel scales very well on the AMD system, but leads to degraded performance when run on the Intel node. When data overreaches cache, each iteration of the parallel indirect dot product calculation takes twice as much time to complete relative to the serial case.

Applications

We assembled what we hope is a comprehensive and representative list of applications to benchmark both the Intel Xeon and AMD Opteron 2P systems, and these include codes which perform the following types of calculations: molecular dynamics, solution of a system of partial differential equations using finite difference, optimized matrix-matrix multiplication, and the NPB benchmark suite. For each test, we use aggressive compiler optimization options to generate the program binaries.

1. Molecular Dynamics

We calculate the dynamics of a solid argon lattice for one picosecond, using the velocity Verlet method to advance the particle positions and velocities for each time step. The code is parallelized by distributing the particles among the two MPI tasks. Table 3 summarizes the results for serial and parallel runs on both Intel Xeon and AMD Opteron systems when the simulation is carried out on a lattice with 256 argon atoms.

Table 3. Application performance for serial and parallel MD simulations on the Intel Xeon and AMD Opteron 2P systems.

Platform	Serial MD	Parallel MD
AMD Opteron 2P	9.48	5.3
Intel Xeon 2P	7.14	5.4

The Intel Xeon system outperformed the AMD Opteron for the serial MD runs, as one would expect for a floating-point intensive application on a higher clock-speed machine. The AMD result for the parallel simulation shows greater scalability than that for Intel --- 1.79 vs. 1.3. This demonstrates that memory bandwidth still plays a big part in improving code performance even for applications that are compute bound.

2. Finite difference

The Stommel model of ocean circulation is represented mathematically by a 2D partial differential equation. We find the solution over a discretized domain, approximate the derivatives using finite difference, and perform a specified number of Jacobi iterations. The code is parallelized using domain decomposition, where each subdomain contains ghost cells for storing neighboring cells. We present the results in Table 4 below:

Table 4. Application performance for serial and dual-processor runs of the Stommel finite difference code on the Intel Xeon and AMD Opteron 2P systems. The domain size is 500x500.

Platform	Serial Stommel	Parallel Stommel
AMD Opteron 2P	68.0	43.4
Intel Xeon 2P	57.5	63.4

The serial performance on the Intel Xeon system is better than that on AMD. However, the parallel performance on this system is worse than that for serial execution. Although the AMD serial performance is relatively poor compared to that on the Intel Xeon, parallel Stommel exhibits great scalability on this system. For the dual-processor Stommel simulation, the execution time is the same on both systems. Again, we see the importance of memory bandwidth in keeping both CPUs supplied with data. Calculation of the partial derivatives involves strided memory access, and hence the poor performance and scalability of the finite difference code on the Intel Xeon system is consistent with what we observed for calculations involving noncontiguous memory access patterns (see results for naïve matrix-matrix multiplication).

3. Matrix-matrix multiplication

The performance of matrix-matrix multiplication is of importance to many scientific calculations, since a number of linear algebra calculations are dependent on this operation. Figures 19 and 20 show the effective memory bandwidth obtained when the Intel MKL DGEMM routine is run serially and when two MPI tasks, each running DGEMM, are executed on the Intel Xeon and AMD Opteron systems.

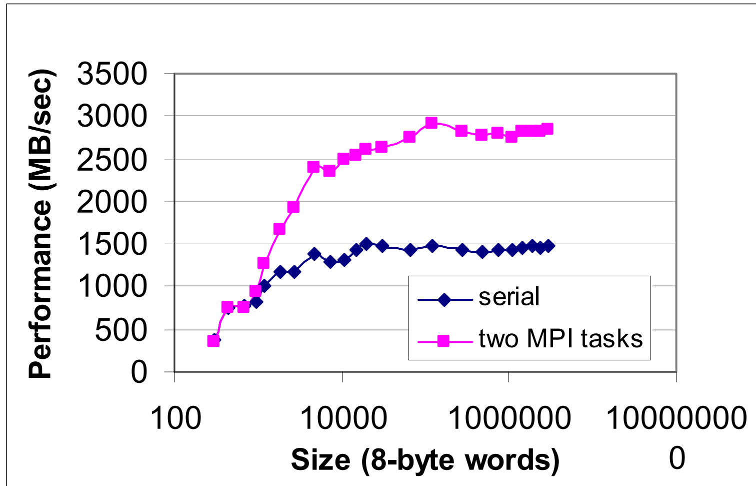

Fig. 19. Serial and parallel DGEMM performance on the AMD Opteron system.

