Transform your TFS Build Agents into 500+ core machines for free (almost)

Accelerate Team Foundation Server using "Process Virtualization" technology

Written by Assaf Stone, Senior ALM Architect at Sela Group, and Dori Exterman, CTO at IncrediBuild

Abstract

The process of releasing new software can be complex, and in many cases, lengthy. A rapid feedback cycle that notifies whether the latest build of the software passed successfully or not can often mean the difference between being able to quickly fix defects in the product, and not discovering those defects until later in the project's lifecycle, thereby increasing exponentially the cost and difficulty of fixing them. While Microsoft's Team Foundation Server (TFS) helps you automate and streamline the process of building software, larger projects may still take too long a time to provide the necessary feedback. This white paper shows how combining the build process automation capabilities of TFS with IncrediBuild's unique process virtualization capabilities can help you speed up your build process by allowing your TFS build machine to utilize *all* the cores in *all* the machines across your network.

Introduction

In this section, we will describe the TFS build process and the limitations that current technology imposes on the process, as well as how to change workflows in order to overcome these limitations. If you are already familiar with TFS (especially versions 2010 or later) and its build process, you may safely skip over the following section.

Team Foundation Server Build Process

Team Foundation Server (TFS) is part of Microsoft's Application Lifecycle Management (ALM) platform, a suite of tools that addresses all of a development team's needs, from requirement gathering and work management, through software development, version management, unit and acceptance testing, to software packaging, building, and deployment. TFS has native integration with Microsoft's own integrated development environment (IDE) called Visual Studio, and under its default settings is geared towards building software written in Microsoft-supported software languages. However, TFS also fully integrates with other IDEs, and can build products in virtually any software language.

Starting with TFS 2010, the build process is managed by running a workflow that can be customized and extended either with Visual Studio's built-in workflow editor or (for the exceptionally brave) with any text editor. This can be done as the workflow itself is stored as an XAML (eXtensible Application Markup Language) file, which is a specialized XML format.

When the development team wishes to build a new version of the product, they send a request to TFS to *queue a new build*. TFS forwards the request to a workload coordinator known as its *Build Controller*. Upon receiving the request, the build controller selects a machine (known as a *Build Agent*) to perform the actual task of building the software, based on availability and any constraints/criteria specified in the *Build Definition* or in the individual build request.

The default build workflow is often used as-is (although in many cases customized and extended by TFS users), goes through the following process:

- It sets up the environment on the build agent.
- It gets the required version of the source code (the latest version, unless specified otherwise).
- It iterates over the projects (and solutions) in the request, and compiles each of them in turn.
- If requested, the build agent then runs automated tests on the compiled output – usually unit-tests and integration tests; however, any process including customized tests or UI-tests can be performed.
- The outputs are then packaged for deployment. This process can be as simple as copying the outputs to a specified folder, or running an MSI-installer generation tool in order to prepare a deployment kit.
- If specified, the build output is copied to a drop location where it is stored for later deployment.
- In some workflows, the build output can actually be deployed to a target site as is often the case with a cloud or web application, or into a *Lab* or test environment, where it can be later tested manually or otherwise.

The TFS build process allocates a single build agent, which often utilizes a single CPU core in order to run this entire workflow. Most of the workflow is done sequentially, and very few parts run in parallel. This means that in order to gain an improvement in performance, it is necessary to increase the speed of the build agent's CPU for computational processes, and the IO capabilities for IO-bound processes, such as reading and writing to the file-system or network.

Moore's Law and the Limits of Single-Process Algorithms

Moore's Law is the observation that over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years. The period is often quoted as "18 months", due to Intel's history of doubling chip performance during that cycle. It is important to note that this is a trend, and by no means a law of nature.

Moore's Law is often understood to refer to the raw computing speed in the form of the chip's clock speed. The trend in chip development seemed to more or less reinforce this interpretation until 2003. From the 1970's and up until that point, the CPU clock speed seemed to double itself as predicted. From that point in time and until today, consumer CPUs cannot safely top that speed, as the heat generated by 4 GHZ CPUs is too great and too difficult to dissipate. And so, CPUs' clock speeds peaked at just over 3 GHZ, and therefore, chip manufacturers diverted their efforts from making faster CPUs to making CPUs with multiple computational cores.

