Multi-Core CPU's require new development approaches

Robert LeRoy

April 3, 2008

Hardware Advances

Traditionally, the majority of Sogeti's development efforts have assumed the application would always execute on a single CPU. The entire computer configuration is seldom taken into our design considerations; application performance could be increased by upgrading to a faster CPU. In the rare cases when multiple CPU's were required, the solution was load balancers; but, the programming logic wouldn't change.

Thanks to chip makers Intel and AMD, computers have new multi-core systems; however, designs continue to be single-threaded applications. Even large-scale web applications are single-threaded and rely on application servers to manage concurrency.

When you consider that data volumes are increasing dramatically, SOA is everywhere and mash-ups rule the web, these are all reasons to build parallelism in our applications. The future, specifically cloud computing, promises to make this an even bigger issue.

To put this into perspective, I recently found that Microsoft released a .Net toolkit called ParallelFX. This package simplifies the process of leveraging the full capacity of a system without having to write custom thread management. I used this framework to explore how it affects performance.

Puzzling Vacation

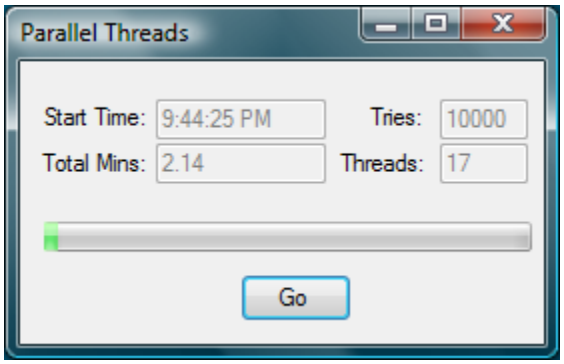
On a recent family vacation, my sister brought a puzzle with only nine squares. Along the four edges of each square is a head or tail of a dog. To complete the puzzle, you have to match the heads and tails of all the dogs. I spent hours and hours on this and couldn't solve the puzzle. So, being a technical person who likes to tinker, I wrote a program.

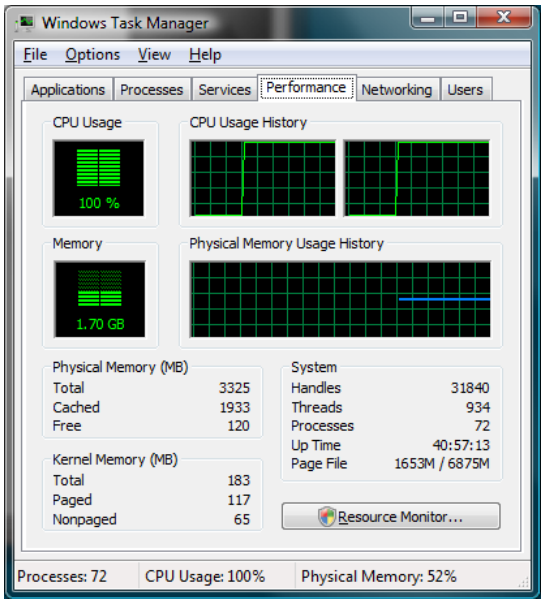
Nine tiles factors out to over 380,000 permutations. Each permutation had to run nine nested loops of four steps which totals 292,000 calculations. That's 99 billion combinations. An excellent test for parallelism!

This is a great example because each permutation can be executed independently. Large result sets from a database can also be processed in parallel. Likewise, if running a mash up, it makes sense to execute all the calls in parallel. Although the true benefit in mash ups may be questionable due to latency and other network traffic.