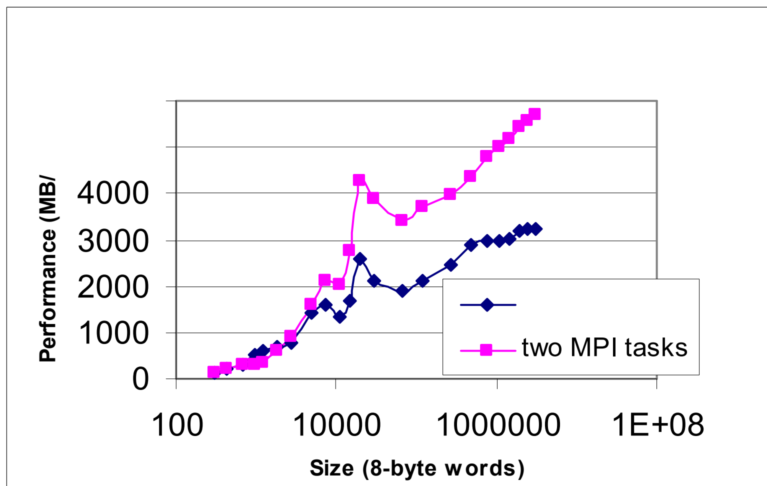

Fig. 20. Serial and parallel DGEMM performance on the Intel Xeon system.

The Intel Xeon system delivered more memory bandwidth for both serial and parallel execution of the DGEMM subroutine. The maximum bandwidth obtained from parallel DGEMM on the Intel Xeon system is twice that measured on the AMD. However, the Intel MKL library is optimized for the Intel architecture and hence should not be expected to deliver the same level of performance on the AMD Opteron. There is very good scaling of the memory bandwidth for both systems in going from serial to parallel execution especially at higher matrix dimensions. This is not surprising since the optimized DGEMM makes use of matrix blocking in order to maximize cache utilization.

4. NAS Parallel Benchmarks (NPB)

The NPB consists of kernels that form the basis of many scientific and mathematical calculations, and is useful for predicting application performance on different platforms. Tables 5 and 6 summarize the NPB benchmark results on the Intel Xeon and AMD Opteron systems.

Table 5. NPB measuments on the AMD Opteron system. Time is expressed in seconds.

	Serial	Dual-processor
cg.B	522	221
is.B	13	7
lu.B	1729	681
mg.B	55	26

Table 6. NPB measuments on the Intel Xeon system. Time is expressed in seconds.

	Serial	Dual-processor
cg.B	876	666
is.B	14	9
lu.B	1437	1182
mg.B	58	49

The parallel performance and scalability of the four NPB programs on the AMD Opteron system are superior to those obtained on the Intel Xeon system. Serial performance of the NPB programs in tables 5 and 6 is generally better or comparable to those on the Intel system, with the exception of LU. The LU kernel is numerically intensive and hence benefits from the faster clock rate on the Intel Xeon.

Remote versus local memory access on the AMD Opteron

The non-uniformity or "NUMA-ness" of the Newisys 2P node architecture arises from the latency difference between local versus remote memory access. The following kernel provides an estimate of the overhead involved in the retrieval of contiguous locations in remote memory:

```
1:     !$OMP PARALLEL Private(time,j1,j2,i,j)
2:           do time=1,ntimes
3:              <assign columns j1-j2 to thread>
4:              A(:,j1:j2)=…
5:              <set j1 to other thread’s old j1>
6:              <set j2 to other thread’s old j2>
7:              do j=j1,j2
8:                 do i=1,n
```

```
 9:                      A(i,j)=time*A(i,j)
10:                   end do
11:                end do
12:             end do
13:      !$OMP END PARALLEL
```

Running the above kernel with and without lines 5 and 6 results in streaming from remote and local memory, respectively. Removing these lines of code ensures that a thread's block of A remains in its processor's local memory. Figure 21 illustrates the difference between local and remote data streaming:

