Uma analogia pode ajudar.

Você tem um monte de cartas que precisam ser entregues a diversos endereços na cidade. Então você contrata um *motoboy* para entregar as suas cartas.

Considere que sinais de trânsito (semáforos, faróis) em sua cidade são *perfeitos*. Eles estão sempre verdes a menos que alguém esteja no cruzamento.

**Adicione *threads***

O *motoboy* precisa entregar rapidamente várias cartas. Uma vez que não há mais ninguém nas ruas, cada luz está verde. Mas isso poderia ser mais rápido. Melhor contratar outro piloto para a moto.

O problema é que você ainda só tem uma moto. Então agora o seu primeiro *motoboy* dirige a motocicleta por um tempo, e então de vez em quando ele pára, abandona a moto, e o segundo contratado começa pilotá-la.

Isso fica mais rápido? Não, claro que não. Isto é mais lento. **Adicionando mais *threads* não faz nada mais rápido**. *Threads* não são mágicas. Se um processador é capaz de fazer um bilhão de operações por segundo, acrescentando outra *thread* não fará o processador executar outro bilhão de operações por segundo. Em vez disso, ele rouba recursos de outras *threads*. Se uma moto pode correr a 180 km/hora, parando a moto e ter um outro piloto pulando nela não irá tornar a entrega mais rápida! Claramente, em média, as cartas não estão sendo entregues mais rapidamente neste esquema, elas apenas estão sendo entregues em uma ordem diferente.

**Adicione processador**

OK, então o que acontece se você contratar dois pilotos e duas motos? Agora você tem dois processadores e uma *thread* por processador, de modo que vai ser mais rápido, certo? Não, porque nós esquecemos sobre os semáforos. Antes, só havia uma motocicleta dirigindo em alta velocidade a qualquer momento. Agora, existem dois *motoboys* e duas motocicletas, o que significa que agora, por vezes, uma das motocicletas deverá esperar, já que a outra está no cruzamento. Mais uma vez, a adição de mais *threads* retarda o andamento já que você gasta mais tempo disputando os cruzamentos das ruas. **Quanto mais processadores você adicionar, pior fica**, você acaba com mais e mais tempo de espera no sinal vermelho e menos e menos tempo entregando cartas.

Adicionando mais *threads* **pode** causar **escalabilidade negativa se tiver mais travamento (*lock*) competindo entre si**. Quanto mais *threads*, mais disputa, e tudo fica mais lento.

Claro que você terá dois entregadores trabalhando, isto pode deixar a entrega mais rápida se eles não perderem muito tempo parados nos semáforos.

**Aumente o *clock***

Suponha que você arrume motocicletas mais potentes - agora você tem mais processadores, mais*threads* e processadores mais rápidos. Agora isso sempre será mais rápido? Não. Quase sempre, não. O aumento da velocidade do processador pode fazer programas *multithread* ficarem mais lentos. Mais uma vez, pense sobre o tráfego total.

Suponha que você tem uma cidade com milhares de motoristas e 64 motocicletas, todos os*motoboys* vão se revezando entre as motos, algumas delas estão em cruzamentos bloqueando outras motocicletas. Agora você tem todas essas motos correndo mais rápido. Será que isso ajuda? Bem, na vida real, quando você está dirigindo por aí nas ruas normais, você chegará ao seu destino duas vezes mais rápido em um Porsche do que em um Honda Civic? Claro que não, **a maior parte do tempo, dirigindo pela cidade, você está preso no trânsito**.

Se você pode dirigir mais rápido, muitas vezes você acaba esperando mais tempo no trânsito porque você acaba dirigindo mais rápido no congestionamento. Se todas pessoas chegam mais rápido ao congestionamento, então o congestionamento piora.

**Exemplos onde ajuda e onde não ajuda**

Desempenho *multithread* pode ser profundamente contraditório. Se você quer alta performance é recomendável não usar uma solução de várias *threads*, a menos que você tenha uma aplicação que é "intrinsecamente paralela" - ou seja, alguma aplicação que é, obviamente passível de usar vários processadores, como por exemplo, cálculo do [conjunto de Mandelbrot](http://pt.wikipedia.org/wiki/Conjunto_de_Mandelbrot) que faz *[ray tracing](http://pt.wikipedia.org/wiki/Ray_tracing)*. E então, não coloque no problema mais *threads* que processadores disponíveis. Portanto, **para muitas aplicações, usando mais *threads* pioram a performance**. Usando um exemplo mais simples, somar uma lista de números quaisquer pode ser paralelizada, mas achar [sucessores Fibonacci](http://pt.wikipedia.org/wiki/N%C3%BAmero_de_Fibonacci) não pode.

