Speed up loop using multithreading in C# (Question)

Asked 16 years, 3 months ago Modified 14 years, 7 months ago Viewed 22k times

16

Imagine I have an function which goes through one million/billion strings and checks smth in them.

f.ex:

```
foreach (String item in ListOfStrings)
{
    result.add(CalculateSmth(item));
}
```

it consumes lot's of time, because CalculateSmth is very time consuming function.

I want to ask: how to integrate multithreading in this kinda process?

f.ex: I want to fire-up 5 threads and each of them returns some results, and thats goes-on till the list has items.

Maybe anyone can show some examples or articles..

Forgot to mention I need it in .NET 2.0

c# multithreading .net-2.0

Share
Improve this question
Follow

edited Sep 19, 2008 at 12:21

Keith
155k ● 82 ● 306 ● 446

asked Sep 19, 2008 at 7:39

Lukas Šalkauskas
14.3k ● 20 ● 63 ● 77

Do you need the results back in the same order? – Keith Sep 19, 2008 at 12:22

Could you use multiple background workers? create some sort of logic that would take the count of list of strings then create X amount of BWs and divy up each one – Crash893 May 5, 2009 at 17:53

6 Answers

Sorted by: Highest score (default)

You could try the Parallel extensions (part of .NET 4.0)

These allow you to write something like:

19

```
Parallel.Foreach (ListOfStrings, (item) =>
    result.add(CalculateSmth(item));
);
```

Of course result.add would need to be thread safe.

Share

Improve this answer

Follow

edited May 3, 2010 at 13:01

answered Sep 19, 2008 at 7:42

Tobi

81.4k ● 6 ● 34 ● 37

in this case, will there be any race condition in the result collection? after all multiple threads may be executing result.add simultaneously... – cruizer Sep 19, 2008 at 7:44

Forgot to mention I need it in .NET 2.0 – Lukas Šalkauskas Sep 19, 2008 at 7:45

Ok, well, then you could look at the Parallel.Foreach sourcecode in reflector... Though I believe there's an entire other layer below it so it won't be a simple copy pasta to get similar functionality in .NET 2.0. – Tobi Sep 19, 2008 at 7:47

Parallel extensions are now part of .net 4.0, so no longer a CTP. :) – noocyte Apr 29, 2010 at 10:18

minor issue: wrong case Parallel.ForEach not Parallel.Foreach – Simon Dec 11, 2010 at 8:09

The Parallel extensions is cool, but this can also be done just by using the threadpool like this:

18

```
using System.Collections.Generic;
using System.Threading;

namespace noocyte.Threading
{
    class CalcState
    {
        public CalcState(ManualResetEvent reset, string input) {
            Reset = reset;
            Input = input;
        }
        public ManualResetEvent Reset { get; private set; }
        public string Input { get; set; }
    }

    class CalculateMT
    {
        List<string> result = new List<string>();
        List<ManualResetEvent> events = new List<ManualResetEvent>();

        private void Calc() {
            List<string> aList = new List<string>();
            aList.Add("test");
```

```
            foreach (var item in aList)
            {
                CalcState cs = new CalcState(new ManualResetEvent(false),
item);
                events.Add(cs.Reset);
                ThreadPool.QueueUserWorkItem(new WaitCallback(Calculate), cs);
            }
            WaitHandle.WaitAll(events.ToArray());
        }

        private void Calculate(object s)
        {
            CalcState cs = s as CalcState;
            cs.Reset.Set();
            result.Add(cs.Input);
        }
    }
}
```

Share

Improve this answer

Follow

edited Sep 22, 2008 at 17:13

answered Sep 19, 2008 at 7:55

noocyte

2,522 ● 5 ● 29 ● 44

1 And how do you know when it is finished? mmm. – leppie Sep 19, 2008 at 12:23

Could have a ManualResetEvent that the WaitCallback function calls and the main thread WaitOne on. – Mats Fredriksson Sep 19, 2008 at 13:12

What is WaitHandle.WaitAll() for ? I'm getting a NotSupportedException : "The number of WaitHandles must be less than or equal to 64" – Eduardo Mello Jul 24, 2010 at 22:57

You are generating more than 64 threads... That's probably not a good idea... :) Try with less threads. – noocyte Aug 4, 2010 at 20:17

12

Note that concurrency doesn't magically give you more resource. You need to establish what is slowing CalculateSmth down.