What does this mean? Why is this relevant to you? Simple – the free lunch is over. While as a software developer, you could in the past simply throw more hardware at speed-related performance issues, this is no longer possible. You cannot simply do things faster; you need to do them in parallel, and this requires (some) effort.

Utilizing Multi-Core Machines

One cannot simply take a multi-core machine, say a 2 GHZ dual-core CPU, run a process, and expect it to run twice as fast as a single-core machine with the same clock speed. While it is true that a dual- core CPU can run twice as many operations as its single-core counterpart, it must run them in parallel, two at a time, in order to achieve the goal of doubling the speed. As a software developer, you need to break down processes into independent processes, and run each on a separate core in order to multiply the overall speed. Otherwise, you will remain at the same speed as a single-core CPU, and run on an underutilized machine. You must, therefore, seek to break down and parallelize as many of your CPU-bound operations as you can, in order to maximize the speed of your processes.

Tapping into More Resources

In this section, we look at the different ways in which the TFS build process can be improved in order to make better use of multi-core CPUs. We will examine the benefits, costs, and limitations of each method.

With processing speed capped for the foreseeable future (and further limited by the hardware available at hand), you must change the way you build your software if we are to let the TFS build process provide valuable and timely feedback on the faults of our software. You will need to modify the build workflow by splitting the process into as many independent processes as you can, and run them in parallel.

Before you can do so, you need to make a distinction between the operations that you *can* parallelize in this fashion, and those you cannot. This distinction is made based on where these operations are executed. Operations that are executed by the CPU are called CPU-bound operations. Others, primarily those that involve reading from and writing to resources (such as the memory, file system, or network), are called I/O bound operations.

Since you are trying to improve your usage of our multi-core CPUs, you should focus on CPU-bound operations, which means that you should look to parallelize activities that consist primarily of such operations.

Commonly Used Parallelization Options for CPU-Bound Operations

Parallelization does not happen by itself. While certain software development frameworks may make parallelization easier than others or even make it all but invisible to the developers, some effort must be made in order to achieve it. There are several levels at which you can parallelize your activities. The most common solutions involve setting up a *cluster* (also called a *farm*) made of multiple machines, or setting up a multi-core machine – a *High Performance Computer*.

Following is a description and evaluation of each type of parallelization.

Clustering

Clustering is the act of making use of multiple machines for the same purpose. TFS uses clustering in order to govern the build process, where the build controller allocates a build agent for each build request it receives. In essence, what happens is that the TFS build controller parallelizes the build process *workflow* level, running one complete workflow on each agent, so that multiple build requests can be handled in parallel.

Is Clustering a Good Enough Solution?

In a theoretical situation where an infinite stream of build requests is sent to the controller and the build agents are fully utilized, you will have completed as many workflows as you have agents in the time it takes to run a single build (assuming, of course, for simplicity's sake, that each build takes the same amount of time to process). While the build system is fully utilized, it is not *optimally* utilized.

It would be far better to allocate as many resources as possible to completing *a single* build, and when that build is done, to once again allocate as many resources as possible to running the next build, and so on. This way, rather than having to wait the full length of time for *all* the builds to complete, *one* of the builds can be completed in a fraction of that time, and after that same fraction of time, another build can complete, and so on, until after the full length of time, they are *all* done, just as they would have been if they were running in parallel.

With this in mind, how then can we use a cluster to improve performance for running *one* build through the TFS build process?

Note that the following solution is not advisable for complex projects. While it offers *some* increase in performance, the difficulty to implement and maintain it grows exponentially with its size and complexity.

The first thing you would need to do is break the aforementioned build workflow into multiple workflows, i.e. have a separate workflow for compiling each project solution, running each test, etc. You could do so by having the original workflow send a new build request for each such sub-workflow.