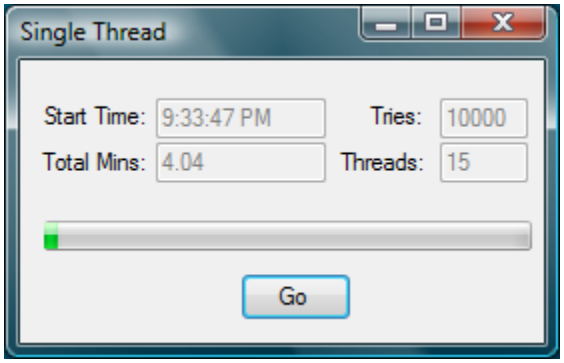
Parallel Solution

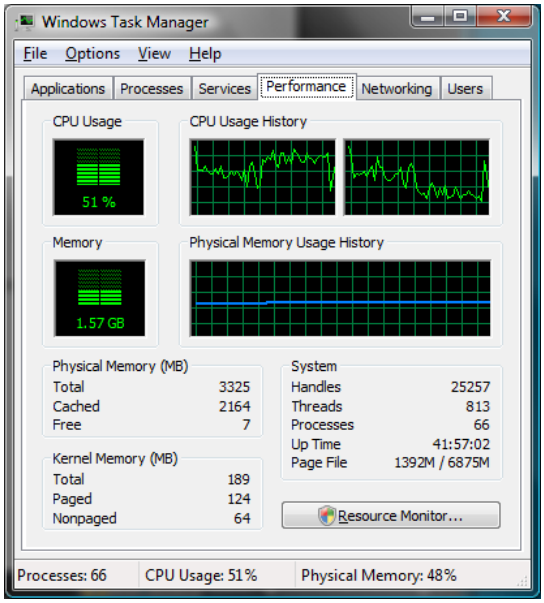
To complete this example in a manageable time frame, I only ran 10,000 permutations. It took my Dell D820 laptop only two minutes to execute them all in Parallel. Note that task manager shows both CPU's running at full capacity.





After switching the logic to a traditional single-threaded loop, it took over four minutes to complete the same amount of work. Task manager shows that the computer was running at 51% capacity.





That's an increase of nearly 100% so the overhead from the ParallelFX is minimal. If the computer had more processors, it would have taken advantage of them and reduced the duration further. As manufacturers continue to add multiple cores to CPU's and multiple CPU's to servers, the benefit of this model increases rapidly.

Below is a snippet of the actual code. The key to focus on is the "Parallel.For" statement highlighted in Red. Within ParallelFX, there are multiple implementations of the 'For' method and also 'While' and 'Foreach' implementations. By uncommenting the 'for' statement, we can run the same logic in a single-threaded model.

```
private void SolvePuzzle()
{
    compute permutations
    int count = combsPuzzle.Count;
    //for (int t = 0; t < count; t++)
    Parallel.For(0, count, delegate(int t)
    {
        loop variable declarations
        int[,] currSet = (int[,])combsPuzzle[t];

        for (int t1 = 0; t1 < 4; t1++)
        {
            a3 = currSet[0,(2 + t1) % 4];
            for (int t2 = 0; t2 < 4; t2++)
            {
                b1 = currSet[1,(0 + t2) % 4];
                for (int t3 = 0; t3 < 4; t3++)
                {
                    c1 = currSet[2,(0 + t3) % 4];
                    for (int t4 = 0; t4 < 4; t4++)
                    {
                        d2 = currSet[3,(1 + t4) % 4];
                        for (int t5 = 0; t5 < 4; t5++)
                        {
                            e1 = currSet[4,(0 + t5) % 4];
                            for (int t6 = 0; t6 < 4; t6++)
                            {
                                f1 = currSet[5,(0 + t6) % 4];
                                for (int t7 = 0; t7 < 4; t7++)
                                {
                                    g2 = currSet[6,(1 + t7) % 4];
                                    for (int t8 = 0; t8 < 4; t8++)
                                    {
                                        h1 = currSet[7,(0 + t8) % 4];
                                        for (int t9 = 0; t9 < 4; t9++)
                                        {
                                            i1 = currSet[8,(0 + t9) % 4];
                                            if (CheckSolved())
                                            {
                                                Boolean solved = true;
                                            }}}}}}}}}}

        );
    }
```

Conclusion

This example is written in .Net, but there are many multi-threading design patterns that can be leveraged. Most involve a manager thread that assigns tasks to worker threads. For more information, search the internet for "thread design patterns". There are many, many articles written on this topic.

Clients expect "thought leadership" and this is another way to demonstrate capability. The best way to take full advantage of powerful, multi-core systems is by writing multi-threaded applications. I encourage everyone to apply these concepts to their development.