For example, if it's CPU-bound (and you're on a single core) then the same number of CPU ticks will go to the code, whether you execute them sequentially or in parallel. Plus you'd get some overhead from managing the threads. Same argument applies to other constraints (e.g. I/O)

You'll only get performance gains in this if CalculateSmth is leaving resource free during its execution, that could be used by another instance. That's not uncommon. For example, if the task involves IO followed by some CPU stuff, then process 1 could be doing the CPU stuff while process 2 is doing the IO. As mats points out, a chain of producer-consumer units can achieve this, if you have the infrastructure.

Share Improve this answer Follow

answered Sep 19, 2008 at 7:58

slim
41.2k ● 14 ● 99 ● 128

5

You need to split up the work you want to do in parallel. Here is an example of how you can split the work in two:

```
List<string> work = (some list with lots of strings)

// Split the work in two
List<string> odd = new List<string>();
List<string> even = new List<string>();
for (int i = 0; i < work.Count; i++)
{
    if (i % 2 == 0)
    {
        even.Add(work[i]);
    }
    else
    {
        odd.Add(work[i]);
    }
}

// Set up to worker delegates
List<Foo> oddResult = new List<Foo>();
Action oddWork = delegate { foreach (string item in odd)
oddResult.Add(CalculateSmth(item)); };

List<Foo> evenResult = new List<Foo>();
Action evenWork = delegate { foreach (string item in even)
evenResult.Add(CalculateSmth(item)); };

// Run two delegates asynchronously
IAsyncResult evenHandle = evenWork.BeginInvoke(null, null);
IAsyncResult oddHandle = oddWork.BeginInvoke(null, null);

// Wait for both to finish
evenWork.EndInvoke(evenHandle);
oddWork.EndInvoke(oddHandle);

// Merge the results from the two jobs
List<Foo> allResults = new List<Foo>();
allResults.AddRange(oddResult);
allResults.AddRange(evenResult);

return allResults;
```

Share Improve this answer Follow

answered Sep 19, 2008 at 9:45

Hallgrim
15.5k ● 11 ● 48 ● 54

2

The first question you must answer is whether you should be using threading

If your function CalculateSmth() is basically CPU-bound, i.e. heavy in CPU-usage and basically no I/O-usage, then I have a hard time seeing the point of using threads, since the threads will be competing over the same resource, in this case the CPU.

If your CalculateSmth() is using both CPU and I/O, then it might be a point in using threading.

I totally agree with the comment to my answer. I made a erroneous assumption that we were talking about a single CPU with one core, but these days we have multi-core CPUs, my bad.

Share

Improve this answer

Follow

edited Sep 19, 2008 at 8:07

answered Sep 19, 2008 at 7:53

Mats Wiklander

401 ● 3 ● 10

1 Depends if it's a multi-core system. If you have four cores available, for example, then using four threads should see an approximate four times speed-up in processing (assuming no inter-dependencies between threads). – Greg Beech Sep 19, 2008 at 7:57

1

Not that I have any good articles here right now, but what you want to do is something along Producer-Consumer with a Threadpool.

The Producers loops through and creates tasks (which in this case could be to just queue up the items in a List or Stack). The Consumers are, say, five threads that reads one item off the stack, consumes it by calculating it, and then stores it else where.

This way the multithreading is limited to just those five threads, and they will all have work to do up until the stack is empty.

Things to think about:

- Put protection on the input and output list, such as a mutex.
- If the order is important, make sure that the output order is maintained. One example could be to store them in a SortedList or something like that.
- Make sure that the CalculateSmth is thread safe, that it doesn't use any global state.

Share Improve this answer Follow

answered Sep 19, 2008 at 7:45

Mats Fredriksson

20.1k ● 6 ● 38 ● 57