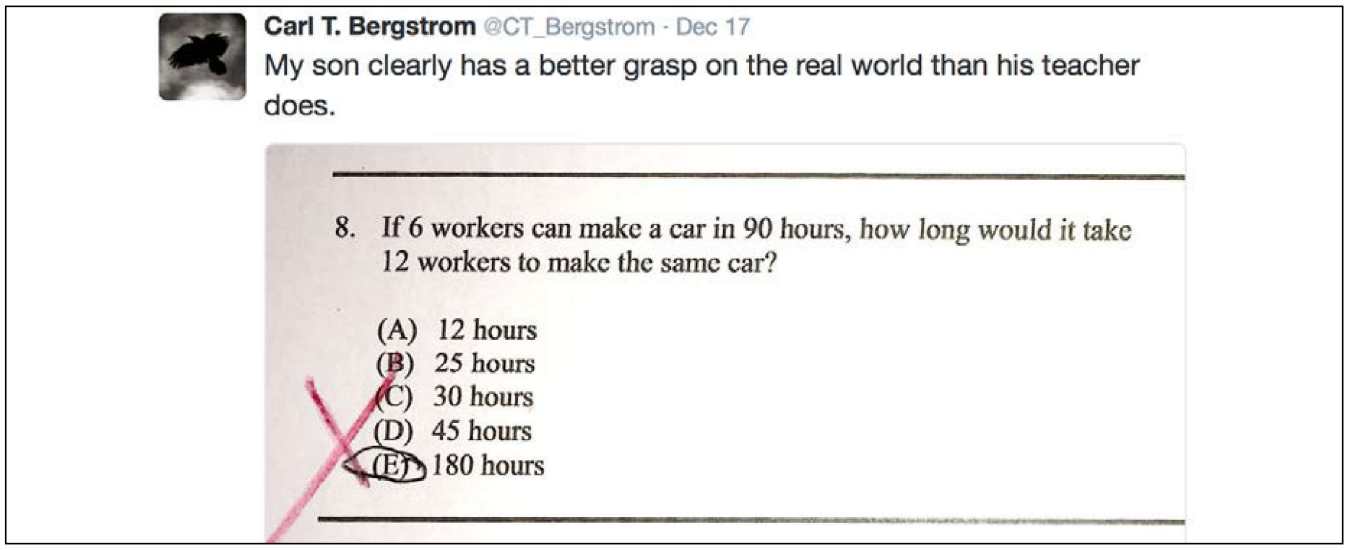

Figure 13.2: Doubling the number of workers doesn't necessarily half the time to complete a task

Good Practice: Never assume that more threads will improve performance! Run performance tests on a baseline code implementation without multiple threads, and then again on a code implementation with multiple threads. You should also perform performance tests in a staging environment that is as close as possible to the production environment.

Monitoring performance and resource usage

Before we can improve the performance of any code, we need to be able to monitor its speed and efficiency in order to record a baseline that we can then measure improvements from.

Evaluating the efficiency of types

What is the best type to use for a scenario? To answer this question, we need to carefully consider what we mean by *best*, and through this, we should consider the following factors:

- **Functionality**: This can be decided by checking whether the type provides the features you need.
- **Memory size**: This can be decided by the number of bytes of memory the type takes up.
- **Performance**: This can be decided by how fast the type is.
- **Future needs**: This depends on the changes in requirements and maintainability.

There will be scenarios, such as when storing numbers, where multiple types have the same functionality, so we will need to consider memory and performance to make a choice.

If we need to store millions of numbers, then the best type to use would be the one that requires the least bytes of memory. But if we only need to store a few numbers, yet we need to perform lots of calculations on them, then the best type to use would be the one that runs fastest on a specific CPU.

You have seen the use of the `sizeof()` function, which shows the number of bytes a single instance of a type uses in memory. When we are storing a large number of values in more complex data structures, such as arrays and lists, then we need a better way of measuring memory usage.

You can read lots of advice online and in books, but the only way to know for sure what the best type would be for your code is to compare the types yourself.

In the next section, you will learn how to write code to monitor the actual memory requirements and performance when using different types.

Today a `short` variable might be the best choice, but it might be an even better choice to use an `int` variable, even though it takes twice as much space in the memory. This is because we might need a wider range of values to be stored in the future.

There is another metric we should consider: **maintenance**. This is a measure of how much effort another programmer would have to put in to understand and modify your code. If you use a nonobvious type choice without explaining that choice with a helpful comment, then it might confuse the programmer who comes along later and needs to fix a bug or add a feature.

Monitoring performance and memory use

The `System.Diagnostics` namespace has lots of useful types for monitoring your code. The first one we will look at is the `Stopwatch` type:

1. In the `Code` folder, create a folder named `Chapter13` with two subfolders named `MonitoringLib` and `MonitoringApp`.
2. In Visual Studio Code, save a workspace as `Chapter13.code-workspace`.
3. Add the folder named `MonitoringLib` to the workspace, open a new **TERMINAL** window for it, and create a new class library project, as shown in the following command:

```
dotnet new classlib
```

4. Add the folder named `MonitoringApp` to the workspace, open a new **TERMINAL** window for it, and create a new console app project, as shown in the following command:

```
dotnet new console
```

5. In the `MonitoringLib` project, rename the `Class1.cs` file to `Recorder.cs`.
6. In the `MonitoringApp` project, open `MonitoringApp.csproj` and add a project reference to the `MonitoringLib` class library, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
```