BuildPal Documentation

Release 0.1.1

PKE sistemi

August 05, 2014

Contents

Distributed compilation of your C/C++ project

Summary

Release 0.1.1
Date July 31, 2014
Authors **PKE sistemi**
Target developers
status alpha

CHAPTER 1

Introduction

1.1 What is it?

`BuildPal` is a tool for speeding up large C/C++ project build by distributing compilation to other machines on the network.

1.2 Why another distributed compiler?

The existing open-source distributed compilers have some, if not all, of the following limitations.

- **No Windows support.**

  Pretty much all open-source distributed compilers are designed for *NIX systems, usually targetting GCC.

- **No precompiled header support.**

  It is difficult to reduce project build time if you have to forfeit the best single-machine optimization.

- **Sub-optimal task distribution algorithms.**

  Task distribution is usually round-robin, possibly weighted by a number of parallel jobs a machine can perform. This does not necessarily work well with a farm containing slaves with heterogeneous performance characteristics.

- **Slow task propeller.**

  Pushing tasks from the client machine to the farm must be as fast as possible. The speed of the leader is the speed of the gang.

`BuildPal` tries to overcome all of the mentioned limitations.

1.3 Features

- **Easy setup**

  No additional files, other than `BuildPal` Server, are needed on the slave machines. All required files are automatically transferred on-demand.

- **Non-intrusive**

  `BuildPal` does not require any changes to build system or project files.

- **Node auto-detection**

  Build nodes on LAN are automatically detected and used.

- **Build Consistency**

  `BuildPal` takes care to produce object files which are equivalent to the files which would be produced on local compilation.

- **Remote preprocessing**

  `BuildPal` does not preprocess headers on the local machine. Headers used by a source file are collected and transfered to the slave. These headers will be reused by the slave machines for subsequent compilations.

- **PCH support**

  `BuildPal` supports precompiled headers. Precompiled headers are created locally, on the client machine and are transferred to slave machines as needed.

- **Self-balancing**

  `BuildPal` tries to balance the work between the nodes appropriately by keeping track of their statistics, giving more work to faster machines. Additionally, if a node runs out of work, it may decide to help out a slower node.

1.4 Supported platforms and compilers

At the moment, the only supported compiler toolset is MS Visual C++ compiler.

This includes:

- `Visual C++ 2008 (9.0)`
- `Visual C++ 2010 (10.0)`
- `Visual C++ 2012 (11.0)`
- `Visual C++ 2013 (12.0)`

CHAPTER 2

Quick Start

2.1 Requirements

1. A C/C++ project, using a build system capable of running parallel tasks.
2. A client build machine connected to a Local-Area Network.
3. As many as possible machines (slaves) on LAN capable of running the compiler your C/C++ project uses.
    - Given that the only compiler currently supported is MSVC, this means that all slave machines need to run Windows.

2.2 Setting up Server (slave) machines

On each machine:

- Install BuildPal. This will create 'BuildPal' program group.
- Run the 'BuildPal Server' shortcut.

That's it - the server will be automatically discovered by client machine via UDP multicast.

2.3 Setting up the Client

- Install BuildPal. This will create 'BuildPal' program group.
- **Run the 'BuildPal Manager' shortcut.**
    - The Manager is the mediator between a compilation request and the build farm. It performs many tasks, including:
        * Server detection.
        * All network communication towards the farm.
        * All (IPC) communication with the clients (i.e. compilation requests).
        * **Source file preprocessing.**
            · Needed in order to determine which files are required for successful remote compilation.
        * **Local filesystem information caching.**
            · Source file contents.

· Preprocessing results.

- **Run the 'Buildpal Console' shortcut.**
  - This opens a new command line window. From here you should start your build. Any compiler processes started from this console will be intercepted and distributed to farm.
  - When starting your build, increase the number of parallel jobs (typically `-jN` option).

2.4 BuildPal Console

The console is used to run the build. It is a regular `cmd` console, with installed hooks which detect when a compiler process is created.

BuildPal has who kinds of consoles. The difference between the two is in the method how compilation request is distributed after being intercepted.

2.4.1 Compiler Substitution (default)

`BuildPal` provides a drop-in compiler subtitute `bp_cl.exe`. When buildpal detects that the compiler process is about to be created, it replaces the call to `cl.exe` to `bp_cl.exe`. Note that `bp_cl.exe` is small and relatively lightweight, so most modern hardware should not have any problems in running many concurrently.

2.4.2 CreateProcess Hooking (experimental)

There is a faster, albeit less general and less safe method.

The idea is to intercept all calls a build system makes to the compiler, and to delegate this work to the farm, completely avoiding compiler process creation on the client machine. `BuildPal` will try to fool the build system into thinking that a process was actually created.

This approach works for most build systems. It will not work if the build system attempts do to anything 'smart' with the (supposedly) created compiler process. For example, this technique will not work with *MSBuild*.

2.5 Using BuildPal as a Python package

Starting the server:

```
python -m buildpal server
```

Starting the manager:

```
python -m buildpal manager
```

Starting the build (compiler substitution):

```
python -m buildpal client --run <build_command>
```

Starting the build (CreateProcess hooking):

```
python -m buildpal client --no-cp --run <build_command>
```

CHAPTER 3

`BuildPal` and some build systems

BuildPal works best with build systems which support *-j* option. Although every build system will work with *compiler substition* hook, *createprocess* hook will work better. Here is the current state of affairs some common build systems:

Build system	has -j option	supports cp hook
Boost.Build	yes	yes
JOM	yes	yes
MSBuild	no	no
Ninja	yes	yes
Nmake	no	yes
SCons	yes	yes

It seems that Microsoft really goes out of its way to prevent parallel build support with their build systems.