Outra forma de ver isso: **Nove mulheres não farão um bebê em um mês**.

**Custo de administração**

Suponha que você tem uma tarefa a realizar. Vamos dizer que você é um professor de matemática e você tem 20 trabalhos para corrigir. Leva 2 minutos para corrigir cada um, por isso vai demorar cerca de 40 minutos.

Agora vamos supor que você decida contratar alguns assistentes para ajudá-lo. Demora uma hora (60 minutos) para localizar 4 assistentes. Cada um de deles corrigirão 4 trabalhos e tudo será feito em 8 minutos. Você trocou 40 minutos de trabalho por 68 minutos no total, incluindo a hora extra para encontrar os assistentes. Isto não é um ganho. **A sobrecarga de encontrar os assistentes é maior do que o custo de fazer o trabalho sozinho**.

Agora, suponha que você tem 20.000 trabalhos para corrigir, por isso vai demorar cerca de 40.000 minutos. Agora, se você passar uma hora encontrando os mesmo 4 assistentes, isso é uma vitória. Cada um pega 4000 trabalhos e são gastos um total de 8060 minutos em vez de 40.000 minutos, uma economia de quase 5 vezes. A sobrecarga de encontrar os assistentes é basicamente irrelevante.

Paralelização não é grátis. **O custo de dividir o trabalho entre as diferentes *threads* deve ser pequena em comparação com a quantidade de trabalho realizado por *thread***.

**O problema não é de processamento**

Se as tarefas não usam fortemente o processador então claramente não pode haver algum aumento de velocidade, porque enquanto o processador está ocioso esperando o disco ou a rede responder, ele poderia estar fazendo o trabalho de outra *thread*.

Vamos supor que você tem duas tarefas ligadas ao processador (que usa processamento propriamente dito), um único processador, e uma *thread* ou duas *threads*. Ignorando o tempo de administração, no cenário de uma *thread* temos os seguinte:

* Fazer 100% do trabalho da tarefa 1. Suponha que isso leva 1.000ms.
* Fazer 100% do trabalho da tarefa 2. Suponha que isso leva 1.000ms.

Tempo total: 2 segundos. Total de tarefas feitas: 2. Mas aqui é a parte importante: **o cliente que estava esperando pela tarefa 1 tem seu trabalho feito em apenas 1 segundo**. O cliente que estava à espera da tarefa 2 teve que esperar 2 segundos.

Agora, se temos duas *threads* e uma CPU vemos o seguinte:

* Fazer 10% do trabalho da tarefa 1, por 100ms.
* Fazer 10% do trabalho da tarefa 2, por 100ms.
* Fazer 10% do trabalho da tarefa 1
* Fazer 10% do trabalho da tarefa 2 ...

Mais uma vez, o tempo total de 2 segundos, mas desta vez **o cliente que estava esperando pela tarefa 1 tem seu trabalho feito em 1,9 segundos, quase 100% mais lento do que o cenário de uma *thread***!

E nem foi considerado o tempo para ficar trocando de tarefas que não é tão pequeno.

Então se as seguintes condições existem:

* as tarefas são limitas pela capacidade da CPU
* há mais *threads* que CPUs
* A tarefa é útil apenas pelo seu o resultado final e não suas partes

Então, **adicionar mais *threads* só torna tudo mais lento**.

Mas se qualquer uma das seguintes condições não forem satisfeitas, adicionar mais *threads* é uma boa ideia:

* Se as tarefas não são limitadas pela CPU, adicionar mais *threads* permite que a CPU trabalhe quando estaria ociosa, esperando pela rede ou disco, por exemplo.
* Se houver CPUs ociosas, adicionando mais *threads* permite que essas CPUs sejam agendadas para trabalhar.
* Se os resultados parcialmente computados são úteis, adicionando mais *threads* melhora a situação porque há mais oportunidades para que os clientes consumam resultados já computados. No segundo cenário, por exemplo, os clientes de ambas as tarefas estão obtendo resultados parciais a cada 200 milissegundos, o que é importante.

*Thread* não é a única solução para tornar CPUs ociosas por fatores externos disponíveis para uso. Mas isso é outra estória.

O crédito desta resposta é essencialmente do [Eric Lippert](http://stackoverflow.com/users/88656/eric-lippert) nas respostas:

* [Threading vs single thread](http://stackoverflow.com/questions/2902264/threading-vs-single-thread/2902718#2902718)
* [How does threading save time?](http://stackoverflow.com/questions/17279392/how-does-threading-save-time/17280253#17280253)
* [Parallel.ForEach Slower than ForEach](http://stackoverflow.com/questions/6036120/parallel-foreach-slower-than-foreach/6036611#6036611)