برنامه نویسی موازی با #C

توسعه مهارت موازی سازی با

.NET Framework 4

محمد زحمتکش کناری
zahmatkesh8519@gmail.com

تقدیم به

پدر و مادر مهربانم به پاس تعبیر عظیم و انسانی شان

از کلمه ایثار و از خودگذشتگی شان

به پاس عاطفه سرشار و گرمای امیدبخش وجودشان

که در سردترین روزگاران بهترین پشتیبانم است

به پاس قلب های بزرگشان که فریاد رس است و

سرگردانی و ترس در پناهشان به شجاعت می گراید

و به پاس محبت های بی دریغ شان که هرگز فروکش

نمی کند.

این مجموعه را به پدر و مادر عزیزم تقدیم می کنم.

فهرست مطالب :

فهرست شکل ها : .... ه

فهرست تکه کد ها : .... و

فهرست لیست ها (کد ها کلی) : .... ز

چکیده : ۱

مقدمه : ۲

۱ فصل اول ۳

برنامه نویسی TASK-BEASE ۳

مباحثی که در این فصل مطرح می شود : ۳

۱-۱ کار با پردازنده های چند هسته ای SHARED-MEMORY ۴

۱-۱-۱ اختلاف پردازشگر های چند هسته ای حافظه– مشترک با سیستم های حافظه- توزیع شده ۷

۱-۱-۲ برنامه نویسی موازی و برنامه نویسی چند هسته ای ۹

۱-۲ درک نخ های سخت افزاری و نخ های نرم افزری ۱۰

۱-۳ درک قانون AMDAHL ۱۷

۱-۴ ملاحظه قانون GUSTAFSON ۲۱

۱-۵ کار با همزمانی سبک وزن ۲۵

۱-۶ ایجاد موفق طرح های TASK-BASED ۲۶

۱-۶-۱ طراحی با همزمانی در ذهن ۲۸

۱-۶-۲ درک اختلاف بین همزمانی تکه تکه ای ، همزمانی و موازی سازی ۲۹

۱-۶-۳ موازی سازی Task ها ۳۰

۱-۶-۴ حداقل سازی نواحی بحرانی ۳۰

۱-۶-۵ درک قوانین برنامه نویسی موازی برای چند پردازنده های هسته ای ۳۱

۱-۷ آماده کردن NUMA و مقایسه پذیری بالاتر ۳۴

۱-۸ تصمیم راحت موازی سازی ۴۱

۱-۹ خلاصه ۴۳

۲ فصل دوم ۴۴

موازی سازی اجباری داده ها ٤٤

۲-۱ شروع TASK های موازی ۴۵

۲-۱-۱ کلاس System.Threading.Tasks.Parallel ۴۷

۲-۱-۲ Parallel.Invoke ۴۸

۲-۳ تبدیل کد های ترتیبی به کد های موازی ۵۸

۲-۲-۱ تشخیص Hotspot قابل موازی سازی ۵۹

۲-۲-۲ اندازه گیری موفق SpeedUP توسط اجرای موازی ۶۳

۲-۳ حلقه های موازی ۶۶

۲-۳-۱ Parallel.For ۶۶

۲-۳-۲ Parallel.Foreach ۷۵

۲-۴ تشخیص درجه موازی سازی مطلوب ۹۲

۲-۴-۱ ParallelOptions ۹۲

۲-۴-۲ شمارش نخ های سخت افزاری ۹۵

۲-۴-۳ هسته های منطقی هسته های فیزیکی نیستند ۹۶

۲-۵ استفاده از نمودار گانت برای تشخیص نواحی بحرانی ۹۹

۲-۶ خلاصه ۱۰۰

نتیجه‌گیری و پیشنهادات ۱۰۱

مراجع ۱۰۲

فهرست شکل ها :

شکل ۱-۱ ..... ۵
شکل ۱-۲ ..... ۹
شکل ۱-۳ ..... ۹
شکل ۱-۴ ..... ۱۴
شکل ۱-۵ ..... ۱۴
شکل ۱-۶ ..... ۱۵
شکل۱-۷ ..... ۱۶
شکل ۱-۸ ..... ۱۹
شکل ۱-۹ ..... ۲۰
شکل ۱-۱۰ ..... ۲۰
شکل ۱-۱۱ ..... ۲۲
شکل ۱-۱۲ ..... ۲۳
شکل ۱-۱۳ ..... ۲۳
شکل ۱-۱۴ ..... ۲۴
شکل ۱-۱۵ ..... ۳۲
شکل ۱-۱۶ ..... ۳۳
شکل ۱-۱۷ ..... ۳۷
شکل ۱-۱۸ ..... ۳۸
شکل ۲-۱ ..... ۴۵
شکل ۲-۲ ..... ۴۶
شکل ۲-۳ ..... ۴۷
شکل ۲-۴ ..... ۵۲
شکل ۲-۵ ..... ۵۷
شکل ۲-۶ ..... ۶۲
شکل ۲-۷ ..... ۶۴
شکل ۲-۸ ..... ۶۴
شکل ۲-۹ ..... ۷۴
شکل ۲-۱۰ ..... ۸۲
شکل ۲-۱۱ ..... ۹۶
شکل ۲-۱۲ ..... ۹۸
شکل ۲-۱۳ ..... ۹۹

فهرست تکه کد ها :

تکه کد ۲-۱ ۴۹
تکه کد ۲-۲ ۴۹
تکه کدی از لیست ۲-۳ ۶۱
تکه کدی از لیست ۲-۳ ۶۳
تکه کدی از لیست ۲-۳ ۶۹
تکه کدی از لیست ۲-۷ ۷۰
تکه کدی از لیست ۲-۵ ۷۰
تکه کدی از لیست ۲-۵ ۷۰
تکه کدی از لیست ۲-۵ ۷۱
تکه کدی از لیست ۲-۵ ۷۲
تکه کدی از لیست ۲-۵ ۷۲
تکه کدی از لیست ۲-۷ ۷۷
تکه کدی از لیست ۲-۷ ۷۷
تکه کدی از لیست ۲-۷ ۷۸
تکه کدی از لیست ۲-۷ ۷۸
تکه کدی از لیست ۲-۷ ۷۸
تکه کدی از لیست ۲-۹ ۸۰
تکه کدی از لیست ۲-۹ ۸۰
تکه کدی از لیست ۲-۱۱ ۸۴
تکه کدی از لیست ۲-۱۱ ۸۴
تکه کدی از لیست ۲-۱۲ ۸۶
تکه کدی از لیست ۲-۱۲ ۸۷
تکه کدی از لیست ۲-۱۲ ۸۷
تکه کدی از لیست ۲-۱۳ ۹۰
تکه کدی از لیست ۲-۱۳ ۹۰
تکه کدی از لیست ۲-۱۳ ۹۱
تکه کدی از لیست ۲-۱۵ ۹۳
تک کدی از لیست ۲-۱۵ ۹۴
تکه کدی از لیست ۲-۶ ۹۴
تکه کدی از لیست ۲-۷ ۹۷
تکه کدی از لیست ۲-۸ ۹۷

فهرست لیست ها (کد ها کلی) :

لیست ۱-۱ : اطلاعات نمایش داده شده بوسیله COREINFO نسخه ۲.۰ برای یک پردازنده INTEL مدل I7 .... ۳۹

لیست ۱-۲ : اطلاعاتی بوسیله وسیله COREINFO نسخه ۲.۰ در دو گره NUMA با INTEL CORE I7 .... ۴۰

لیست ۲-۱ : برنامه کنسول ساده ای است که مشکل ترتیب اجرا را در همزمانی کد در PARALLEL.INVOKE را نشان می دهد...... ۵۲

لیست ۲-۲ : نتایج اجرای کد موازی یکسانی که زمان های مختلفی برای اجرا نیاز دارد و در لیست ۲-۱ نشان داده شده است ۵۴

لیست ۲-۳ : سری ساده ای از کلید های AES و لیست مولد های MD5 .... ۵۹

لیست ۲-۵ : مثالی از تولید خروجی توسط تولید کننده کلید ها و لیست های MD5 که بصورت موازی اجرا می شوند........................ ۶۵

لیست ۲-۵ : تابع اصلی GENERATEAESKEYS با حلقه FOR ترتیبی و نسخه موازی خودش ۶۷

لیست ۲-۶ : تابع اصلی GENERATEMD5HASHES با حلقه FOR ترتیبی و نسخه موازی اش. ۶۷

لیست ۲-۷ : نسخه موازی دیگری از حلقه ترتیبی اصلی با استفاد از PARALLEL.FOREACH با یک تقسیم کننده خصوصی.......... ۷۶

لیست ۲-۸ : خروجی خطا یابی PARALLELPARTITIONGENERATEAESKEYS با پردازشگر QUAD-CORE ۷۹

لیست ۲-۹ : عیب یابی خروجی در نسخه بهینه PARALLELPARTITIONGENERATEAESKEYS با QUAD-CORE........... ۸۱

لیست ۲-۱۰: عیب یابی خروجی در نسخه بهینه PARALLELPARTITIONGENERATEAESKEYS ۸۱

لیست ۲-۱۱ : نسخه موازی از GENERATEMD5HASHES با استفاده از PARALLEL.FOREACH ۸۳

لیست ۲-۱۲ : نسخه جدیدی از تابع PARALLELFOREACHGENERATEMD5HASHES با احتمال خروج از حلقه ۸۵

لیست ۲-۱۳ : رخ دادن و مدیریت استثناء با تابع PARALLELFOREACHGENERATEMD5HASHES ۸۹

لیست ۲-۱۴ : خروجی خطایابی با دو استثناء یافته شده در مجموعه INNEREXCEPTIONS ۹۱

لیست ۲-۱۵ : تخصیص حداکثر درجه مطلوب موازی سازی برای حلقه PARALLEL.FOR ۹۲

چکیده :

شرکت های تولید کننده پردازشگر برای افزایش سرعت پردازنده مجبور به بالابردن فرکانس پردازشگر بودند یک راه افزایش ولتاژ مصرفی پردازنده بود که دارای نقاط ضعفی مانند افزایش دما و افزایش مصرف باطری نیز بود. از طرفی تولید کنندگان پردازنده به کمک برنامه نویسان پی به بیکاری زیاد پردازشگرها در زمان سویچ کردن فرایند ها و نخ ها شدند که حدود نیمی از زمان پردازش را به هدر می داد برای جبران حافظه کش را گسترش دادند اما به دلیل گران بودنش باز دچار محدودیت بودند بنابراین پردازنده هایی تولید کردند که بتواند پرازش موازی را (در ابتدا) در دو هسته به اجرا برسانند. نام این هسته ها هسته های سخت افزاری یا فیزیکی گذاشتند. اندک زمانی بعد فناوری ای برای رسیدن به پردازش موازی اما در سطح محدود تری و ارزان تر با نام ابر نخی یا Hyper-Threading ارائه کردند و نام آن را هسته های منطقی یا نخ های سخت افزاری گذاشتند. حال نوبت برنامه نویسان بود تا برنامه های برای استفاده از این فناوری های نوین بنویسند. برنامه نویسی موازی عنوانی است، که موضوعی گسترده در دنیای نرم افزار ایجاد کرد.

کلمات کلیدی :

برنامه نویسی موازی، پردازنده های چند هسته ای حافظه- مشترک، حافظه- توزیع شده، نخ های سخت افزاری، نخ های نرم افزری، قانون Amdahl ، قانون Gustafson ، همزمانی سبک، همزمانی سنگین، همزمانی تکه تکه ای، همزمانی، موازی سازی ، Task ، ناحیه بحرانی ، NUMA ، کد های ترتیبی ، کد های موازی ،Hotspot ، SpeedUP، درجه موازی سازی مطلوب ، ParallelOptions و نمودار گانت .

مقدمه :

ساده ترین شیوه موازی سازی در قالب Task ها صورت می گیرد، در هر Task تابع یا قطعه کدی نوشته می شود و سپس بوسیله Delegate ای که کار مدیریت Task ها را بر عهده دارد، این Task ها بصورت موازی بسته به هسته های منطقی در دسترس اجرا می شوند. روش های بسیاری برای موازی سازی وجود دارد مانند استفاده از کلاس Parallel.For یا Parallel.ForEach که در جای خود کاربرد های مختص به خودشان را دارند. همشه الگریتم های ترتیبی را نمی توان به الگریتمی موازی تبدیل کرد چرا که کد های ترتیبی ای هستند که اجرای کد های دیگر نیاز به تکمیل شدن آن ها دارد. بسته به الگریتم درصدی از آن را می توان موازی کرد. قبل از موازی سازی باید موازی سازی را ذهنتان طراحی کنید .

۱ فصل اول

برنامه نویسی Task-bease

مباحثی که در این فصل مطرح می شود :

- ✓ کار با حافظه-مشترک[1] و چند هسته ای ها[2]
- ✓ درک اختلاف بین حافظه-مشترک در چند هسته ای ها با سیستم های با حافظه توزیع شده[3]
- ✓ کار با برنامه نویسی موازی و چند هسته ای در معماری حافظه-مشترک[4]
- ✓ درک نخ های نرم افزاری [5]و نخ های سخت افزاری[6]
- ✓ درک قانون Amdahl
- ✓ درک قانون Gustafson
- ✓ کار با مدل همزمانی سبک وزن
- ✓ ایجاد موفق طرح های task-bease
- ✓ درک اختلاف های بین همزمانی[7] ، همزمانی تکه تکه شده[8] و موازی سازی[9]
- ✓ task های موازی و حداقل کردن نواحی بحرانی[10]
- ✓ درک قوانینی برای برنامه نویسی موازی و CPU هایی با معماری چند هسته ای

---

[1] shared-memory
[2] multicores
[3] distibute memory system
[4] shared-memory architectures
[5] softwear threads
[6] hardware threads
[7] Concurrency
[8] interleaved concurrency
[9] parallelism
[10] critical sections

✓ آماده سازی معماری های NUMA

این فصل برنامه نویس task-bease به شما معرفی می شود که به شما امکان موازی سازی در برنامه هایتان را می دهد . در موازی سازی استفاده از معماری های پردازنده های چند هسته ای جدید با حافظه مشترک الزامیست . همچنین در این فصل مدل های همزمانی سبک وزن توضیح داده می شود و ارتباطشان بین همزمانی و موازی سازی شرح داده می شود . در ادامه اطلاعاتی جهت یادگیری ۱۰ فصل آینده آورده شده که به شما پس زمینه های فکری لازم جهت مطالعه ۱۰فصل باقی مانده را می دهد .