CHAPTER 4

Building `BuildPal`

In order to build `BuildPal` you need:

- `Python 3.4` (with setuptools)
- `Visual C++ 2012 (11.0)`

Other dependencies will be downloaded and built automatically. setuptools comes bundled with Python 3.4. If it is missing for some reason you can easily install it by running:

```
python -m ensurepip
```

4.1 Get the sources from GitHub

Get the sources from BuildPal GitHub repository.

4.2 Building

BuildPal uses distutils and setuptools. Just use any of the usual setuptools targets:

```
python setup.py build
python setup.py install
python setup.py develop
...
```

See `python setup.py --help`

Note: First time build will take a while. BuildPal will download, unpack and build several chubby libraries (*Boost* and *LLVM/Clang*). Subsequent builds will be much faster.

4.3 Creating standalone packages with `cx_Freeze`

Usually, you want to avoid installing Python on every machine on the build farm. For this you can create an stand-alone distribution package with cx_Freeze.

- Install cx_Freeze.

Note: You need to use cx_Freeze 4.3.3 or newer. Previous releases do not support Python 3.4 very well. In addition, if cx_Freeze release is not built for your exact Python release (including minor version), there is a good chance that the executable it produces will not work. If this happens, you need to build cx_Freeze yourself.

- Do either `setup.py install` or `setup.py develop`.
- Run `cx_freeze_setup.py bdist_msi`

CHAPTER 5

Benchmarks

5.1 Environment

- 100Mbit/s Ethernet network.
- Client machine: 4 core i3-M39, 2.67GHz, 8GB RAM
- Slave #1: 8 core Intel i7-2670QM, 2.20GHz, 6GB RAM
- Slave #2: 8 core AMD FX-8120, 3.10GHz, 4GB RAM
- Slave #3: 4 core Intel i5-2430M 2.40GHz, 6GB RAM

The client machine is by far the weakest one. Build times are about 4-5 times longer than on #1.

Benchmarks are done by compiling real code. As Boost and Clang are required for BuildPal itself, it was only natural to use these libraries for benchmarking.

5.2 Boost

Tested command for building boost is:

```
b2 stage --stagedir=. link=static runtime-link=shared -j64 toolset=msvc-11.0 --build-type=complete -a
```

The numbers in the table are seconds it took to build the project. Each entry is the mean of 10 consecutive builds, along with standard deviation.

	regular [3]	local [4]	client + 1 slave	client + 2 slaves		client + 3 slaves
client						
slave #1	220.3?2.8	272.3?4.0	242.6?2.3	#2+#3	211.1?2.8	
slave #2	294.6?7.9	400.2?3.5	377.2?5.2	#1+#3	166.0?1.7	128.5?4.0
slave #3	419.1?12.9	466.6?15.9	446.5?5.1	#1+#2	153.8?4.1	

5.3 Clang

Todo

[1] regular project build, without BuildPal
[2] BuildPal build with a single server running locally
[3] regular project build, without BuildPal
[4] BuildPal build with a single server running locally

Measure Clang build times.

CHAPTER 6

Dependencies

"Theft from a single author is plagiarism. Theft from two is comparative study. Theft from three or more is research."

—Anonymous

6.1 Python

6.2 LLVM & Clang

6.3 Boost

Used by all C++ parts of the project.

- *Boost.ASIO* for Client's (`bp_cl.exe`) IPC.
- *Boost.MultiIndex* for Managers header cache.
- *Boost.Spirit* as an alternative to `atoi`/`itoa`/etc.
- *Boost.Thread* for read-write mutexes.
- ...

6.4 pytest

6.5 cx_Freeze

CHAPTER 7

Bugs and caveats

- **Debug symbols/PDB files/precompiled headers.** It is difficult to handle PDB file generation when distributing build. PDB format is closed and there is no known way to merge two PDB files into a single one. In other words, if two objects are compiled on different servers, `BuildPal` cannot create a single PDB containing debug info for both objects.

  BuildPal currently avoids the issue by replacing any `/Zi` compiler switches it detects with `/Z7`, i.e. debug info gets stored in the object file itself.

  When generating precompiled headers there additional complications, so `/Zi` is just dropped. You won't get debug info for PCH itself.

- **Header cache and volatile search path** Cache assumes that a fixed search path and header name will always resolve to the same file. If you place a new header file in a directory on include path before the pre-existing header file with the same name, it is possible that the pre-existing header will still be used.

CHAPTER 8

Future development wish-list

- **Support more platforms.**
  - GCC compiler support (Windows).
  - Clang compiler support (Windows).
  - Linux platform support (GCC/Clang).
  - ...
- IPV6 support.
- **Communicate with the farm via a single machine (supervisor)**
  - Let the supervisor dispatch tasks to other machines.
  - This would make the farm 'client aware', providing better performance when multiple clients use the same farm.
- Create a file system driver for the Server to allow mimicking Client's file system hierarchy (currently done in userland via DLL injection/API hooking).
- **Object file caching support.**
  - Farm could store object files, and return them later on in case of a duplicate request.
- **Reporting.**
  - Generate detailed report about build process.
  - Report information is already being collected and stored in the database, but is not yet user-friendly.