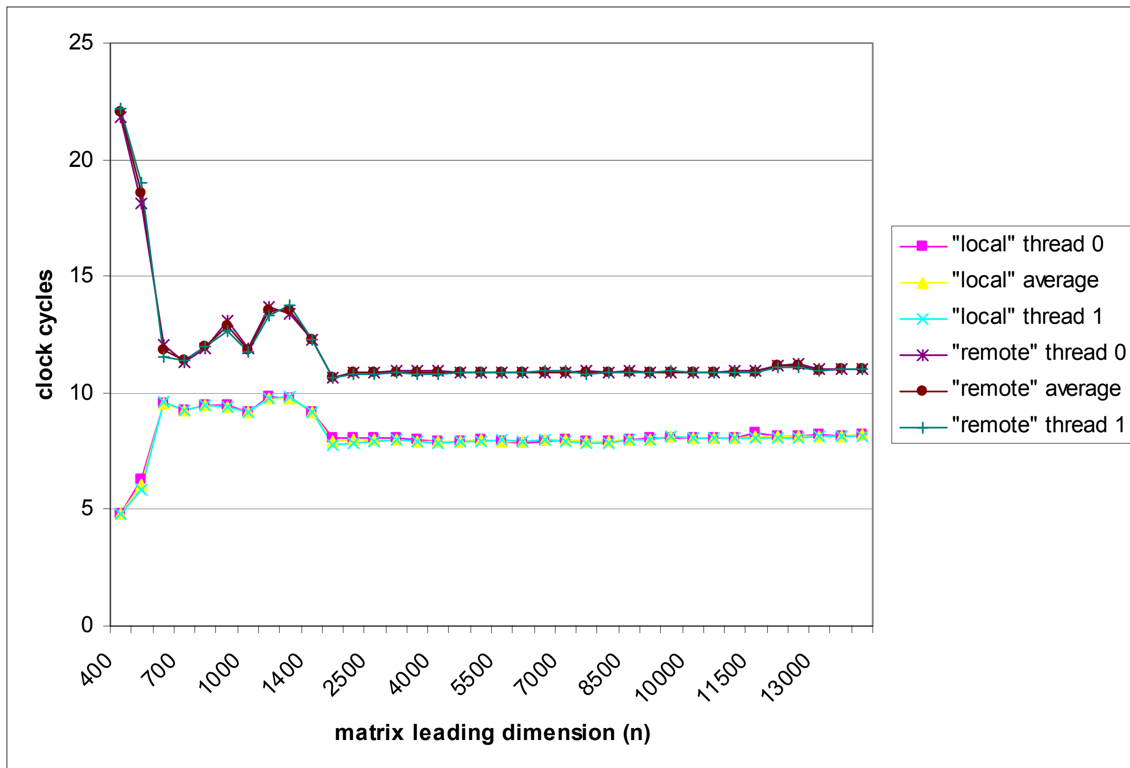

Fig. 21. Overhead of accessing contiguous data from local vs. remote memory.

The time gap is broadest (17 cycles) when each thread's portion of A fits into cache, and reaches an asymptotic value of 3 cycles as the array gets larger. The huge latency difference at small matrix dimensions (i.e. when A fits in cache) is partly explained by what is known as false sharing, which arises when multiple threads are writing to the same cache line. This situation causes severe memory stalls since cache lines are flushed from a processor's cache to memory, and from memory to cache on the other processor if different threads write to the same cache line. False sharing can occur at low matrix dimensions when a thread that has finished its initialization is performing the re-scaling (lines 7-11) while the lagging thread is still at the initialization step.

At matrix dimensions larger than cache, each loop iteration for the "remote access" version of the code takes three clock periods longer than that on the corresponding "local" version of the kernel. This number represents the difference in contiguous read and write to remote memory versus local memory.

III. Conclusions

Our study has focused on two important objectives in characterizing the memory subsystem of two different dual-processor node architectures: 1) measurement of bandwidth and latency to different levels of memory and 2) on-node scalability of memory intensive kernels and applications.

We used the STREAM benchmark and a kernel that calculates sustained read bandwidth from all areas of memory. The STREAM values are mostly unchanged between serial and parallel runs on the Intel Xeon system, since a single processor in a shared memory bus architecture is able to use most of the available bandwidth on the shared bus. In contrast, the AMD Opteron bandwidths show almost 2x scaling from serial to dual-processor execution of STREAM. This is to be expected from a memory architecture where each processor has a separate interface to local memory, thus enabling independent streaming of data from memory to each CPU. From main memory to a single CPU, the read bandwidth on both Intel Xeon and AMD Opteron systems is the same. When two read processes are executed, the AMD Opteron bandwidths scale very well, and each process gets the same bandwidth as that obtained by a single read process from all areas of memory. On the Intel Xeon system, the bandwidth from memory to the CPU is halved in going from serial to dual-processor reads.

Our latency measurements show comparable values for fetching cache-resident data on both Intel Xeon and AMD processors. However, the overhead for retrieving a location from main memory is higher on the Intel Xeon compared to the AMD Opteron. To offset the effects of memory latency, both Intel Xeon and AMD Opteron contain support for hardware data prefetching at runtime. In addition, prefetching of data also increases the memory bandwidth if multiple data prefetch streams are activated. We ran different dot-product loops, with two, four, six, and eight data streams exposed, in order to present prefetching opportunities to the compiler. While loop bisection, trisection, etc. increased the memory bandwidth on the AMD Opteron, the best memory bandwidth obtained for the Intel Xeon corresponds to the unmodified loop.