۱-۱ کار با پردازنده های چند هسته ای SHARED-MEMORY

انتشارات Herb Sutter در سال ۲۰۰۵ مقاله ای را در مجله Dr.Dobb را با نام "بیش از یک نهار رایگان : سیری اساسی درباره همزمانی در نرم افزار" منتشر کرد (www.gotw.ca/publications/concurrency-ddj.htm). در این مقاله او در این باره صحبت می کند که یکی از نیازها در شروع توسعه نرم افزار، در نظر گرفتن تعریفی کامل و پیوسته برای عملکرد همزمانی و کسب توان عملیاتی بالاتر برای ریز پردازنده است. ریز پردازنده ها بجای افزایش فرکانس ساعت[1] تعداد هسته های پردازشگر را می توانند اضافه کنند . توسعه دهندگان نرم افزار روی کارایی عملکرد free-lunch که با افزایش فرکانس ساعت فراهم می شود تکیه نمی کنند . اکثر کامپیوتر های امروزی لااقل یک پردازشگر dual-core دارند. اما استفاده از پردازشگر های quad-core و octal-core به ترتیب با چهار و هشت هسته، بر روی Servere های ایستگاه های کاری[2] پیشرفته و حتی در سطح بالایی از کامپیوتر های همراه رواج دارد . اکثر هسته ها در یک پردازنده درست در نزدیکی گوشه ها قرار دارد. ریز پردازنده های پیشرفته معماری چند هسته ای جدیدی را ارائه می دهند. بنابراین نکته ای بسیار مهم در طراحی نرم افزار این است که نرم افزار را طوری طراحی کنید که کد نوشته شده بهترین استفاده را معماری داشته باشد. انواع مختلفی از برنامه های اجرایی با Visual C# 2010و NET. Freamwork 4 بر روی یک یا چند (بیشتر) واحد پرداش مرکزی[3] (CPU ها) تولید شده اند که ریز

---

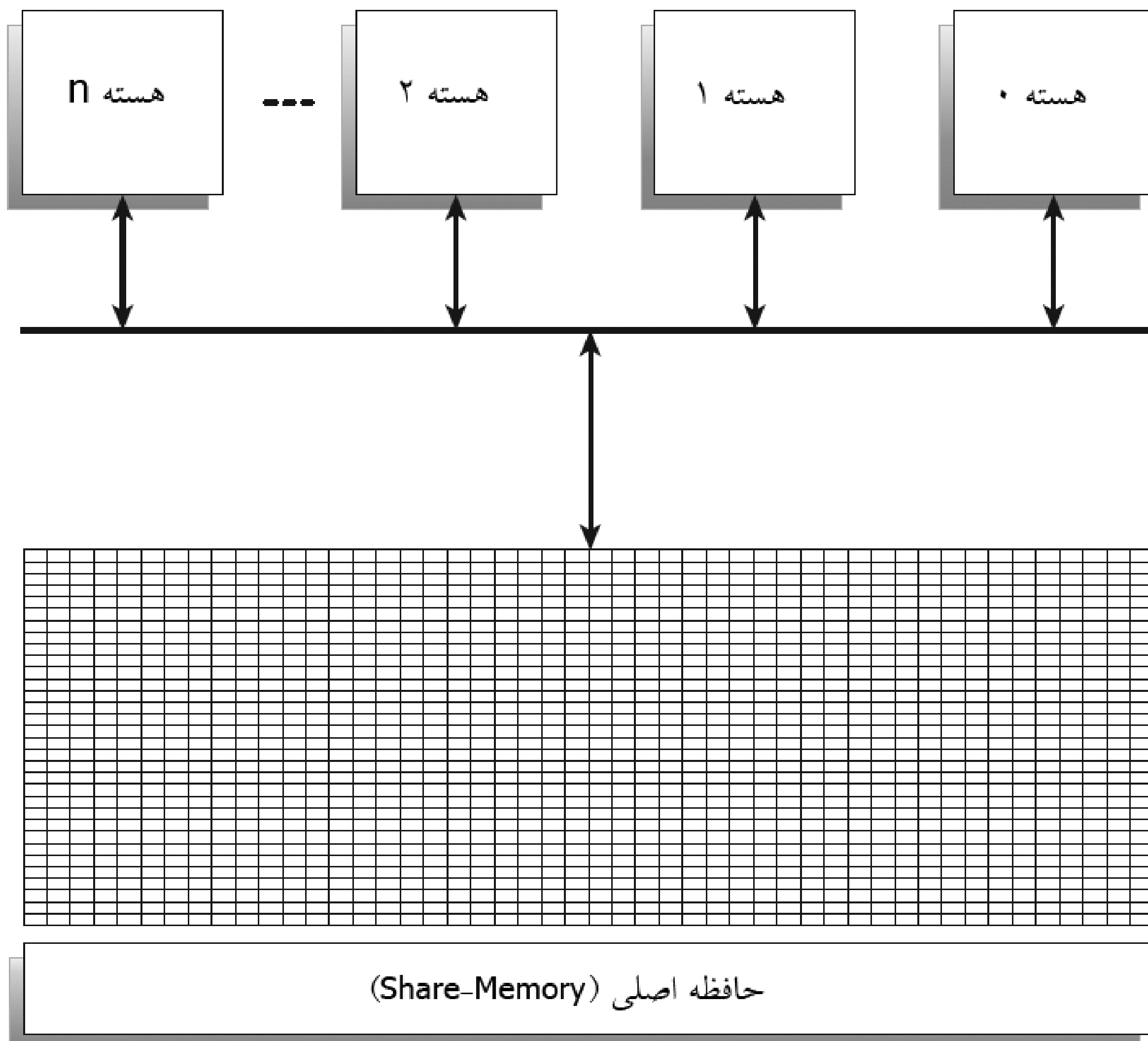
[1] Clock frequency

[2] Workstations

[3] Central Process Unit

پردازنده اصلی می باشند. هر یک از این ریز پردازنده ها تعداد متفاوتی هسته دارند که قابلیت اجرای دستور العمل ها را به آنها می دهد .

به این شکل می توانید فکر کنید که هسته ریز پردازنده مانند تعدادی ریز پردازنده است که از داخل به هم وصل شده اند وهمه هسته ها به حافظه اصلی دسترسی دارند ، شکلی از آن در شکل ۱-۱ نمایش داده شده است. این معماری shared-memory جند هسته ای می باشد ودر آن حافظه های مشترک در این شیوه به آسانی می توانند محدودیت های عملیاتی[1] را مدیریت کنند .



شکل ۱-۱

[1] performance bottleneck

ریز پردازنده های چند هسته ای با تعداد زیادی از ریز معماری های پیچیده مختلف طراحی شده اند تا توانایی اجرای موازی را داشته باشند و توان عملیلتی کل را افزایش دهند و همچنین محدودیت های عملیاتی بلقوه را کاهش دهند . در برخی زمان ها سعی کرده اند توان مصرفی و گرمای تولید شده در ریز پردازنده های چند هسته ای را کاهش بدهند بنابراین در بسیاری از ریز پردازنده های مدرن پیشرفته می توان فرکانس کار برای هر هسته را مطابق با بار کاری کاهش یا افزایش داد و حتی قادرند در حالتی که از هسته استفاده نمی شود آن را به حالت Sleep ببرند .

Windows 7 و windows Servere 2008 R2 از ویژگی جدیدی بنام CoreParking پشتیبانی می کنند، این ویژگی زمانی فعال می شود که تعداد هسته های بلا استفاده بالا رود سپس CoreParking هسته ها را به حالت Sleep می برد . هنگامی که هسته ای مورد نیاز باشد سیستم عامل هسته بخواب رفته را بیدار می کند . ریز پردازنده های پیشرفته با فرکانس هایی پویا برای هر کدام از هسته هایشان کار می کنند . چرا که هسته ها با فرکانس ثابتی کار نمی کنند و همچنین عملکرد توالی دستورالعمل هایشان با هم فرق دارد ضمنا نمی توان آن را پیش بینی کرد برای مثال Intel Turbo Boost[1] فناوری است که فرکانس هسته های فعال را زیاد می کند . فرآیند افزایش فرکانس برای هسته را Overclocking می نامند .

اگر هسته ای تحت بارکاری[2] سنگینی قرار بگیرد این فناوری اجازه پخش فرکانس زیاد، را به هسته هایی که در حالت IDLE[3] قرار دارند را می دهد . اگر هسته هایی که زیر بار کاری سنگینی اند زیاد باشد ، در بالاترین فرکانس اجرا می شوند، اما بیشترین موفقیت توسط یک هسته بدست می آید. ریز پردازنده تمام هسته را به حالت OverClock نگه نمی دارد. به این دلیل که توان مصرفی و درجه حرارت به سرعت افزایش می یابد . زمانی که بار کاری زیادی اعمال می شود میانگین فرکانس ساعت را برای تمامی هسته

---

[1] تکنولوژی است که در پردازنده‌های نسل جدید اینتل مورد استفاده قرار گرفته ، این تکنولوژی سرعت بیشتری به پردازنده می‌دهد. به صورت پویا فرکانس پردازنده را در صورت نیاز افزایش می‌دهد. همچنین عملکرد این قسمت خودکار است و هنگامی که شما نیاز بیشتری برای پردازش دارید دست به کار خواهد شد. به عبارتی استفاده از این تکنولوژی باعث می‌شود که کاربر بتواند بیش از هفت فایل سنگین را بدون هیچ مشکلی باز کند. نحوه کار هم به این صورت است که پردازنده انرژی موردنظر برای باز کردن فایل‌های سنگین را از بخش‌های دیگر به طور اتوماتیک دریافت می‌کند. تمامی پردازنده های Core i5 از این فناوری پشتیبانی می کند، اما برخلاف آن تراشه های Core i3 فاقد این تکنولوژی می باشند.

[2] Workload

[3] بیکار

های حداقل نگه می دارد بجز برای یک هسته تا به موفقیت لازم که اتمام کارش است برسد . بنابراین در شرایط خاص کد هایی می توانند در در بالاترین فرکانس نسبت به دیگر کد ها اجرا شوند و نمی توان کارایی واقعی را دراین رقابت اندازه گیری کرد .

۱-۱-۱ اختلاف پردازشگر های چند هسته ای حافظه- مشترک [۱] با سیستم های حافظه- توزیع شده[۲]

سیستم های کامپیوتری با حافظه توزیع شده ، ترکیب از تعداد بسیاری از ریز پردازنده ها با حافظ خصوصی خودشان است به عنوان نمونه شکل ۱-۲ را نگاه کنید. هر ریز پردازنده می تواند در یک کامپیوتر مختلف با انواع گوناگونی از کانال های ارتباطی بین آنها باشد . مثالی از کانال های ارتباطی، شبکه های سیمی می باشد ، اگر کار[۳]ی که در یک ریزپردازنده در حالا اجراست ، تقاضایی برای دسترسی به داده ای از راهی دور را بدهد آن ریز پردازنده باید از میان کانال های حتما ارتباطی نظیر به نظیر را برقرار کند. از پرتکل های رایجی که برنامه های موازی از آن سود می برند پرتکل واسط عبور پیام یا MPI[٤] می باشد که قابل اجرا شدن بر روی سیستم های کامپیوتری با حافظه توزیع شده (Distributed Memory) می باشد . ممکن است از مزیت پرتکل MPI در پردازنده های چند هسته ای Shared-Memory با C# و NET. Framework نیز استفاده شود .تمرکز اصلی MPI بر روی توسعه برنامه های کاربردی در اجرا روی کلاستر[۵] ها است . بنابراین در جایی که همه هسته ها بدون ارسال پیام به حافظه دسترسی دارند ، استفاده از پردازنده چند هسته ای با حافظه تقسیم شده[٦] نیاز نیست چرا که OverHead[۷] بزرگی را اضافه می کند .