But is this good enough? You would still need to synchronize the environments and outputs of each such workflow. You would need to send the source files to each compiling workflow, retrieve and organize the compilation results, and then send them to new workflows for testing, and synchronize the results of *those* workflows.

It is not impossible to do this, but there is no small amount of work involved in creating and maintaining these workflows – one such workflow would be necessary for each process that you wish to parallelize, and you would have to set up the necessary software on each participating build agent. This means that your parallelization efforts would be capped at the *number of machines that are dedicated to the build process*.

Regardless of the customization efforts, without intimate knowledge of the projects' dependency trees, it is impossible to parallelize at an inter-solution level. Even *with* that knowledge, it is extremely expensive and time consuming to parallelize discrete projects. So as stated above, it is highly unlikely that such a solution will ever be cost-effective.

High Performance Computing

One alternative to governing many single-core agents is to have a single multi-core agent. Most business PCs today have at least four, and often as many as eight cores. However, depending on the needs of each individual software product, your build system may need a lot more than that in order to complete the build in a timely fashion. Remember that if building the software takes too long, and too much time elapses between the moment that the developer requests that the product be built and the time that feedback is received, the developer will move on to the next task on the list, and any defects that are reported will be handled at a (much) later time, increasing the cost even further.

This is where High Performance Computing (HPC) comes in. HPCs are essentially computers (physical or virtual) that can aggregate computing power in a way that delivers much higher performance than that of an individual desktop computer or workstation. HPCs make use of many CPU cores in order to accomplish this. Windows-based machines can support up to 256 cores.

Running the build process on a multi-core agent can be easily parallelized to some extent. If, for example, you parallelize the software projects (solutions) that you need to run, you can compile as many solutions as you have (limited only to the number of available cores) in parallel. If you need to compile them in multiple configurations or for multiple platforms, these too can be easily done in parallel. All you need to do is replace the *For-Each* loops of these activities with *Parallel-For-Each* loops, and they will be executed as such. Additionally, if you are compiling your software with MSBuild (Microsoft's build-task management tool that is utilized in the default build workflows), you can set the maximum CPUs to a number that is as high as the amount of CPUs in your build agent's machine. MSBuild will use as many cores as it can, up to that limit. Your performance improvement depends on MSBuild's task parallelization, which, of course, varies widely based on your software project's dependency tree.

Automated tests can usually be parallelized as well. Unit tests are (or should be) designed to be independent and thus easily parallelizable. Other tests can often be parallelized as well. Depending on your testing framework, you may need to configure either the tool or the build workflow (or both) in a way similar to that described above: replacing loops with parallel loops, and/or allocating more than one core to the operation, if the tool can be thus configured.

The Limitations of High Performance Computing

While the above method can increase your overall performance, limitations still exist. The limit is set primarily by two factors: capacity and granularity.

Firstly, as previously mentioned, a 64-bit Windows-based system is (currently) limited to 256 cores. In many (most) cases this may be more than your product's build needs; however, in other cases, this is a difficult and insurmountable limitation. If your product is broken down into, say, 16 software project solutions, and you need to build each project in two different configurations for four different platforms, you can already utilize 128 cores. This means that if your compiler makes use of more than two cores, you have already reached your limit.

The second factor is granularity. Without spending a great amount of effort into modifying the way your software is compiled, the project-solution (or top-level project) is atomic, and building *it* cannot be done in parallel.

There is of course a third factor: Price. HPCs are extremely expensive, often prohibitively so, costing tens or even hundreds of thousands of dollars. Even if your company can afford to buy one, it is highly unlikely that it will be dedicated in its entirety to the build process.

Going Beyond the Limitations of Clusters and HPCs

Is there a way that you can go beyond the limitations of these methods? Can you parallelize the build process without expending an exorbitant amount of effort or money on breaking down the process into discrete workflows or buying a several hundred thousand dollars machine? Is there a way that you can go beyond the granularity limits of the workflow or solution?

The answer is *yes*. You can use IncrediBuild to virtualize and parallelize individual OS processes and delegate their execution across a cluster.

