Subject: Vector performance on a specific situation
Posted by crydev on Tue, 18 Jun 2013 07:01:26 GMT
View Forum Message <> Reply to Message

Hello,

I have a question about the Vector's performance in a specific situation. I have a program that utilizes 8 threads on new systems, heavy utilization of paralellism. Say I have a Vector containing 300 items. I split the indexes of those items over 8 threads, meaning the Vector will be accessed from 8 threads simultaniously, but every thread accesses a different item. The same memory location is never modified.

I have read something about Vector cache lines. What is the performance of the U++ implementation of the Vector in this situation? I tried to copy the thread-specific data into arrays and passed them into the functions, but it seems like just as fast.

If there is a better way to do this, I appreciate any suggestions.

Subject: Re: Vector performance on a specific situation
Posted by mirek on Tue, 18 Jun 2013 08:52:34 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 18 June 2013 03:01Hello,

I have a question about the Vector's performance in a specific situation. I have a program that utilizes 8 threads on new systems, heavy utilization of paralellism. Say I have a Vector containing 300 items. I split the indexes of those items over 8 threads, meaning the Vector will be accessed from 8 threads simultaniously, but every thread accesses a different item. The same memory location is never modified.

I have read something about Vector cache lines. What is the performance of the U++ implementation of the Vector in this situation?

It all dependes on sizeof(T) etc... but if you are doing a lot of access to elements and distribute threads in nearby indicies, cacheline sharing between threads is indeed a big problem.

Note that trivial Vector->Array does not really help here, as individual elements will be likely allocated in the same cachelines cells.

So it all depends on what you are doing with elements. 300 cells does not sound like too many, indicating that per-cell computation is pretty heavy (if it there is any advantage to use multiple threads).

For more qualified reply I would need to know definition of T and some description about computation.

Mirek

Subject: Re: Vector performance on a specific situation
Posted by Novo on Tue, 18 Jun 2013 17:11:09 GMT
View Forum Message <> Reply to Message

crydev wrote on Tue, 18 June 2013 03:01Hello,

I have a question about the Vector's performance in a specific situation. I have a program that utilizes 8 threads on new systems, heavy utilization of paralellism. Say I have a Vector containing 300 items. I split the indexes of those items over 8 threads, meaning the Vector will be accessed from 8 threads simultaniously, but every thread accesses a different item. The same memory location is never modified.

I have read something about Vector cache lines. What is the performance of the U++ implementation of the Vector in this situation? I tried to copy the thread-specific data into arrays and passed them into the functions, but it seems like just as fast.

If there is a better way to do this, I appreciate any suggestions.
If you are just reading data there will be no problems. But if you write to elements (even if they are not shared among threads) you get yourself into false sharing problem. Basically, the idea is that CPU doesn't work with words, it works with cache lines. The simplest way to fix that is to add padding to your data. Example: instead of using raw int you can use a structure below.

```
struct MyData {
    int data;
    char padding[64 - sizeof(data)];
};
```

Size of cache line is usually 64 bytes, so you need to add padding to make you data land onto different cache lines.

Subject: Re: Vector performance on a specific situation
Posted by crydev on Wed, 19 Jun 2013 07:03:01 GMT
View Forum Message <> Reply to Message

Thank you for your answers.

sizeof(T) is 16-bytes.

```
struct
{
    unsigned int;
```

```
    int;

    struct
    {
        int;
        int;
    }
}
```

What my code does is reading a struct instance from the vector and editing the two integer fields in the sub-struct. What I did now, also stated in my first post, is copying the Vector.GetCount() / 8 count of structs from the vector into an array and performing operations there. Afterwards I copy them back into the vector at the original positions.

As I stated, it seems just as fast, the profiler also notes so. Can I conclude from that finding that this operation is faster to prevent cacheline sharing?

Subject: Re: Vector performance on a specific situation
Posted by mirek on Wed, 19 Jun 2013 07:14:17 GMT
View Forum Message <> Reply to Message

crydev wrote on Wed, 19 June 2013 03:03Thank you for your answers.

sizeof(T) is 16-bytes.

```
struct
{
    unsigned int;
    int;

    struct
    {
        int;
        int;
    }
}
```

What my code does is reading a struct instance from the vector and editing the two integer fields in the sub-struct.

This does not sound like awful lot of computation. I think that single thread will handle the task as fast or perhaps faster than multiple threads...

Now if you had 3 millions of elements instead of 300...

(Of course, I can still be mistaken about the amount of computation per element performed. Have you benchmarked that (I mean, single-threaded time to do one element)?

Subject: Re: Vector performance on a specific situation
Posted by crydev on Wed, 19 Jun 2013 07:37:43 GMT
View Forum Message <> Reply to Message

The computation on these elements is not very heavy, but the information in these structs is used to read over gigabytes of memory and compare every byte. If I use only one thread to do that it will be busy for a few minutes, where 8 threads will handle it in a few seconds.

The amount of elements differs per process running on a windows machine. A small process has around 300 pages, which makes the vector contain 300 elements, but bigger processes can contain over 2000 pages, which increases workload a lot.

I have not yet benchmarked it for one thread, because I think it doesn't matter. If you use only one thread you simultaniously read from 0 to the end, where this problem is not really applicable. When 8 threads operate on the Vector, the first one operates on 0-49, the next on 50-99, and so on.