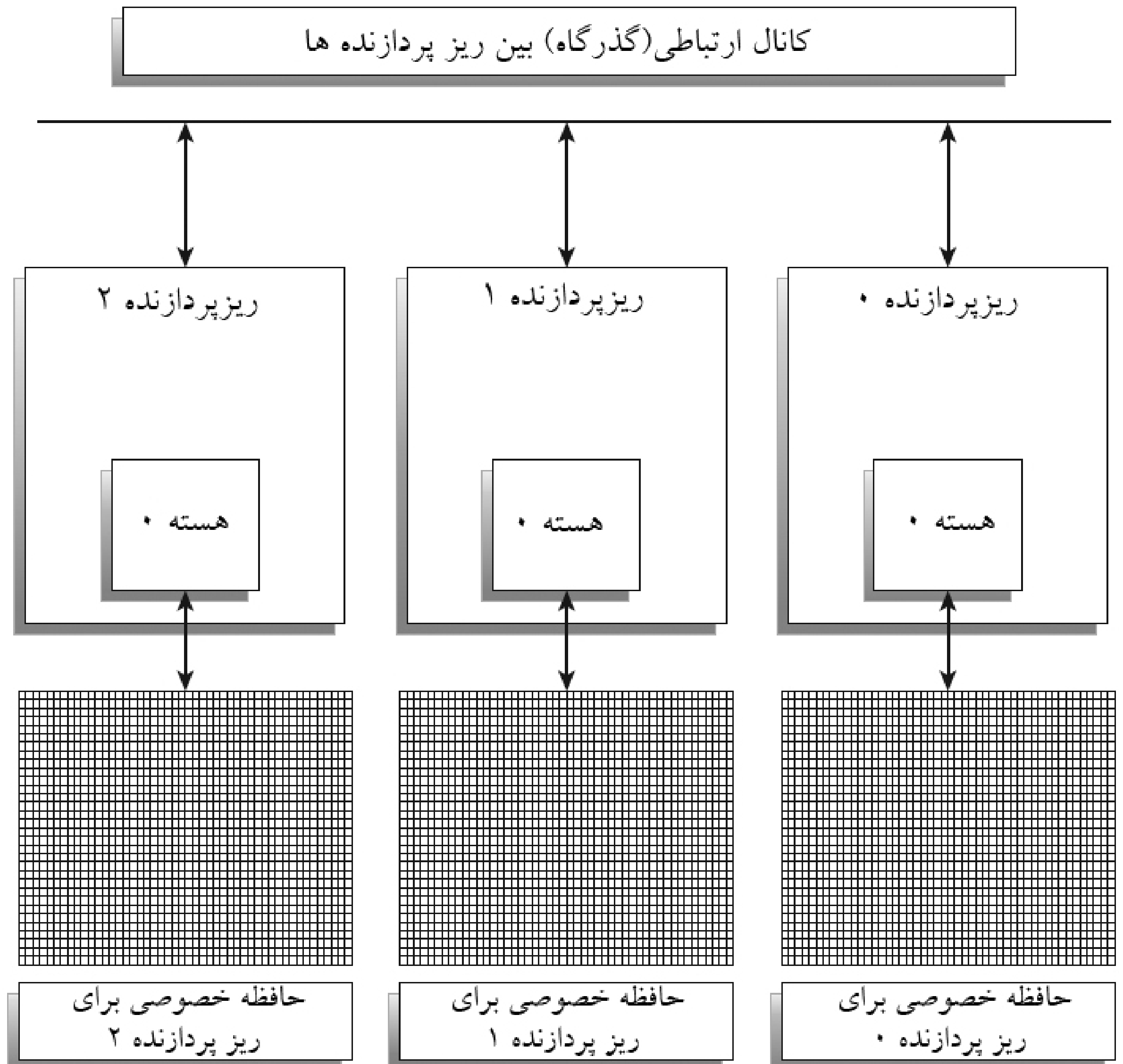
در شکل ۱-۳ سیستم های کامیوتری با حافظه توزیع شده در سه ماشین نشان داده شده است . هر ماشین یک ریز پردازنده چهار هسته ای دارد و دارای معماری Shared-Memory برای این هسته ها می باشد. در

---

[1] Sared-Memory Multicore
[2] Distributed-Memoy System
[3] Job
[4] Message Passing Interface
[5] Clusters
[6] Share-Memory MultiCore

[۷] مدت زمان پردازش مورد نیاز برای اتمام کاری معیین .

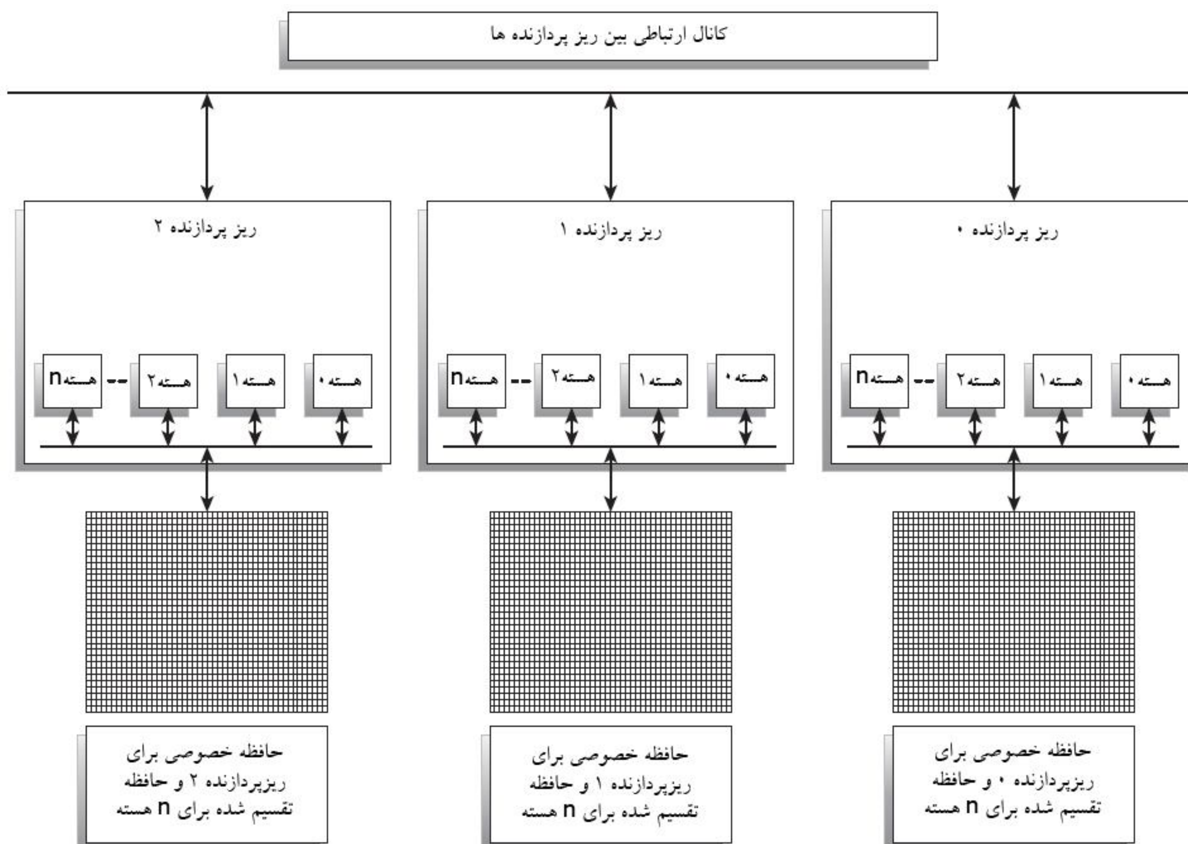
این شیوه حافظه خصوصی هر ریز پردازنده به عنوان حافظه تقسیم شده (Shared-Memory) برای چهار هسته عمل می کند. در سیستم هایی با حافظه توزیع شده ، اجباراً باید به توزیع داده فکر کنید زیرا هر پیامی با بازیابی اطلاعات از راه دور معرف یک تاخیر مهمی است[1] . زیرا با اضافه کردن ماشین[2] های جدید تعداد ریز پردازنده ها برای سیستم افزایش می یابد . سیستم های با حافظه توزیع شده در مقایسه پذیری بزرگ ارائه می شود از سیستم های Distributed-Memory در مقیاس های بزرگی ارائه می شود به همین خاطر ماشین (node) های جدیدی را می توانید اضافه کنید تا تعداد ریز پردازنده ها برای سیستم افزایش یابد .



شکل ۱-۲

[1] منظور نویسنده تاخیر زمانی ارسال پیام نا زمان برگشت پیام می باشد .

[2] Node



شکل ۱-۳

۱-۱-۲ برنامه نویسی موازی و برنامه نویسی چند هسته ای

کدهای ترتیبی[1] در ابتدا دستورالعمل ها را یکی پس از دیگری اجرا می کردند و از مزیت های چند هسته ای سودی نمی برند چرا که توالی دستورالعمل ها برای اجرا تنها یک هسته را در دسترس داشتند . اگر ویژگی های جدید ارائه شده در NET Framework. را استفاده نکنید و کار را به هسته های زیادی تقسیم نکند، در این صورت کدی ترتیبی به حالت عادی در Visual C# 2010 نوشته اید که از مزیت چند هسته ای بودن پردازنده استفاده ای نمی کرده اید. در کد های ترتیبی موجود موازی سازی به صورت خودکار اعمال نمی شود. برنامه نویسی موازی نوعی برنامه نویسی ای است که در آن کد از مزیت های اجرای موازی توسط امکانات ارائه شده بوسیله سخت افزار اساسی[2] برخوردار می شود .

---

[1] Sequential Code
[2] Under Hardwear

در برنامه نویسی موازی دستور العمل های زیادی در یک زمان اجرا می شوند . همانطور که شرح داده شده انواع گوناگونی از معماری های موازی وجود دارد و مطمئناً نیازمندی هایی که در این باره در کتاب برای کامل کردن مطلب اختصاص داده شده را تحلیل می کنیم .

برنامه نویسی چند هسته ای[۱] نوعی از برنامه نویسی است که در آن کد از اجرای تعداد بسیاری از دستور العمل ها بصورت موازی در چند هسته سود می برد. کامپیوتر های چند پردازنده ای و چند هسته ای بیشتر از یک هسته پردازنده را در یک ماشین را ارائه می کنند از این رو هدف این است که در زمان کمتر، توزیع کار در هسته های در دسترس، بیشتری انجام گیرد. پردازنده های امروزی توانایی اجرای دستور اعمل های یکسان روی داده های مختلف را دارند و چنین موردی را Michael J.Flyn در طبقه بندی فلاین[۲] در ۱۹۶۶ به عنوان SIMD [۳] دسته بندی کرد . در این شیوه با استفاده از ریز پردازنده های برداری[۴] مطمئناً زمان مورد نیاز برای اجرای الگریتم کاهش می یابد .

این کتاب دو ناحیه برنامه نویسی موازی با جزئیات بزرگی را پوشش می دهد : برنامه نویسی چند هسته ای حافظه- مشترک و استفاده از توانایی پردازش برداری . هدف کلی کاهش زمان اجرای الگریتم ها و افزایش قدرت پردازنده هاست، به نحوی مناسب، که شما را قادر به اضافه کردن مشخصاتی جدید به نرم افزار موجود کند .

۱-۲ درک نخ های سخت افزاری[۵] و نخ های نرم افزری[۶]

ریز پردازنده های چند هسته ای بیش از یک هسته فیزیکی[۷] دارند و بطور واقعی هر واحد پردازنده (هسته) مستقل بوده و امکان اجرای دستورالعمل ها را همزمان فراهم می کند. به منظور دستیابی به هسته های

---

[1] Multicore Programming
[2] Flyn Taxonomy
[3] Single Instrunction Multiple Data
[4] Vector Processor
[5] Hardware Threads
[6] Software Threads
[7] Physical Core

فیزیکی چند گانه[1] لازم است فرآیندهای زیادی اجرا شوند و یا بیش تر از یک نخ در یک پردازنده اجرا شود تا بتوانیم کد های چند نخی شده را ایجاد کنیم .

با این حال ، هر هسته فیزیکی بیش از یک نخ سخت افزاری را می تواند ارائه دهد و نیز به عنوان هسته منطقی[2] یا پردازنده منطقی[3] شناخته می شود. ریزپردازنده ها با فناوری اَبَر نخی اینتل[4] با نام HT یا HTT معرفی شده اند که حالت های معماری بسیاری را در هسته فیزیکی دارند . برای مثال ریز پردازنده هایی با چهار هسته و با HT دو برابر ، دارای معماری ای در هسته فیزیکی اند، که ۸ نخ سخت افزاری را شامل می شوند .این تکنیک را چند نخی متقارن[5] می نامند و برای اضافه کردن حالاتی به معماری کاربرد دارد که برای افزایش و بهینه سازی اجرای موازی در سطح دستورالعمل ریز پردازنده بکار می رود . SMT فقط به دو نخ سخت افزاری در یک هسته محدود نشده است ،شما چهار نخ سخت افزاری در یک هسته نیز می توانید داشته باشید این مطلب به این معنا نیست که هر نخ سخت افزاری در یک هسته فیزیکی ارائه می شود. SMT قادر به بهبود عملکرد کد چند نخی تحت سناریویی خاص است. در فصل های بعدی چندین مثال را برای بهبود عملکرد سیستم فراهم شده است .

هر برنامه ای که در ویندوز اجرا می شود یک فرآیند[6] است . هر فرآیند یک نخ را ایجاد و سپس اجرا می کند که این نخ به نخ نرم افزاری معرف می باشد و با نخ سخت افزاری شرح داده شده فرق دارد هر فرآیند دست کم یک نخ که شامل نخ اصلی[7] است داراست . زمانبند سیستم عامل منابع پردازشی موجود و در دسترس را به طور منصفانه بین همه فرآیند ها ونخ هایی که باید اجرا شوند تقسیم می کند . زمانبندِ ویندوز زمان پردازش را به نخ نرم افزاری واگذار می کند . هنگامی که زمانبند ویندوز روی یک ریز پردازنده چند هسته ای اجرا شود مجبور است زمان را به یک نخ سخت افزاری که توسط هسته فیزیکی پشتیبانی می شود واگذار کند تا هر نخ نرم افزاری که دستورالعملی از آن نیاز به اجرا دارد اجرا شود . به عنوان یک مقایسه ،

---

[1] Multiple Phisical Core
[2] Logical Core
[3] Logical Process
[4] Intel Hyper-Threading
[5] Simultaneous Multi Threading (SMT) – ارسال و دریافت همزمان به وسیله تکنیک چند نخی
[6] Process
[7] Main Thread