IncrediBuild Transforms Your TFS Build Agent into a 500-Core High Performance Machine

With IncrediBuild's unique parallel processing solution, you can easily achieve dramatically faster builds. IncrediBuild transforms any TFS build machine into a virtual supercomputer by allowing it to harness idle CPU cycles from remote machines across the network even while they're in use. No changes are needed to source code, and absolutely no additional hardware is required.

IncrediBuild can be incorporated into your existing TFS build workflow, and accelerate your build engine, by replacing the default compiler (such as MSBuild). When activated, it intercepts the calls to the compiler and detours these calls to run remotely on another machine. IncrediBuild makes sure that the remote executor has what it needs for running the compilation process, and the remote machine sends the compilation results back to the initiator (the build agent).

It is important to note that nothing needs to be installed or copied to the remote machine; IncrediBuild copies everything that will be needed by the remote compilation process on-the-fly (including DLLs, executables, source files, etc).

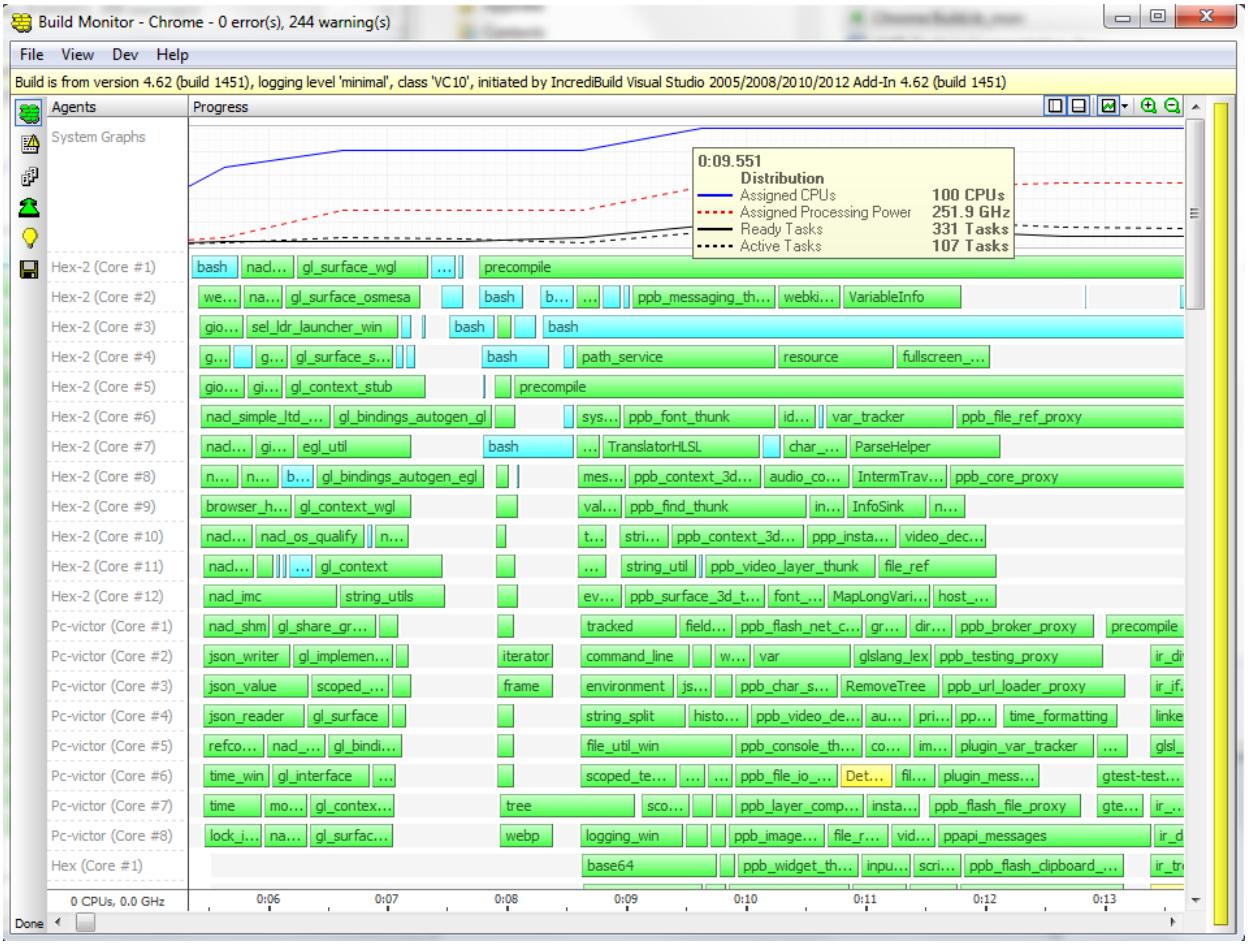


Figure 1: Building the Google Chrome Browser with IncrediBuild

In Figure 1 (above), you can see all the cores participating in the build acceleration (100 cores in this scenario). Every horizontal bar in the Monitor represents a task being executed.

In fact, this can be done for *any process*, whether activated by the TFS build agent or by a developer's workstation! This means that effectively, **IncrediBuild transforms *any machine* into a 500+ core workstation!**

Tapping into your Untapped Resources

As previously mentioned, IncrediBuild does not require any additional hardware. This means that you do not even need to provide a dedicated cluster of machines for IncrediBuild to use; IncrediBuild's agent is a small service that can be installed on any and every machine in your network – both local *and* WAN, meaning that you can effectively scale limitlessly into the cloud, if you so desire!

IncrediBuild's client service is small enough and smart enough to not disturb the normal performance of existing operations on the remote machine. IncrediBuild **only** makes use of idle CPU cycles on any cores

on the network. This means that, as far as IncrediBuild is concerned, you *already have* a cluster for it to use for its operations.

The following section describes how process virtualization works. If you are less interested in the theory of IncrediBuild's process virtualization than you are in the practical application of it, you may safely skip over the following section.

The Solution – Process Virtualization over the Network

The following section describes IncrediBuild's unique implementation of process virtualization for task distribution and workflow parallelization.