The scalability of a calculation is affected by its computational intensity, which can be quantified by the ratio of its memory references to FLOPS. The kernels used in our measurements span the different levels of computational intensities found in applications --- from light (indirect dot-product) to moderately high (DAXPY). Execution of a parallel dot-product on two processors shows great scalability on the AMD Opteron system. However, the per-processor performance on the Intel system is less than half that of the serial dot-product run. Calculation of the indirect dot-product (where one vector is addressed using an index array) yields the same per-processor performance on the AMD Opteron system but results in a halving of the per-processor performance on the Intel Xeon system when scaled from one to two processors. The same scalability results apply to the naïve MxM (matrix–matrix multiplication) kernel. Because of its higher computational intensity relative to the other kernels, DAXPY shows good scaling on the Intel Xeon system, and excellent scaling on the AMD Opteron system. While parallel performance is important, it is also necessary to evaluate serial performance on a node. The faster clock rate and slightly higher cache speed on the Intel Xeon processor provide an advantage when running numerically intensive and cache-friendly codes. In fact, serial performance is better on the Intel Xeon for all the numerical kernels tested.

The use of separate memory interfaces on each CPU leads to great scaling of memory bandwidth for dual-processor parallel execution. However, one needs to be aware of remote versus local memory placement of global data in a shared memory parallel application. We ran an OpenMP kernel to calculate the overhead involved in streaming data from remote memory to each CPU. The result of this measurement indicates that there is little additional cost (~3 clock periods per read and write to sequential memory locations) in streaming data from remote memory versus local memory.

Application performance follows the same trend as those observed for the kernels. A computationally intensive simulation like molecular dynamics scales excellently and moderately well on the AMD Opteron and Intel Xeon systems, respectively. Finite difference, which involves strided memory access patterns and has less FLOPS per memory reference than MD, scales rather poorly on AMD (relative to

other applications and kernels) and has worse-than-serial performance when run in parallel on the Intel Xeon system. The NPB timing result for MG reinforces this observation. For all the applications except two of the NPB kernels, the serial performance on the Intel Xeon is superior to that on the AMD Opteron.

In summary, this study revealed the following major points:

- The Intel Xeon memory architecture provides good memory bandwidth to a single processor from all levels of the memory hierarchy. The higher CPU clock speed and slightly better bandwidth from cache to the registers relative to the AMD Opteron provide a performance edge to computationally demanding applications. However, the limitations of the shared bus memory architecture become apparent when executing two memory intensive processes in parallel on a dual-processor node. Because bandwidth is shared, per-processor throughput will decrease for memory bound applications with synchronous memory access needs. The effect of memory contention, and the resulting degradation in performance, is especially bad for random or strided memory access.
- The AMD Opteron processor has a slower clock rate than the Intel Xeon, and consequently will not perform as well for compute-bound applications. Because of the memory architecture and (resulting) excellent scalability of the memory subsystem, the AMD Opteron node is best suited for memory intensive applications that are not cache-friendly, and/or have random or strided access patterns.
- Because of the shared-bus architecture, a serial process executed on the Intel Xeon node can potentially take advantage of the entire FSB bandwidth. The same cannot be said for a serial process running on the AMD Opteron node. Because there is no memory interleaving across the HyperTransport connection, a processor can expect at most the memory bandwidth provided by the two channels to local memory, provided that memory is interleaved across these two channels.

Good scalability of the memory subsystem leads to efficient use of the second CPU in a dual-processor node, since this increases the capability to deliver the additional memory bandwidth required by two processors. On the other hand, excellent scaling cannot compensate for poor single processor performance and inefficient utilization of CPU and memory resources. While the memory subsystem plays a central role in determining application performance, deficiencies in areas like memory scalability must be weighed against possible mitigating attributes like faster CPU clock rates and higher cache rates. In the end, the most important metric of application performance (to the user at least) is still the execution time.

IV. References

1. AMD Opteron processor.
http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_8804,00.html

2. AMD HyperTransport Technology-Based System Architecture.
http://www.amd.com/usen/assets/content_type/white_papers_and_tech_docs/AMD_HyperTransport_Technology_based_System_Architecture_FINAL2.pdf

3. HyperTransport technology. http://www.hypertransport.org/technology.html

4. http://www.dell.com/us/en/esg/topics/esg_pedge_rackmain_servers_1_pedge_2650.htm

5. Intel Xeon processor.
http://www.intel.com/products/server/processors/server/xeon/index.htm?iid=ipp_browse+featureprocess_xeon&

6. http://www.newisys.com/products/index.html

7. NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/

8. STREAM benchmark. http://www.cs.virginia.edu/stream/