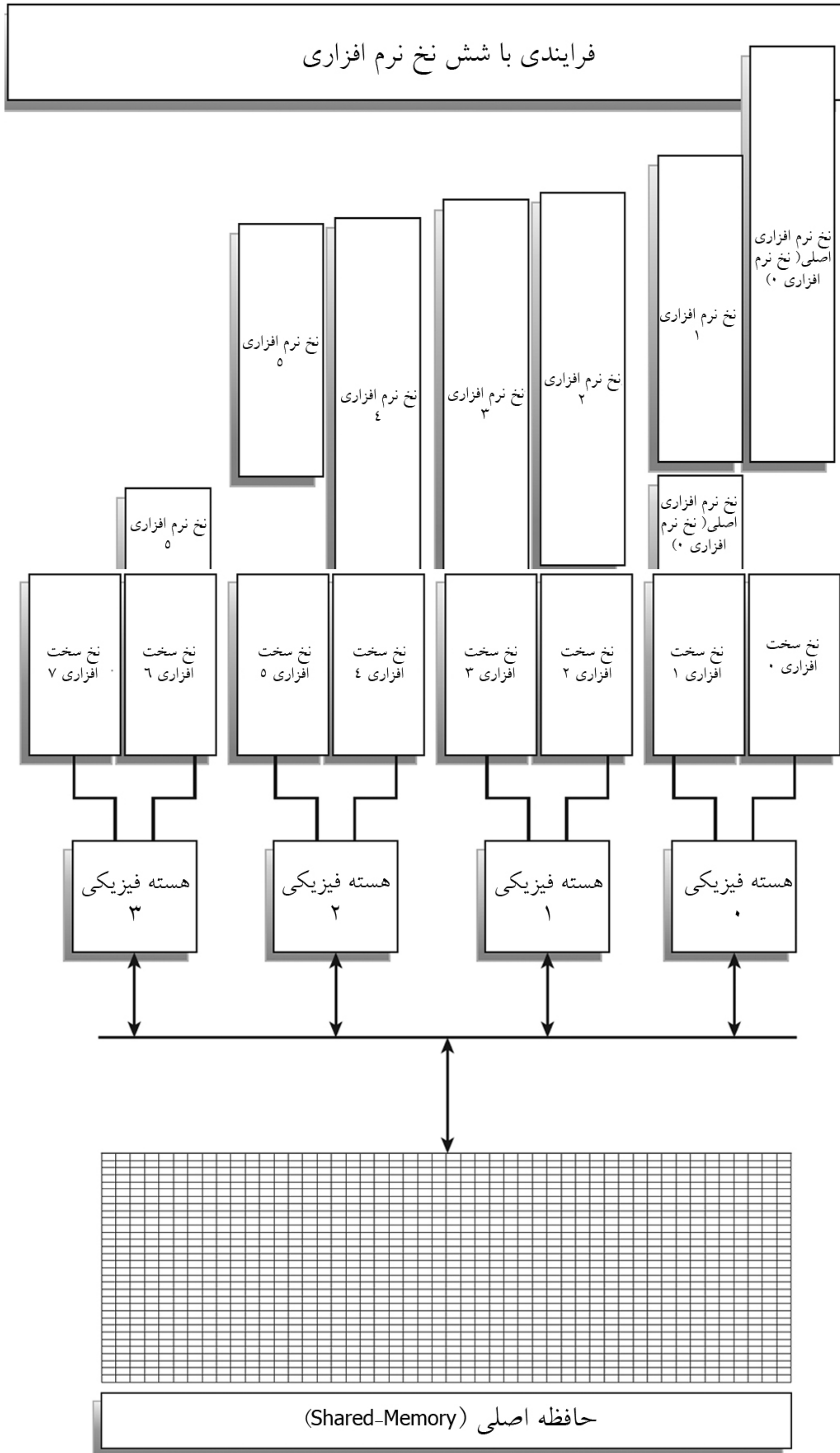
مسیر باریک شنای شناگری را نخ سخت افزاری در نظر بگیرید و نخ نرم افزاری را به عنوان خود شناگر تصور کنید .

هر نخ نرم افزاری فضای حافظه خصوصی که مختص خودش است را با فرآیند والدش سهیم می شود. با اینکه Register , Stack و فضای ذخیره سازی محلی و مخصوص به خود دارد.

ویندوز هر نخ سخت افزاری را به عنوان پردازنده ای منطقی و قابل زمانبندی تشخیص می دهد. هر پردازنده منطقی کد یک نخ نرم افزاری را اجرا می کند فرآیندی که کد را در نخ های سخت افزری چندگانه اجرا می کند قادر است از برتری های نخ های سخت افزاری و هسته های فیزیکی برای اجرای دستورالعمل ها بصورت موازی استفاده کند . شکل ۱–٤ نخ های نرم افزاری که در حال اجرا بر روی نخ های سخت افزاری و هسته های فیزیکی اند را نشان می دهد . زمانبند ویندوز قادر به تصمیم گیری برای واگذاری مجدد یک نخ نرم افزاری به نخ سخت افزاری دیگر برای حفظ تعادل بار کاری بوسیله نخ سخت افزاری است، زیرا معمولا تعداد کثیری نخ نرم افزاری منتظر گرفتن پردازنده [دریافت زمانی برای پردازش شدن] هستند .ایجاد تعادل در بار کاری[1] به نخ های دیگر این امکان را می دهد تا بوسیله سازماندهی منابع در دسترس دستورالعمل هایشان را اجرا کنند . در شکل۱–۵ Windows Task Manager با هشت نخ سخت افزاری نشان داده شده است (هسته های منطقی[2] و بار های کاری شان) .

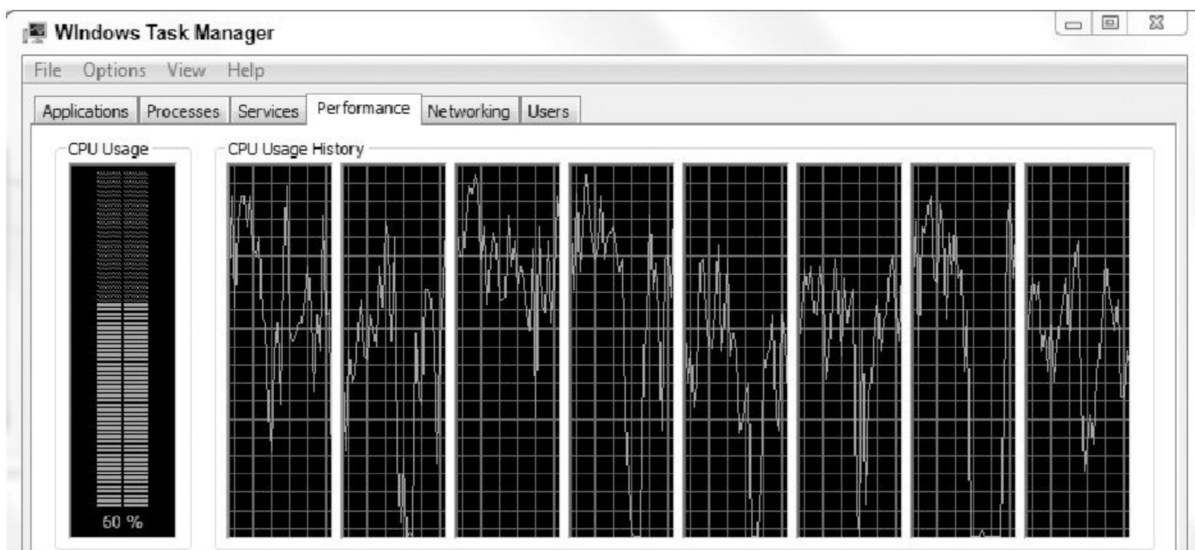
load Balancing به تکرار کار توزیع نخ های نرم افزاری بین نخ های سخت افزاری اشاره دارد . به طوری که بارکاری در سرتاسر نخ های سخت افزاری به طور عادلانه ای تقسیم شده باشد. دستیابی کامل به ایجاد تعادل در موازی سازی برای برنامه کاربردی بستگی به بار کاری ، تعداد نخ های نرم افزاری و سخت افزاری در دسترس و نظم در Load-Blancing دارد . Windows Monitor Resource و Windows Task Manager تاریخچه استفاده CPU از نخ های سخت افزاری به شکل گرافیکی را نشان می دهد. برای مثال اگر شما یک پردازنده یا چهار هسته فیزیکی و هشت نخ

[1] Load Balancing
[2] Logic Cores



شکل ۱-٤

سخت افزاری داشته باشید. ابزار های موجود در این نرم افزار ها هشت شکل گرافیکی مستقل ر ا نشان می دهند .



شکل ۱-۵

ویندوز صد ها نخ نرم افزاری را به وسیله اختصاص تکه ی از زمان پردازش با هر نخ سخت افزاری در دسترس اجرا می کند. با استفاده از Windows Resource Monitor تعداد نخ های نرم افزاری به یک فرآیند خاص را در سربرگ OverTap می توان مشاهده کرد. پنل CPU نام Image را برای هر فرآیند به همراه تعداد نخ های نرم افزار وابسته را در ستون Threads نمایش می دهد . همان طوری که در شکل ۱-٦ می بینید فرآیند VLC.EXE ۳۲ نخ نرم فزاری دارد .

CorePacking بر توانایی هسته ویندوز مدیریت می کند و تکنولوژی ای برای زمانبندی هسته می باشد که هدف آن بهبود بخشیدن کارایی انرژی مصرفی در سیستم های چند هسته ای است . به طور مدارم دنبال کردن بار کاری وایسته یه هر نخ سخت افزاری می باشد که این خود به نخ های سخت افزاری دیگر وابسته است . و تصمیم به قراردن برخی از آنها در حالت Sleep می باشد .

Core Parking به شکلی پویا تعداد نخ های سخت افزاری که در بار کاری بطور اساسی استفاده می شوند را مقایسه می کند، هنگامی که بار کاری برای یک نخ سخت افزاری کمتر از یک حد خاصی باشد، الگریتم Core Parking سعی در کاهش تعداد نخ های سخت افزاری ای دارد که در Parking یکسان نخ های سخت افزاری در سیستم استفاده شده است .

Resource Monitor

File Monitor Help

Overview | CPU | Memory | Disk | Network

CPU — 3% CPU Usage — 90% Maximum Frequency

Image	PID	Description	Status	Threads	CPU	Average CPU
perfmon.exe	5576	Resource and Per...	Running	19	0	0.72
vlc.exe	1456	VLC media player	Running	32	1	0.50
System interrupts	-	Deferred Procedur...	Running	-	1	0.48
WINWORD.EXE	7660	Microsoft Office W...	Running	9	0	0.25
SynTPEnh.exe	4512	Synaptics TouchP...	Running	6	0	0.12
dwm.exe	4236	Desktop Window	Running	6	0	0.12
csrss.exe	568	Client Server Runti...	Running	11	0	0.10
audiodg.exe	4860		Running	5	0	0.10
explorer.exe	4284	Windows Explorer	Running	36	0	0.08
SnippingTool.exe	8164	Snipping Tool	Running	15	0	0.06

شکل ۱-۶

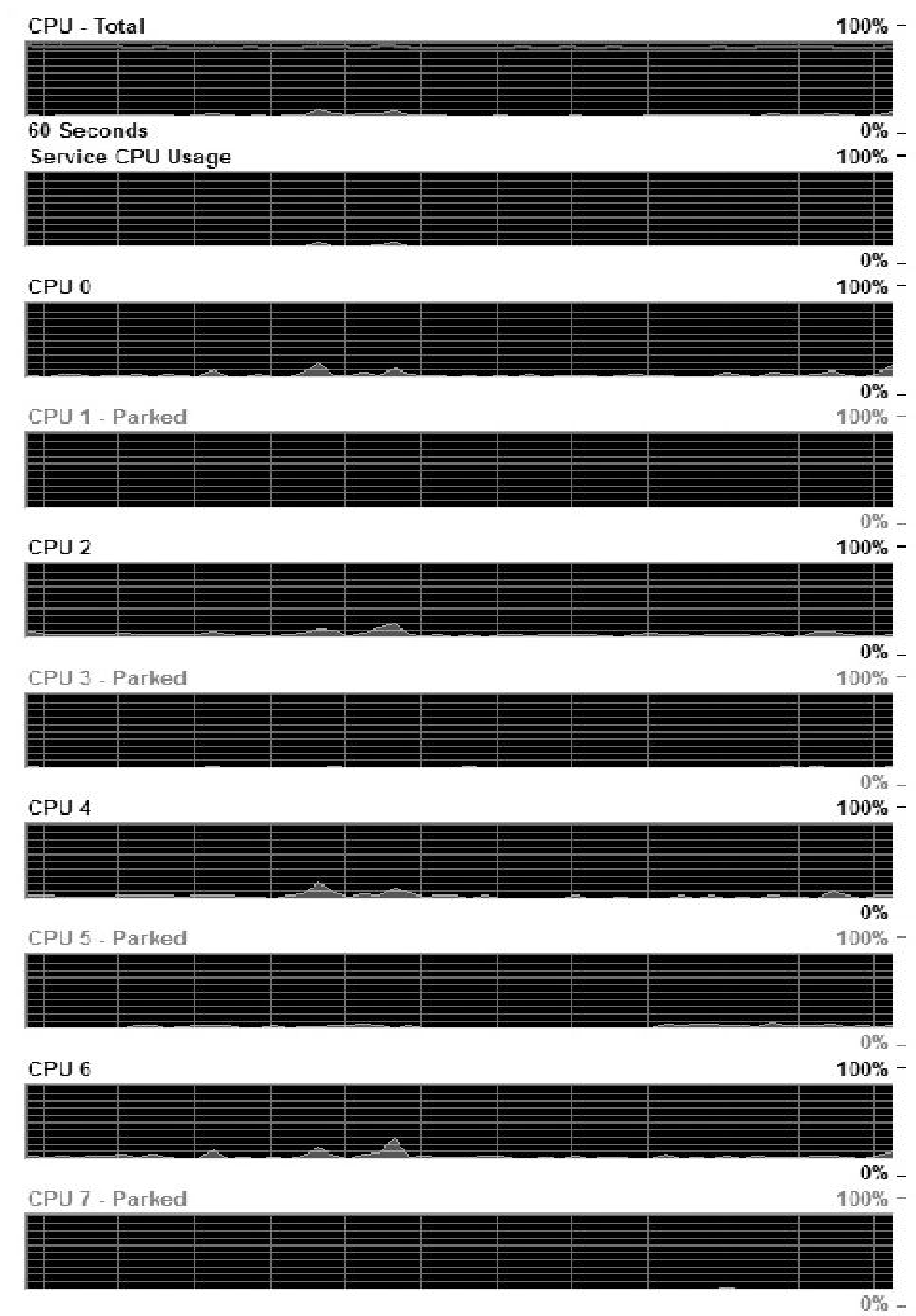
برای افزایش کارایی این الگریتم هنگامی که زمانبندی نخ های سخت افزاری انجام شد ، زمانبند هسته برای Unparked (از حالت Sleep خارج کردن نخ های سخت افزاری) کردن نخ های سخت افزاری الویت بالاتری در نظر می گیریم . یکی از اهداف زمانبندی هسته[1] توقف[2] نخ های سخت افزاری که در حالت IDLE در آمدند می باشد. این کار باعث انتقال به حالت کم مصرف Low-Power می شود .