What is Process Virtualization?

Process virtualization consists of two actors – the *initiator* machine and the *helper* machines (delegates). The initiator distributes computational processes over the network as if the remote machines' cores were its own, all while the processes executed on the helpers (remote machines) see the initiator's file system and registry instead of the helpers' own.

How Process Virtualization Works

Process virtualization is based on five concepts: *local interception* of requests to execute processes, delegation of these processes for *remote execution*, interception of file system requests performed by the remotely executed processes, on-demand synchronization of these resources from the *initiating* machine to the *helper* machine, and rerouting of the remote process file system calls to the files that were synched and cached from the initiating machine.

Local Interception

When IncrediBuild is set to compile a project, it actually activates the standard compilation tool (Visual Studio, MSBuild, etc.). When the compiler attempts to create a compilation process (e.g. compiling C# with csc.exe), it *intercepts* the call to the operating system, and sends the request, along with the necessary inputs (the compiler executable, source code, and environmental settings) to an idle remote *helper* core for execution. This information is, of course, compressed and cached in order to optimize and reduce network traffic between the initiator and helpers. It is recommended to remotely execute processes which are expected to be computationally expensive.

Remote Execution and Injection (Interception)

Upon receiving a process, the helper (remote machine) begins to execute it, using the parameters it received from the initiator. The helper retrieves any file-system or registry resource that it needs (and is missing in the cache) from the initiator, in order to complete the process. This is done, again, by *intercepting* any I/O or machine-specific information, and forwarding the request back to the initiator. In this way, the helper has access to the initiator's registry, file-system, and other machine-specific information. The helper's core becomes, virtually, and for all intents and purposes, a core of the initiator machine.

Response Synchronization

Finally, when the computation is complete, the output, i.e. the compilation result of this remotely run process is returned to the initiator, where it is stored as though it were run on that machine (any StdOut, StdErr, etc. are transferred as well). In this manner, every lengthy CPU bound process is delegated to another machine, allowing as much as the entire network to work on them at the same time, rather than having one CPU run them all sequentially. All this is achieved without changing the way the software is written, without any need to install anything on the remote machine, and without changing how TFS runs its build workflows.

