Q2 a

Identify any parallelism in the code.

The code consists of an outer loop which executes nbodies times and an inner loop which executes nbodies-i times.

The outer loop iterations are all entirely data independendent and can be executed in parallel. There is a cyclic data dependency in the inner loop on bodies[i].

However the dependence in the inner loop is the result of adding a new value onto bodies[i] each time, that is a sequence of values are summed into bodies[i]. This summing can be performed in parallel.

In principle if an infinite number of processing elements were available the time taken would e O(log nbodies).

In practice if P processing elements are available, we will normalle contruct P sub-sums, assuming that nbodies is much greater than P.

Q2 b

Discuss the suitability of various parallel computer architectures for executing this code, assuming a large array.

(i)

Out-of-order (OOO) superscalar.

An out-of-order superscalar processor exploits very fine-grained instruction-level parallelism (ILP),

by using sophisticated hardware in the processor to detect, and execute in parallel, independent instructions.

Such a processor will have few problems exploiting parallelism in this code, because it is so abundant. There are no poorly predictable branches in the code, and a superscalar processor will be able to disambiguate memory accessed by examining the addresses at execution time, so it will be easy for the superscaler processor to exploit parallelism. The main limiter on the exploitation of the parallelism is the data dependency between succesive values of bodies[i]. A superscalar processor cannot restructure the code at execution time, so it has no way to remove this dependency.

(ii)

Very long instruction word (VLIW).

A VLIW processor also exploits instruction level parallelism, but the compiler is responsible for identifying parallel instructions and scheduling them on the simple hardware of the VLIW. As with the superscalar processor, there is abundant parallelism that could be exploited. The main difference is that VLIW machines normally have extremely sophisticated compilers to restructure the code to exploit more ILP. In the case of this code the compiler may be able to recognize the reduction sum on each field of bodies[i] and restrucutre it into prehaps 2 or 4 separate sums which can be computed in parallel and combined together after the loop.

(iii)

Vector processor.

In addition to regular instructions that operate on one data value, vector architectures have instructions that operate on a whole single dimensional array (vector) of data. This code operates on arrays, so a vector architecture should perform well. If the code were suitably restructured it should be possible to perform the vector width number of iterations in parallel and use a horizontal sum operation to sum these values.

However, a complication is that the program operates on structured data. Different operations on different fields of the structure. However the standard vector load and store operations are aimed at performing the same operation on data that is laid out sequentially in memory.

Therefore it would be necessary to either change the data layout or perform extensive (and expensive) shuffling of vector data to bring together the corresponding data values from each structure.

(iv)

Multithreaded/simultaneous multithreaded.

A multithreaded (MT) architecture runs multiple threads of execution and can switch between them rapidly. Thus, if one thread becomes stalled (perhaps due to a cache miss) another thread can continue to progress. Fine grain MT processors can switch thread every cycle. Coarse grain MTs switch on long stalls and simultaneous MT processors mix instructions from multiple threads in one cycle. A major advantage of a multithreaded architecture is that if the array is large, pipeline stalls due to cache misses can be covered by having more threads running, which can continue to make progress while one thread stalls. To exploit MT architecture the code needs to be divided into multiple threads by either the compiler or the programmer, such as with OpenMP.

(v)

Shared memory multi-core processor.

A shared memory multi-core processor also requires that the code is divided into multiple threads. Once divided this code will run reasonably well on such an...

(vi)

Multi-chip symmetric multi-processor.

The same is true of a multi-chip symmetric multi-processor (SMP). The SMP may in fact be even better for very large arrays/matrices because the separate chips will usually have separate memory buses and thus more total memory bandwidth to fetch array elements. For many problems the cost of communication between different chips can outweigh the benefits of parallelism. However in this particular application there is no communication required between threads. All array entires can be processed indepently.

(vii)

Non-uniform memory access (NUMA).

A numa multiprocessor has a single shared memory address space, but the physical memory is divided into separate regions, typically with each region physically close to one of the processor. Thus, different areas of memory can be accessed more cheaply by some processors and accessing a distant memory can be expensive. In this particular program the entire array is accessed by all threads. Therefore if the array is large, this non-uniform memory cost may be a significant performance penalty. This code executes O(N*N) operations on O(N) data, so there is significantly more computation than data movement, however the locality of reference to the bodies is poor.

(viii)

Distributed memory multicomputer.

A distributed memory multicomputer has no shared memory, and data must be explicitly sent (usually using message passing) from one processor to another for parallel computations. To build an efficient message passing version of this code it...