به طرزی هوشمندانه Core Parking سعی در زمانبندی کار بین نخ های در حال اجرایی دارد که روی چند نخ سخت افزاری در هسته فیزیکی ای یکسان قرار دارند و بر روی سیستم های که شامل پردازنده هایی با تکنولوژی HT می شوند . زمانبند، کار کاهش قدرت مصرفی را انجام می دهد .

Windows Server 2008 R2 کاملا از فناوری Core Parking پشتیبانی می کند همچنین Windows 7 نیز از این الگریتم استفاده می نماید و دارای زیر ساخت هایی است که باعث ایجاد تعادل در عملکرد پردازنده ما بین نخ های سخت افزاری با ریزپردازندهایی که از فناوری HT بهره می برند می شود .در شکل ۱-۷ برنامه Windows Resource monitor فعالیت هشت نخ سخت افزاری را نشان می دهد بطوری که چهار عدد از آن ها در حالت Parked به سر می برند.

[1] Kernel Scheduler
[2] Parked

با وجود اینکه در شکل ۱–۷ تعداد چهار عدد نخ سخت افزاری در حالت Parked مشاهده می شود ولی تعداد نخ های سخت افزاری که توسط توابع NET Framework. برگشت داده می شود تعداد کل(هشت عدد) می باشد ، نه تعداد آنهای که در حالت Unparked می باشند. Core Parking فناوری است که محدودیتی برای تعداد نخ های سخت افزاری که برای اجرای نخ های نرم افزاری یک فرایند در دسترس اند ندارد.



شکل ۱–۷

بطور حتم تحت بار کاری ای ، سیستمی با هشت نخ سخت افزار قادر به تغییر سیستمی با دو نخ سخت افزاری است و این در زمانی رخ می دهد که بار کاری کم شده و نخ های سخت افزاری بیشتری برای صرف جویی توان مصرفی به حالت Sleep روند. در برخی موارد Core Paking تاخیر های زمانی را برای زمانبندی نخ های سخت افزاری بسیاری که سعی در اجرای کد بصورت موازی دارند را اضافه می کند. بنابراین در هنگام اندازه گیری عملکرد موازی سازی ، در نظر گرفتن تاخیر های کل اهمیت زیادی دارد .

۱-۳ درک قانون AMDAHL

اگر قصد استفاده از مزیت های پردازنده های چند هسته ای را در اجرای دستورالعمل های زیادی در زمان کوتاه دارید، لازمه این کار، چند تکه کردن کد در توالی موازی است اما اکثر الگریتم ها برای اجرای شدن نیاز به اجرا کد در ترتیبی خاص دارند . بعنوان مثال در ابتدا تکه کد های زیادی بصورت موازی شروع شده و سپس از جمع آوری حاصل آنها نتیجه بدست می آید . که برای اجرای موازی چند تکه می شود اگر حاصل از مجموعه ای از کد های ترتیبی به وجود آید از مزیت موازی سازی استفاده نشده است ، اگر از چنین الگریتم های پیوسته در درصد بالای از کد های ترتیبی استفاده شود احتمالا با کاهش کارایی مواجه می شوید.

Gene Amdahl یک معمار کامپیوتر معروف است وی تحقیقاتی درباره حداکثر بهبود عملکردی که از سیستم های کامپیوتری در زمانی که تنها کسری از سیستم بهبود یافته را انجام داده او مشاهداتش برای تعریف قانون Amdahl استفاده کرده و با در نظر گرفتن فرمول زیر سعی کرد تا حداکثر بهبود عملکردی نظری (معروف به SpeedUP) را با استفاده از پردازنده های چندگانه پیشبینی کند. همچنین با الگریتم های موازی که در ریز پردازنده های چند هسته ای اجرا می شوند نیز بکار برده شده است .

P : تعدا بخش های از کد که بطور کامل موازی سازی شده

N : حداکثر واحد های اجرایی در دسترس (پردازنده های فیزیکی)

بر طبق این فرمول ، اگر شما الگریتمی داشته باشید که ۵۰ ٪ از کل آن بطور موازی اجرا شود (P = 50 %) ، حداکثر SpeedUP با استفاده از یک ریز پردازنده دو هسته ای ۱.۳۳ است . در شکل۱-۸ الگریتمی با

۱۰۰۰ واحد کاری به ۵۰۰ واحد کاری ترتیبی و ۵۰ واحد کاری موازی تقسیم شده است ، اگر در نسخه های ترتیبی اجرای کامل ۱۰۰۰ ثانیه زمان ببرد ، اجرای موازی همان کد در کمتر از ۷۵۰ ثانیه رخ می دهد .

ن)

که حداکثر SpeedUp برای الگریتم فوق روی ریز پردازنده ای با ۸ هسته فیزیکی واقعا کمتر از ۱.۷۷ x است . بنابراین هسته های فیزیکی اضافه شده باعث می شوند تا کد در کمتر از ٥٦٥.٥ ثانبه اجرا شوند .

زمان)

شکل۱-۹ حداکثر سرعت الگریتم بر اساس ۱ الی ١٦ هسته فیزیکی نشان می دهد. همانطور که در شکل می توانید ببینید، SpeedUP خطی نیست و با افزایش تعداد هسته های قدرت پردازنده ها به هدر می رود . در شکل ۱-۱۰ همان اطلاعات با استفاده از نسخه جدیدی از الگریتم نشان داده شده، در این شکل ۹۰ درصد (P = 0.90) از کل کار موازی اجرا شده است. در واقع به اندازه ۹۰ درصد در موازی سازی موفقیت داشته ایم. اما نتایج SpeedUP برابر ٦.٤٠ x در ریز پردازنده ای با ١٦ هسته می باشد .

$$\left(\text{در واحد زمان}\right) \text{SpeedUP حداکثر} = \frac{1}{(1-0.90)+\left(\frac{0.90}{16}\right)} = 6.40\ \text{x}$$

در قانون Amdahl تغیراتی اساسی در تعداد هسته های فیزیکی اعمال شده اما ویژگی های بلقوه جدیدی که شما به برنامه موجودتان برای استفاده از مزیت های قدرت پردازش موازی افزوده اید در نظر نمی گیرد. برای مثال می توانید الگریتم جدید ایجاد کنید تا از مزایای هسته های اضافی سود ببرید و این کار را تا زمانی انجام می دهید که الگریتم های دیگری را موازی اجرا می کنید. آنگاه به موفقیت های بزرگی در کارایی برنامه هاتن می رسید و این زمانی رخ می دهد که سه هسته برای اجرای آن استفاده شود. ضمناً شما برای کاهش فشرده سازی قانون Amdahle می توانید سناریوهای موازی گوناگونی را ایجاد کنید. اجباراً برنامه های کاربردی را با توانایی های سخت افزار های ارائه شده به روز کرد .

نسخه (کد) ترتیبی اصلی

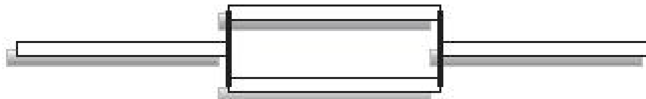
کار کل : ۱۰۰۰ واحد

کار ترتیبی : ۱۰۰۰ واحد

نسخه بهینه شده

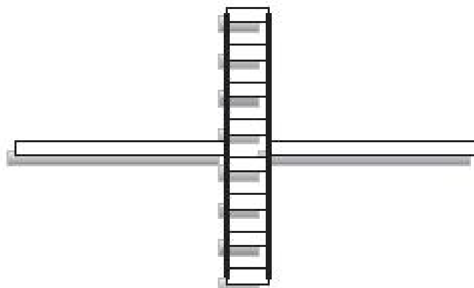
کار کل : ۱۰۰۰ واحد

کار ترتیبی ۲۵۰ واحد	کار کاملاً موازی : ۵۰۰ واحد	کار ترتیبی ۲۵۰ واحد



دو هسته فیزیکی

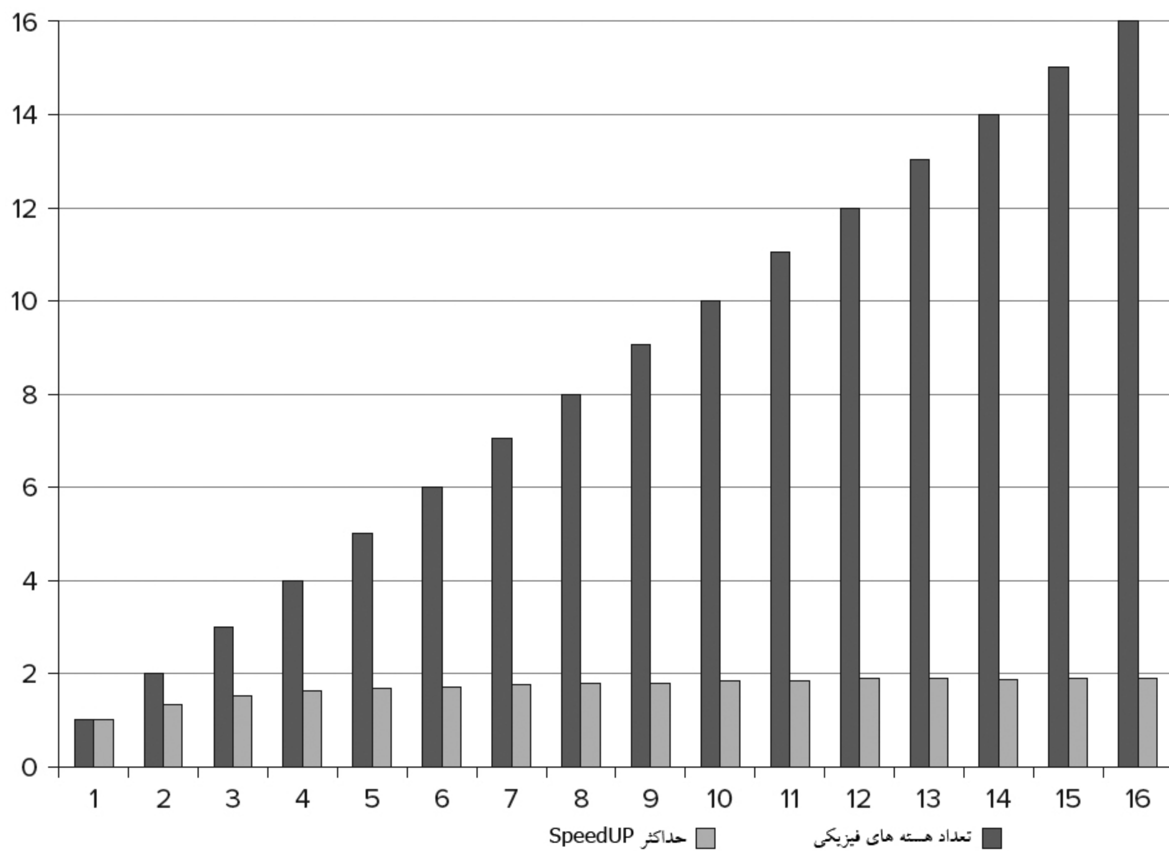
۲۵۰ واحد روی هر هسته فیزیکی



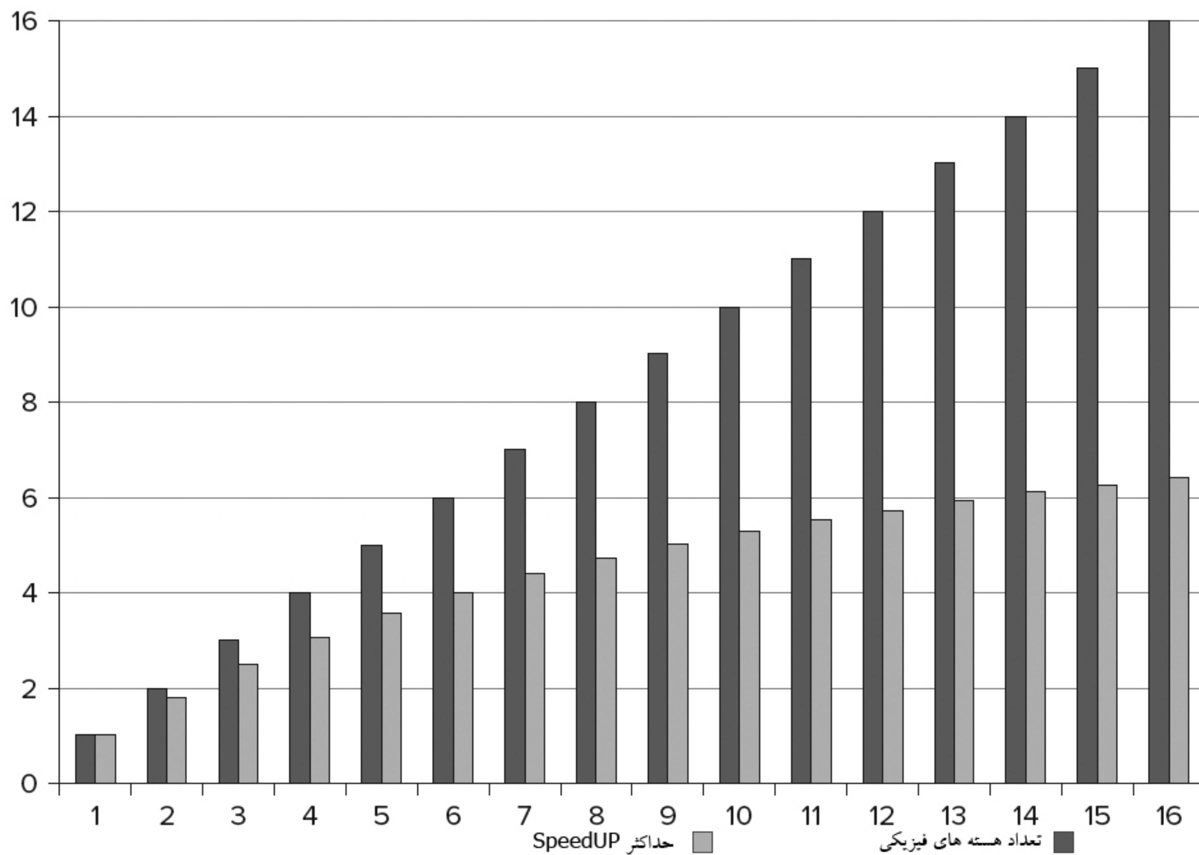
هشت هسته فیزیکی

٦٢ یا ٦٣ واحد روی هر هسته فیزیکی

شکل ۱–۸



شکل ۱-۹



شکل ۱-۱۰

۱-٤ ملاحظه قانون Gustafson

Gustafson درباره قانون Amdahl هشداری بخاطر ثابت بودن الگریتم داد و گفت که این الگریتم تغییرات سخت افزاری ای را که در آن سخت افزار اجرا می شود را مد نظر نمی گیرد. بنابر این در سال ۱۹۸۸ پیشنهاد ارزیابی مجدد الگریتم را داد. او بررسی کرد و به این نتیجه رسید که بهتر است SpeedUP توسط پیمایش مشکل در تعداد پردازنده ها و عدم ثبات در اندازه مشکل شمارش شود. زمانی که افزایش قدرت سخت افزاری امکان پردازش موازی را فراهم آورد مشکل در مقایسه بار کاری بود . قانون Gustafson روی اندازه مشکل برای اندازه گیری میزان کاری که در زمانی ثابت انجام می شود تمرکز کرده است :