Executing the TFS Build Workflow with IncrediBuild's Process Virtualization

The following section describes how to modify the default TFS build workflow so that it can parallelize and distribute the build process with IncrediBuild. Note that using IncrediBuild requires installing one instance of IncrediBuild's *coordinator* (the service that is responsible for grid management and agent allocation throughout the network), and installing IncrediBuild *agents* (capable of initiating and helping remote process execution) on all participating machines. None of these services requires a standalone machine, and in fact it is advisable to simply install the agent service on as many machines in the network as possible. The more CPUs that participate in the pool, the greater the computational power of the grid is.

How to Improve the Build Process with Process Virtualization?

In essence, all you need to do in order to improve the compilation process is replace the TFS build activity that runs MSBuild with IncrediBuild's TFS activity, and configure it. You can further improve the performance by removing the *for-each* loop that iterates over the configurations, and configure IncrediBuild to compile the solution for all of the desired configuration and platform combinations.

It is possible to improve the parallelization even further by customizing the TFS build workflow and configuring IncrediBuild to run multiple solutions simultaneously. However, this is a more complex setup, and explaining it goes beyond the scope of this white paper.

In a similar manner, various other computationally intensive processes may be parallelized and distributed throughout the network. The best example of such a process is unit testing. Unit testing is by nature a process that is discrete, independent, and almost entirely CPU-bound. With very little configuration, IncrediBuild can be set to run each test on a different core.

Other forms of automated testing and build-related processes, such as code analysis, packaging, and miscellaneous asset builds can also be executed in a similar manner.

Finally, it is worth mentioning that while this white paper focuses on building software with TFS, *any* developer's workstation can benefit from having hundreds of cores at its disposal, and that any computationally intensive processes can benefit from IncrediBuild, such as compression, encoding, image rendering, running simulations, and so on.

Summary

In a world of ever-increasing demands for both quality and rapid time-to-market, while hardware performance factors have shifted from power and speed to parallelization, it is crucial to parallelize and distribute our software building efforts in a manner that is both efficient and cost-effective.

IncrediBuild helps us do just that.

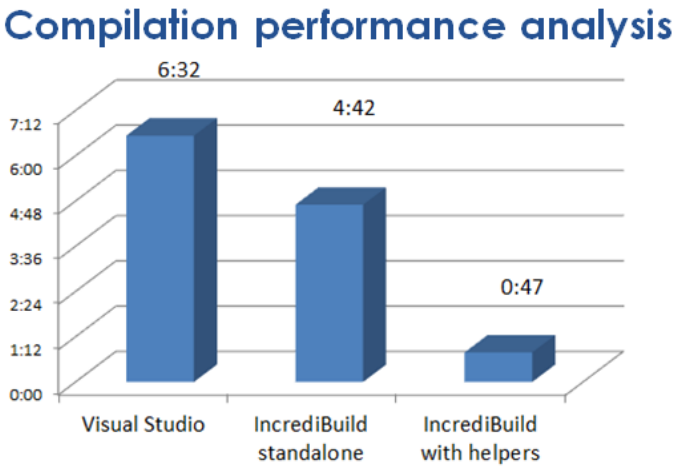
Appendix

Benchmarks

IncrediBuild for Visual Studio
Performance Benchmark

- Performance benchmark was performed by building the ACE open-source project.
- Benchmark was performed on a 4-core Initiator workstation with 64 Helper cores . versus a regular Visual Studio build on the same initiating machine.
- IncrediBuild offers **x8 speed-up**, reducing total build time from 6:32 minutes to 0:47 seconds.

Benchmark results x8 acceleration	
Visual Studio	6:32
IncrediBuild standalone	4:42
IncrediBuild with helpers	0:47



Benchmark done for ACE C++ open source project

- ACE is a popular C++ framework used to simplify various aspects of network programming.
- Helper machines are desktop machines containing 2-4 cores each.

Figure 2: Running IncrediBuild - time to compile the ACE framework. Lower is better.