1a:
The reason we use derived datatype in MPI is mainly for convenience and for a more effective code. Often when we wish to pass a message through MPI, we must use some of the MPIs standard predefined datatype, like MPI_INT MPI_CHAR, etc etc. But often we want to send a pack of non-continuous datatypes f.eks: an coloumn in an array or other form of data, then we can define our own datatypes and send them. Normally the MPI can send over normal types, it can only send different MPI_datatypes. With the derived datatype, we can not only send normal types, but a section of an array, we can send over a struc and so on.

1b:

			0						
			1						
			2						
			3						
			4						
			5						
			6						
			7						
			8						
			9						

	0	1							
		2	3						
			4	5					
				6	7				
					8	9			

Here to illustrate the blocks that got moved, i have chosen to give them a assending numer as names and content.

2a:
The three types of cache misses is : instruction read miss, data read miss, and data write miss.

Instruction read miss is when the instruction is not in the cache, the program would have to fetch the instruction from the main memory. This type of misses would often cause most delay. The way to avoid it would be to read the memory with prefetch. By reading the memory earlier inn with the prefetch, we can get the instruction earlier inn to the cache before the instruction is needed.

Data read miss cause less delay since this type of miss is caused by the data is not in the cache. It would still need to fetch the data from memory, but the cpu could use this spare time to compute other non-data dependent task. A way to avoid this type of misses would be the same as above. By reading inn earlier we would avoid cache miss. And even if we get a miss, we could let the processor do other computation as mentioned before to raise the throughput.

The last one is data write misses, this occurs when we tries to write data to a specific point in the cache, but the cache is not there. This is not a huge problem since we can que the write in a que, or just bypass it and directlie write to memory.

2b:
Temporal and spatial locality is, description of a characteristic in space and time for the cache. Spartial locality is describing a behavior in space for the cache, when a memory is accesed, the neighbour bytes is often used too. The memory access tends to clump together, is byte x is used, then x+1 and x+2 is most likely next. This is why we often fetch not just the specified byte to the cache, but their respectable neighbours too.

If we say that spatial locality describe the space, then temporal locality is the description of time(for cache). We observe temporal locality when a memory access that is just accessed is accessed again. Memory access that have been used, is often reused. This is because programs often repeatedly access the same memory location over and over again. The is why in the cache, we use a LRU(least recently used) protocol to remove old unused cache.

2c:

I) This part of the code is neither, since the code is not accessing the same data. The for loop is accessing the elements in different elements in arrays, and the memory location is not in their close neighbour either, since i increments with 1000 for each steps.

II) Here on the other hand we can observe a spacial locality. The main difference between this and I is that here i is only incrementing with 1. We will therefore access the memory location i, i+1, i+2, etc etc.

III) This code is temporal locality. We can see that the outer for loop does nothing except let the inner for loop run again calling the same memory location that was in the previous iteration. The same memory location a[i], b[i], c[i], will be accessed 100 times in a short period of time.

4a.
Branch prediction is an attempt to predict if an conditional jump would indeed happen. In a conditional branching, it would be a huge timewaste if we have to wait for the execution of of the conditional jump. Instead we can try to predict if the condition would indeed execute. In branch prediction, we try to guess if the condition would pass, and execute it, if afterward we see that the condition did not pass, we would have to discard the previous execution of the instruction and start over in the correct branch. A misprediction would have an impact on the efficiency and increase the overhead. On the other hand, the tradeoff is worth it, this improves the performance by potentially have allready have executed the branch.

4b.
For the code to be beneficial to always run slow, then we must assume some kind of constraints. We have to assume that (f + b) < s. Or else there would be no reason to actually have fast(). The only reason that we would consider to remove fast() is if r is so low, that running special() is non beneficial to us, if r is so small that the time running special is actually worst the to just run slow.

5a.
This can both give a positive and negative result. By storing the result in a table, we can avoid letting the processor do heavy mathematical calculations. The table lookup is often much faster than letting the processor calculate the result. But on the other hand, the outcome to the slow function can vary. If there are too many different outcome, the table would have been huge. And to get a faster lookup, we would like to keep the table in a lower numbered cache, most likely L1 cahce for the fastest lookup time. But the finite amount of space in L1 cache might be better suited for other operations. And if the table is huge and the part where the answer to is not there, then the cache data read miss would injure the performance.

5b.
The cache must at least be big enought for one iteration of the inner for loop. For the exchange to be benefitial, the cache must be able to at least hold 1024 different element at a time. There are multiple numbers that would be asked multiple times. The first itteration would be saking fofr

[0 ,1 ,2 ,...1023], the next would be [128 … 1151], so in to itteration, the numbers [128 … 1023] would be asked twice. so for every itteration, 895 numbers would be asked again, som the most optimal would be to have at least 895.