$$\text{کار کل}\left(\text{در واحد زمان}\right) = S + (\,N \times P)$$

S : واحد های کاری که بصورت ترتیبی انجام می شود .

P : اندازه واحد های کاری که به طور کامل موازی است .

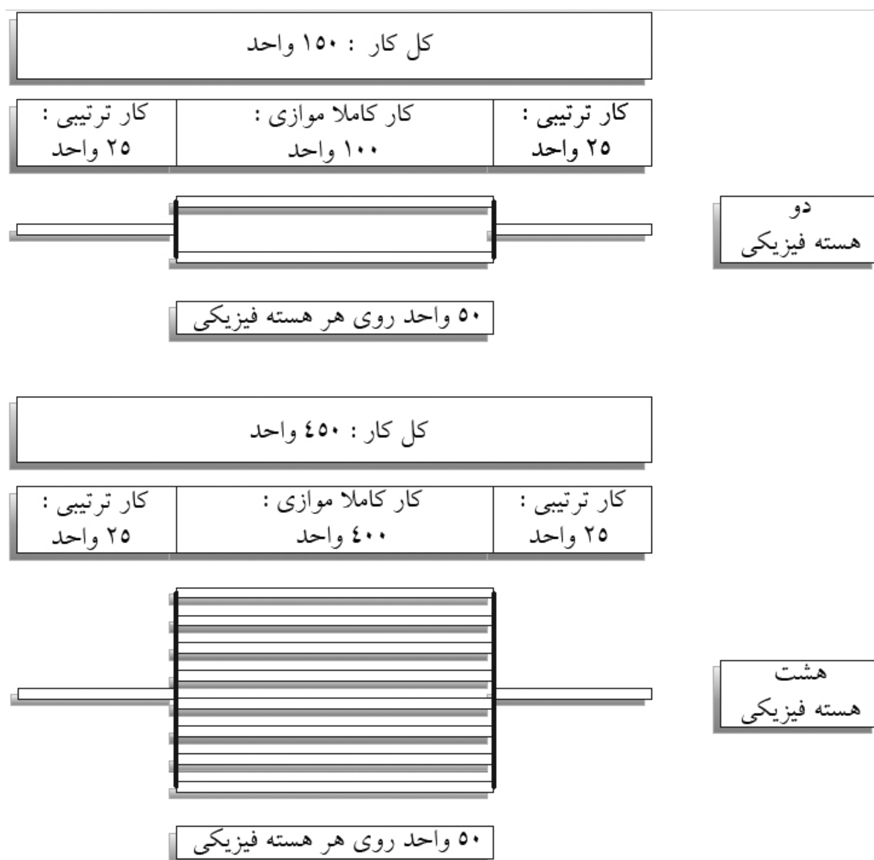
N : تعداد واحد های اجرای در دسترس (پردازنده ها یا هسته های فیزیکی).

به این موضوع فکر کنید که مشکل در ترکیب ۵۰ واحد کاری، در اجرایی موازی است. همچنین این مشکل در زمانبندی ۵۰ واحد کاری برای هر هسته که در دسترس می باشد رخ می دهد. اگر ریز پردازنده ای با دو هسته فیزیکی داشته باشید و قصد انجام حداکثر ۱۵۰ واحد کاری را دارید. در آن صورت حداکثر کار بصورت زیر محاسبه می شود :

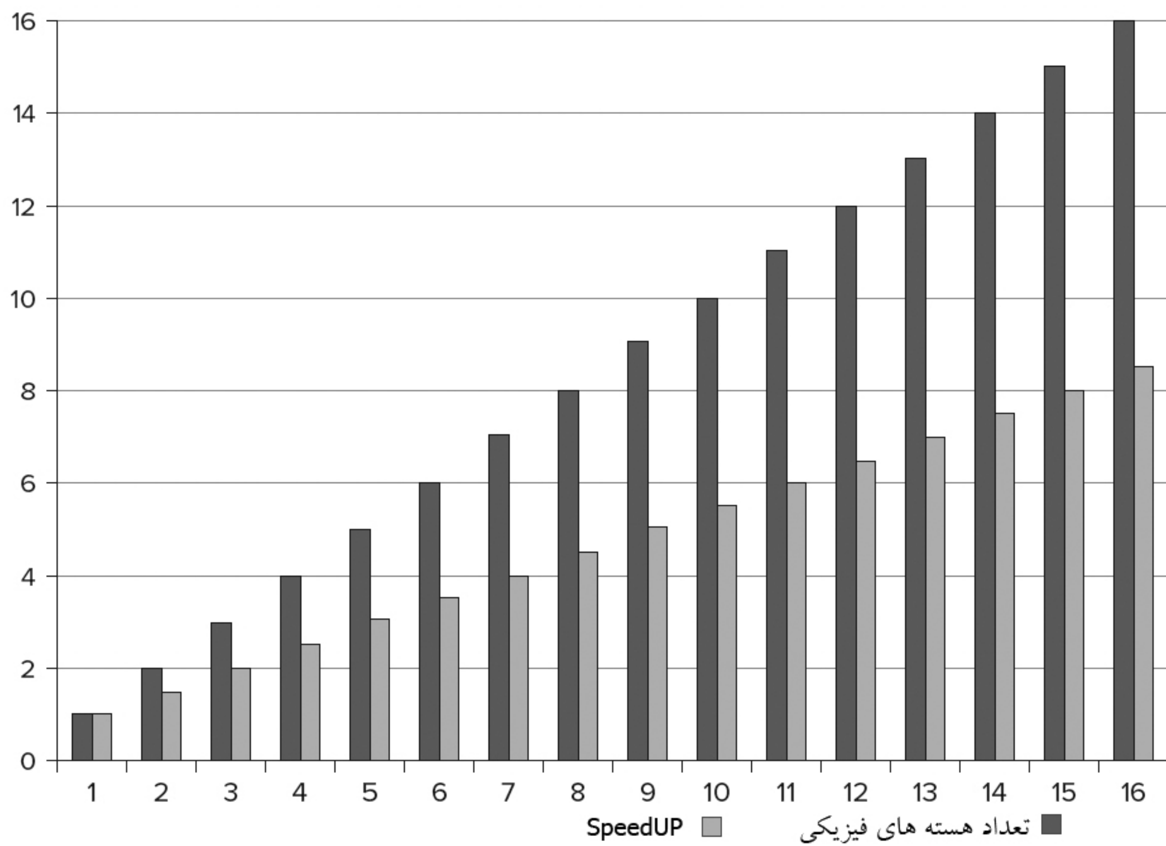
در شکل ۱-۱۱ الگریتمی با ۵۰ واحد کاری رسم شده است که دارای قسمت موازی و ترتیبی می باشد. مقایسه های قبلی بر اساس تعداد هسته های فیزیکی بوده است در این شیوه در طی روندی بخشی که موازی سازی شده است با ۵۰ واحد کاری که قابلیت موازی سازی را داراست مقایسه کرد. در زمانی که بیشتر هسته ها در دسترس اند بار کاری بخش موازی سازی شده بالا می رود. اگر واحد های اضافه شده کاری برای پیش روی بخش موازی سازی کافی باشد الگریتم، داده ها را در زمان کمتری پردازش می کند. در حالتی که الگریتم قبلی روی یک پردازنده ای با هشت هسته فیزیکی اجرا شود، در زمانی برابر با حالت قبلی ٤۵۰ واحد را پردازش می کند :

در شکل ۱–۱۲ SpeedUP الگریتم را بر اساس تعداد هسته های فیزیکی ۱ الی ۱٦ می بینید . با افزایش تعدا هسته ها، احتمالاً SpeedUP واحد های کاری لازم را برای اجرای موازی فراهم می کند. شما SpeedUP بهتری را نسبت به نتیجه قانون Amdhel می بینید. در شکل ۱–۱۳ مقدار کار کل بر اساس تعداد هسته های فیزیکی هسته(از یک الی ۱۲) رسم شده است .

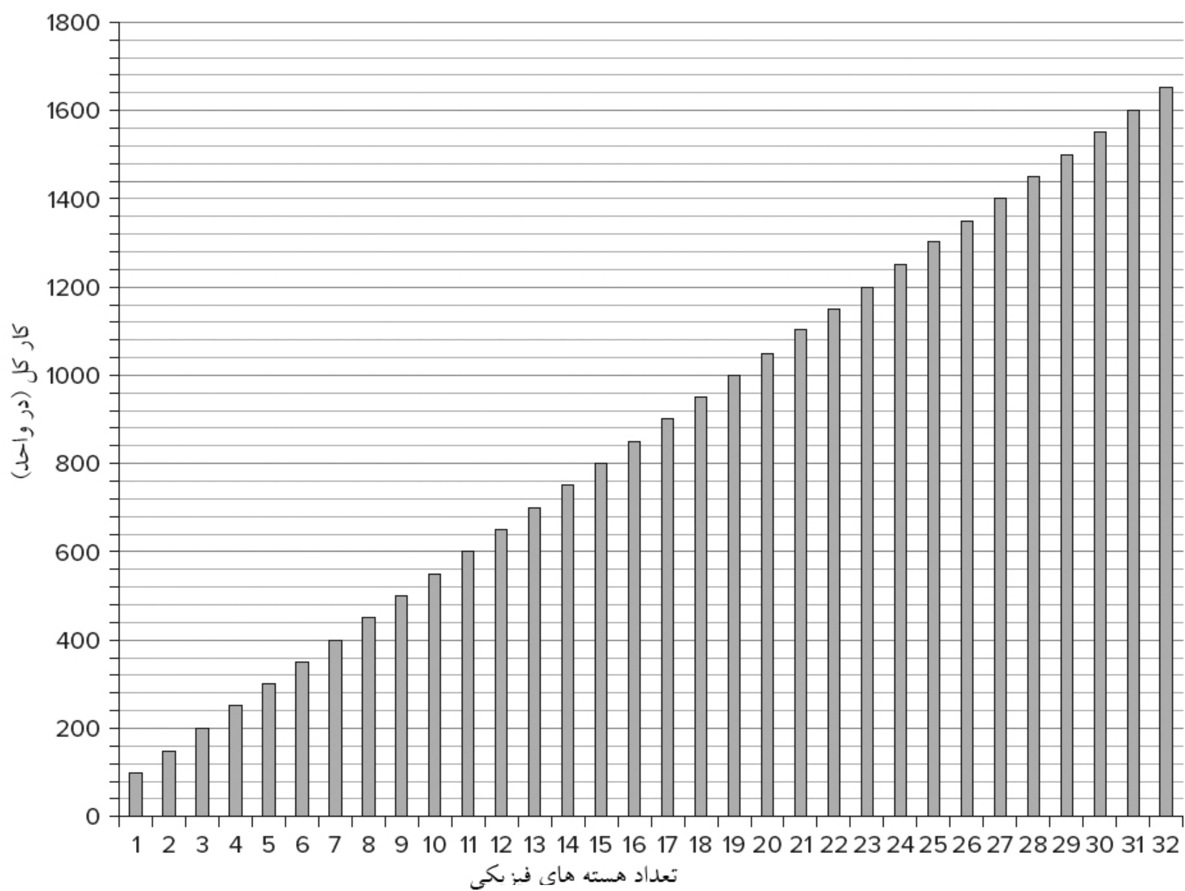
گاه گاهی ، مقدار زمانی که در بخش های ترتیبی برنامه صرف می شود به اندازه مشکل بستگی دارد. در این حالت شما اندازه مشکل را بر اساس افزایش دستیابی تصادفی به SpeedUp ای بهتر نسبت به قانون Amdahl محاسبه شده مقایسه کنید. به هر حال در برخی مشکلات در میزان داده های موازی پردازش شده محدودند و هنگامی چنین موردی رخ می دهد که شما ویژگی های جدیدی را برای کسب مزیت های قدرت پردازش موازی در سخت افزار های پیشرفته اضافه کنید یا با طرح های مختلف کار کنید . در فصل های بعدی تکنیک های زیادی در آماده سازی و بهبود الگریتم برای محاسبه کار کل توسط قانون Gustafson ارائه شده است .



شکل ۱–۱۱

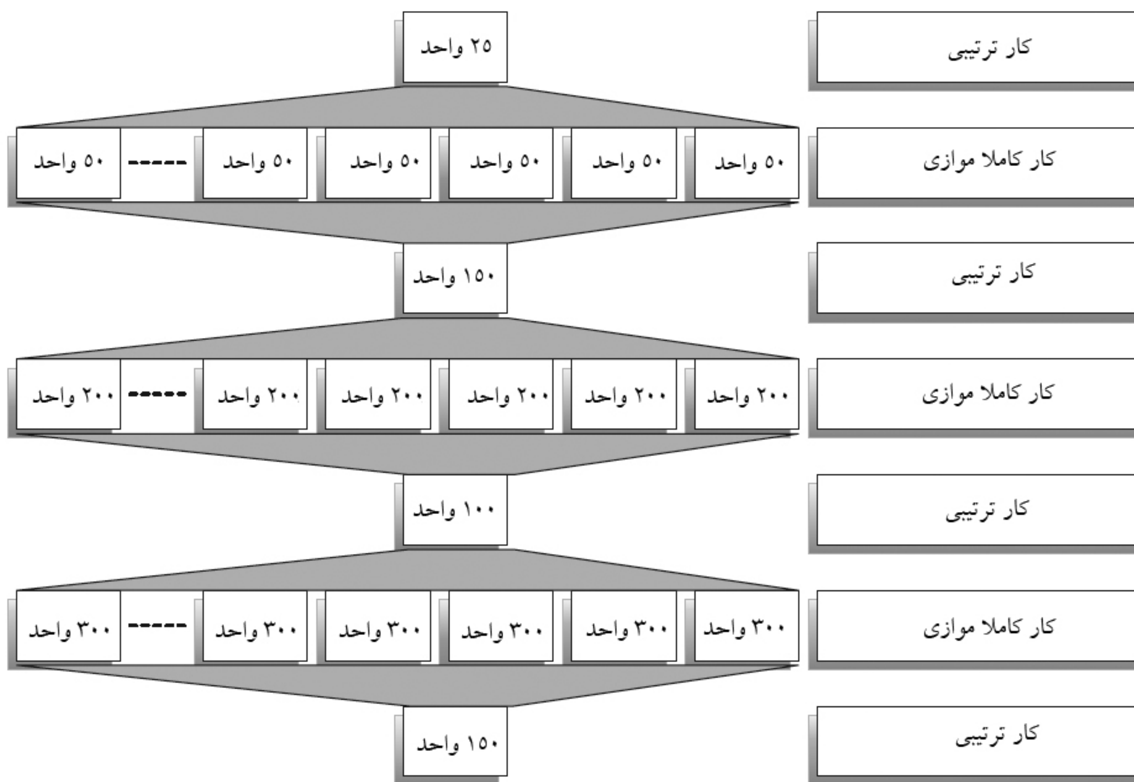


شکل ۱–۱۲



شکل ۱–۱۳

همانطور که در شکل ۱–۱٤ مشاهده می کنید تعداد زیادی الگریتم نشان داده شده است که چند واحد کاری با اجرایی ترتیبی و بخش هایی موازی ترکیب شده است. بخشهای موازی شده زمانی مقایسه می شوند که تعداد هسته های در دسترس افزایش پیدا کنند. کاهش کد های ترتیبی نسبت به کد های موازی به میزان زیادی باعث کاهش واحد های کاری می شود. در این حالت یک نیاز اساسی این است که کل واحد های کاری در کد های ترتیبی و موازی محاسبه شود و سپس با قرار دادن در فرمول کار کل با هشت هسته را می یابیم :



شکل ۱–۱٤

در شکل ۱–۱٤ اجرایی ترتیبی را نشان می دهد که فقط ۹۷۵ واحد کاری را در همان فاصله زمانی قبلی اجرا می کند :

واحد کاری 975 = 150 + 300 + 100 + 200 + 150 + 50 + 25 = کار کل با اجرایی ترتیبی (در واحد ها)

۱-۵ کار با همزمانی سبک وزن

نه قانون Gustafson و نه قانون Amdahl مقدار Overhead موازی سازی را در نظر نمی گیرند. الگو هایی اجازه انتقال بخش های ترتیبی به الگریتم های جدیدی را دارند که از مزیت موازی سازی استفاده می کنند ولی در قانون فوق آنها را در نظر نمی گیرد. برای اینکه بتوانیم از اجرای موازی واحد ها استفاده کنیم باید کد های ترتیبی ای را که اجرایشان اجباری نیست را کاهش دهیم .

در نسخه های NET Framework. برای اینکه کدی که در C# نوشته اید را بصورت موازی اجرا کنید (در یک فرآیند)، مجبور می بودید که خودتان کار ایجاد و مدیریت نخ های نرم افزاری گوناگون را انجام دهید. پس مجبور بودید کد های بنویسید که دارای نخهای متعدد و پیچیده اند . تقسیم الگریتم به چندین نخ، هماهنگی واحد های مختلف کد ، تفسیم کردن اطلاعات در میان آنها و جمع آوری نتایج واقعاً کاری پیچیده در برنامه نویسی است. هنگامی که تعداد هسته های منطقی افزایش یابد، این پیچیدگی هم بیشتر می شود، چرا که نیاز به نخ های بیشتر و دستیابی ای بهتر نسبت به قبل دارید .

مدل چند نخی به عنوان ابزاری کمکی برای توسعه دهندگان در انقلاب چند هسته ای طراحی نشده است در واقع ایجاد نخ به مقداری از دستورالعمل ها برای پردازنده نیاز دارد و نیز به میزانی کم Overhead را برای تمامی الگریتم های ایجاد می کند که به اجبار کار را در نخ هایی موازی شده تقسیم می کنند. مقدار زیادی از Struct ها و Class های مفید برای دسترسی توسط نخ های مختلف طراحی نشده اند و بنابراین این امکانات را برای مقداری از کد ها باید نوشته شود. حال این کد های اضافه شده باعث می شود فکر توسعه دهنده از هدف اصلی که دستیابی به افزایش کارایی در اجرای موازی است ، پرت شود.

زیرا این مدل چند نخی پیچیده تر از آن است که توسط انقلاب چند هسته ای که معروف به همزمانی سنگین است مدیت شود. این کار خود سربار زیادی را اضافه می کند و نیاز به اضافه کردن خطوط بسیار زیادی از کد دارد و باعث می شود کار مدیریت کد با مشکلاتی بلقوه روبرو شود چرا که دسترسی به چند نخی در سطح Framework پشتیبانی نشده و باعث دشوار شدن در فهم کد می شود .

با پیوستن مشکلات مذبور با مدل چند نخی که توسط نسخه های NET Framework. قبلی ارائه شده است و همچنین کاهش تعداد هسته های منطقی در مدل ریزپردازنده های محرک[1]، جواز ایجاد مدل جدیدی از بخش های موازی کد را می دهد. مدل جدید به مدل همزمانی سبک وزن[2] معروف است. دلیل این نام گذاری این است که سربار کل مورد نیاز را کاهش می دهد و کار ایجاد و اجرای کد را در هسته های منطقی مختلف انجام می دهد. حرف گفته شده به این معناست که NET Framework. سرباری که توسط موازی سازی (با نسخه های قبلی) ایجاد می شد را حذف می کند بنابراین این مدل برای کار با ریز پردازنده های چند هسته ای امروزی آماده شده است. تاریخ آغاز همزمانی سنگین وزن[3] برابر با آغاز چند هسته ای ها بود، زمانی که یک رایانه تعداد بسیاری ریزپردازنده فیزیکی با یک هسته فیزیکی در هر کدام داشت. مدل همزمنی سبک وزن در داخل ریز معماری های جدیدی در هسته های منطقی زیادی وجود دارد که بوسیله همان هسته های فیزیکی پشتیبانی می شود. مدل همزمانی سبک وزن فقط درباره کار همزمانی در هسته های منطقی گوناگون صحبت نمی کند. این مدل دسترسی به چند نخی را در سطح Framework پشتیبانی می کند و همچنین فهم کد ها را بسیار ساده تر می سازد. اکثر زبان های برنامه نویسی امروزی به سمت همزمانی سبک وزن حرکت می کنند. خوشبختانه NET Framework 4. جزئی از این انتقال می باشد. بنابراین همه زبان های Visual Studio که برنامه های NET. تولید می کنند از این مزیت جدید استفاده می کنند .

۱-٦ ایجاد موفق طرح های Task-Based

در برخی موارد در موازی سازی مجبور به بهینه کردن راه حل موجود هستید و در برخی زمان ها مجبور به درک طرح متوالی موجود هستید یا الگریتم موازی که مقیاسپذیری را کاهش می دهد و سپس شما باید برای دستیابی به بهبود کارایی بدون معرفی کردن مشکلات یا تولید نتایج متفاوت به نتیجه برسید. شما برای

---

[1] Motirated
[2] Lightweight Concurrency
[3] Heavytweight Concurrency

جزءی کوچک یا مشکل کاملی می توانید طرح Task-Based را ایجاد کنید و سپس موازی سازی را معرفی کنید . هنگامی که شما مجبور به طراحی راه حلی جدید اید از تکنیک های ثابتی استفاده کنید .

برای ایجاد طرح های Task-Based موفق گامهای زیر را دنبال کنید :

- ✓ هر مشکل را به مشکل ریز تری تقسیم کنید و اجرای ترتیبی را فراموش کنید .
- ✓ درباره مشکلاتی که کوچکشان کرده اید تفکر کنید و همه این مراحل را دنبال کنید: داده های که بطور موازی می توانند پردازش شوند – تجزیه داده ها برای به نتیجه رسیدن موازی سازی داده هایی که نیاز به Task های زیادی دارند و قادرند با انواع یکسانی از موازی سازی پیچیده پردازش شوند را دنبال کنید – تجزیه داده ها و وظایف برای به نتیجه رساندن موازی سازی وظایفی که موازی اجرا می شوند. تجزیه وظایف برای رسیدن به موازی سازی .
- ✓ طراحی ای را برای سازمان دهی کنید که موازی سازی را ارائه می دهد .
- ✓ تعیین نیاز برای وظایف متصل برای مشکلات ریزِ گوناگون . سعی کنید از وابستگی های زیادی که ممکن است رخ دهد اجتناب کنید .
- ✓ در ذهن طرحی را با همزمانی و موازی سازی ای بالقوه انجام دهید .
- ✓ طرح اجرای برنامه تان را برای مشکلات رایجی که در موازی سازی برای پردازنده های چند هسته ای و مشخصه معماری ها پیش می آید را تجزیه و تحلیل کنید . برای مقیاس های بالاتر طرح تان را آماده کنید .
- ✓ نواحی بحرانی[1] ممکن را به حداقل برسانید .
- ✓ در هر زمان ممکن برنامه نویسی Task-Based را برای اجرای موازی سازی بکار بگیرید .
- ✓ برگردید و از ابتدا شروع کنید .

مراحل مزبور به این معنی نیست که همه مشکلات خرد شده به Task های موازی در قالب نخهای مختلف اجرا می شوند. در طرح باید مواردی ممکنی که برای موازی سازی رخ می دهد را مد نظر قرار داد. سپس کد را نوشت، شما بهترین تصمیم را با مقایسه اهداف و کارایی می توانید بگیرید. البته این مورد بسیار با

---

[1] Critical Section

اهمیت است که به موازی سازی و تکه تکه کردن کار با Task ها فکر کنید. در این شیوه اگر طرحی را برای کد های ترتیبی معمولی آماده کنید، شما قادر به موازی سازی برای کد مد نظرتان خواهید بود .اگر سعی بسیاری در موازی سازی دارید و می خواهید به نتیجه برسید، از تکنیک های Task-Based استفاده کنید .

می توانید طرح های Task-Based را با Object-Oriented ترکیب کنید در واقع شما با استفاده از قابلیت Object-Oriented در کد می توانید از موازی سازی کپسوله (Encapsulate) استفاده کنید و شیء ها و اجزاء موازی ای را ایجاد کنید .

۱-۶-۱ طراحی با همزمانی در ذهن

زمانی که کدی را برای استفاده از قابلیت های چند هسته ای طراحی می کنید، این مسئله بسیار حائض اهمیت است که به اجرا شدن کد درون تنها یک برنامه #C فکر نکنید. در #C از قابلیت همزمانی کد نیز می توان استفاده کرد، به این معنی که بطور پیوسته تکه های زیادی از کد داخل یک فرآیند (Process) اجرا می شوند یا با یک اجرای تکه تکه[1] این کار انجام می گیرد. تابع در یک کلاس کد های خود را به صورت همزمان اجرا می کنند. اگر این تابع مقدار را در متغیری Static ذخیره کند، سپس این مقدار را استفاده کند، تعداد بسیار اجرا های همزمان ماحصل در نظر گرفتن استثناء های رخ داده و نتایج قابل پیشبینی است .

همانطور که در طرح پیشین شرح داده شد برنامه نویسی موازی برای ریزپردازنده های چند هسته ای با مدل حافظه–مشترک کار می کند. اگر در طراحی، همزمانی را مد نظر نداشته باشیم، آنگاه مقیم سازی داده در حافظه مشترک یکسان باعث وقوع نتایج پیشبینی نشده می شود .

یک تمرین خوب این است که هر کلاس و متد را برای اجرای موازی سازی بدون اثرات متقابل آماده کنیم. اگر کلاس ها، کامپوننت ها یا متد های دارید که برای همزمانی طراحی نشده اند شما باید طراحی شان را خطایابی (تست) کنید.

تا زمانی که مشکلات کوچک بطور همزمان در حالت اجرا باشند. هر مشکل کوچکی که در فرآیند طراحی تشخیص داده شده باید توانایی اجرا را داشته باشد. اگر به این نکته می اندیشید که نیاز به محدود سازی کد

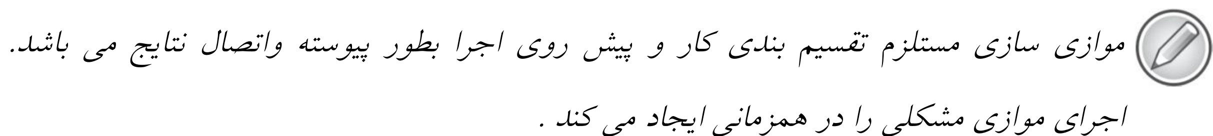
[1] InterLeaved Execusion

همزمان در زمانی هست که مشکل کوچک مطلقی به این دلیل اجرا می شوند ، ارث بری را در کلاس ها، متد ها و کامپوننت ها استفاده کنید بهتر این است که چنین مواردی را در طراحی خود روشن کنید .

یکبار دیگر شروع بکار موازی سازی کد کنید ، در این هنگام ساده است که کلاس ها، متد ها و کامپوننت های موجود دیگر را یکی[منسجم] کنید زیرا برای اجرای همزمان طراحی شده اند .

۱-۶-۲ درک اختلاف بین همزمانی تکه تکه ای ، همزمانی و موازی سازی

در شکل ۱-۱۵ تفاوت های بین همزمانی برگ برگ شده (تکه تکه ای) و همزمانی را روشن می کند که در آن دو نخ سخت افزاری است که هر یک چهار دستور العمل را انجام می دهند . سناریوی همزمانی برگ برگ شده ، اجرا یک دستور العمل برای هر نحو جاگذاری آنها می باشد. اما سناریوی این است که (در زمان یکسان با همزمانی برگ برگ شده) دو دستور العمل را موازی اجرا می کند. طراح را برای هر دو سناریو آماده کنید هر همزمانی نیازمند پیشروی همزمانی بطور فیزیکی برای رخ دادن دارد .



موازی سازی مستلزم تقسیم بندی کار و پیش روی اجرا بطور پیوسته واتصال نتایج می باشد. اجرای موازی مشکلی را در همزمانی ایجاد می کند .

کدی را که موازی سازی کرده اید را در سناریو های همزمانی و همزمانی برگ برگ شده بسیار زیاد و گوناگونی می توانید اجرا کنید. حتی هنگامی که در پیکره بندی سخت افزاری یکسانی اجرا شده باشند بدین گونه بزرگترین مشکل طراحی مطمئن شدن اجراهای معتبر و ممکن است که مرتب شده اند و جاگذاری نتایج صحیح می باشد. در غیر اینصورت معروف به CorrectLess یا صحت است. اگر شما به ترتیبی خاص یا بخش هایی مطمئن در اجرا با یکدیگر [همزمان] نیاز ندارید . لازم است از قطعاتی که به شکل همزمان اجرا نمی شوند مطمئن باشید نمی توانید فرض کنید که به طور همزمان اجرا نمی شوند زیرا کد را چند بار اجرا می کنید و نتایج پیش بینی شده را تولید می کنید. زمانیکه شما برای همزانی و موازی سازی طرحی را طراحی می کنید باید از درستی شکل خود (Correctless) اطمینان حاصل کنید .

در فصل بعدی موارد بسیاری درباره اختلاف بین موازی سازی و همزمانی توسط مشاهده کد مثال های گوناگون فرا می گیرید .

۱-٦-۳ موازی سازی Task ها

Visual C# 2010 و NET Framework 4. انتقال طرح های Task-Based به کد های موازی را ساده کرده اند. به هر حال چیزی که مهم است این است که موازی سازی کد نیاز به تست های خاص و روش هایی منظم دارد تا به اهداف پیش بینی شده برسیم. شما درباره این مطلب در ادامه کتاب چیز های بیشتری را فرا می گیرید. زمانی که شما Task ها را موازی سازی می کنید سرباری که توسط موازی سازی شرح داده شده اثر مهمی می تواند داشته باشد و احتمالا نیاز به تست های جایگزین متفاوتی دارد بطوری که قبلا توضیح داده شد ریز پردازنده های چند هسته ای پیشرفته فوق العاده پیچیده اند و لازم است تا نتایجی که توسط تکنیک های موازی سازی مختلف ارائه شده است را امتهان و یکی را انتخاب کنید. در واقع اتفاقی یکسانی با کد های ترتیب رخ می دهد اما با شناخت قبلیتان فرق دارد . که یک حلقه Foreach کند تر از حلقه For می باشد. مادامی که در Task های موازی، نسخه های موازی سازی شده ای در حلقه For ارائه شده است که نتایج عملکردی متفاوتی دارد بر این اساس که پارامتر های شیوه موازی سازی Task ها برای اجرا را تعیین می کنند. شما با تجربه این سناریو ها می توانید تصورشان کنید هنگامی که به نوشتن کد برای مسائلی مشابه و طرح های Task-Based قابل قیاس مجبورید .

معمولا لازم است در کار های چندگانه زنجیره ای، در جایی که می توانیم کد را تکه تکه کنیم (یعنی مشکلات را تقسیم کنیم .) مسئله را به تکه های کوچکی تقسیم کنیم و برای انجام این Task ها ، Task ها را بصورت موازی اجرا ، نتایج را جمع آوری و سپس چرخه های حیاط بسیاری را تکرار کنیم. بعد از تصمیم موازی سازی ، جایگزینی مشکلی خاص اصلاً صلاح نیست. سعی نمایید نموداری با اجرای موازی بالقوه ایجاد کنید چنانچه ممکن است با ۱۰۰۰ هسته اجرا کنید. اگر برخی از اجزاء یک الگریتم موازی سازی ، مقایسه پذیری مورد نظر را ارائه ندهند شما شانس اجرای موای با دیگر Task ها را دارید .

۱-٦-٤ حداقل سازی نواحی بحرانی

هر دو قانون Amdhel و Gustafson کار پشت سر هم و پی در پی را برای یک مشکل خاص و افزایش کارایی کل در الگریتم موازی تشخیص می دهند. زمان سری ما بین دو ناحیه موازی ، که نیاز به یک اجرای

ترتیبی دارد ناحیه بحرانی[1] می نامند .در شکل ۱-۱۶ با استفاده از قانون Guastafson چهار ناحیه بحرانی در یکی از نمودار ها شناسایی شده است .

زمانیکه Task ها را موازی می کنید ، یکی از پر اهمیت ترین اهداف دستیابی به بهترین عملکرد در حداقل کردن نواحی بحرانی است در بیشتر زمان ها ، غیرممکن است که کدی اجباراً دارای اجرای ترتیبی بین دو ناحیه موازی سازی شده است اجتناب کرد . بنابراین برای اینکه راه اندازی کار های موازی و جمع آوری نتایج لازم است .به هر حال مطلوب کردن کد ناحیه بحرانی و حتی حذف یک مورد غیر ضروری مهمتر از تنظیم کد های موازی است . زمانی که با نواحی بحرانی زیادی در طرح اجرا روبرو می شوید ، قانون Amdhel را بخاطر آورید . اگر نواحی بحرانی را نمی توانید کاهش دهید سعی در یافتن Task های کنید که موازی با نواحی بحرانی اجرا شوند . به عنوان مثال با Pre-Fetch کردن داده های که قصد استفاده توسط الگریتم موازی الگریتم موازی بعدی برای موازی سازی با ناحیه بحرانی را دارد ، عملکرد کلی ارائه شده بوسیله راه حل را بهبود می بخشید . این مسئله بسیار مهم که توانایی سخت افزارهای چند هسته ای پیشرفته را در نظر داشته باشید.از تفکر درباره یک واحد اجرایی دوری کنید

۱-۶-۵ درک قوانین برنامه نویسی موازی برای چند پردازنده های هسته ای

James Reinders مقاله ای را در نشریه Dr.Dob با موضوع "برنامه نویسی موازی برای پردازنده های چند هسته ای " (www.drdobbs.com/hpc-high-performance-computing/201804248) منتشر کرد. او هشت قانون رابرای توسعه دهندگان برنامه های موازی برشمرد. همانگونه که توصیف شده ، قوانین او برای تولید برنامه های موازی با C# و NET Framework. ، مفید می باشد :

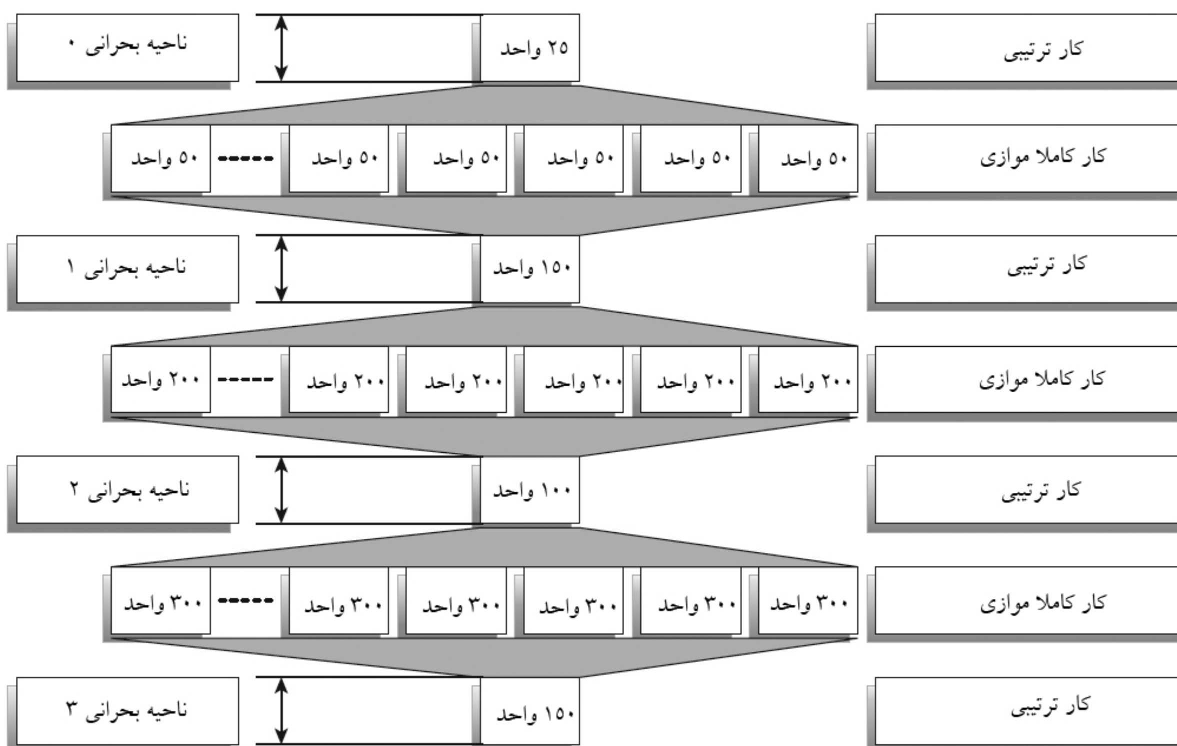
[1] Critical Section

- ✓ به موازی سازی فکر کنید – این قانون درباره طراحی موازی سازی در ذهن می باشد بطوریکه قبلا در مورد آن بحث شد .

همزمانی تکه تکه شده		
دستورالعمل	نخ	زمان
0	0	t0
0	1	t1
1	0	t2
1	1	t3
2	0	t4
2	1	t5
3	0	t6
3	1	t7

همزمانی با پردازش فیزیکی همزمان				
دستورالعمل	نخ	دستورالعمل	نخ	زمان
0	1	0	0	t0
1	1	1	0	t1
2	1	2	0	t2
3	1	3	0	t3

شکل ۱–۱۵



شکل ۱–۱٦

- ✓ برنامه ها در Task ها (خرده کار ها) ، نه نخ ها (هسته ها) – TPL به شما اجازه می دهد : کد بنویسید، طرح های Task-Based را پیاده کنید ، بدون اینکه درباره نخ های کم اهمیت نگران باشید .
- ✓ طرحی با امکان لغو همزمانی – زمانی که کد را با استفاده از TPL می نویسید ، کد تنها در رایانه های با یک هسته در ریزپردازنده اجرا می شود (تنها یک هسته فیزیکی وجود دارد) .
- ✓ اجتناب در استفاده از قفل[1] ها – استفاده از Class ها ، Struct ها و Function های جدید دارای اهمیت بسیاری است چرا که نیاز به راه حل های جدید همزمانی پیچیده را از بین می برد. TPL با ساده تر سازی اجتناب نمودن از استفاده Lockهای سنگین در سناریو های پیچیده بسیاری، مکانیزم های همزمانی سبک جدید را ارائه می دهد.

[1] Lock

- ✓ از ابزار ها و کتابخانه هایی که برای کمک به همزانی استفاده شده اند استفاده کنید .Visual studio 2010 ابزار جدیدی را برای خطایابی ، تست و مقایسه کد های موازی ارائه می دهد . شما در این کتاب در مورد ابزار ها و کتابخانه های موجود مواردی را می توانید فرا بگیرید .
- ✓ از تخصیص دهنده حافظه استفاده کنید – تخصیص دهنده های حافظه مقیاس پذیر در CLR[1] و آن بطور خودکار در زمانی با نخ ها و Task ها کار می کند ، استفاده می کند .به هر حال برای حداکثر استفاده از حافظه کش باید امکانات قسمت بندی بخش های مختلف را تحلیل کنید و سعی کنید بیش از اندازه از حافظه هر Task استفاده نکنید .
- ✓ در طراحی افزایش بارکاری را حذف کنید - یکبار در زمانی که شما مهارت اجرای موازی را یافتید ، قانون Guastafson را با کلاس های ارائه شده در TPL را در نظر بگیرید. اگر شما طرح هایتان را برای استفاده در آینده آماده کنید می توانید کدی را بنویسید که در مقایسه با افزایش تعداد هسته ها کارایی آن نیز افزایش یابد . Widows 7 و Windows Servaer 2008 R2 از ٢٥٦ نخ سخت افزاری یا پردازشگر منطقی پشتیبانی می کنند . بنابراین آنجا اتاقی برای اندازه پذیری می باشد .

۱-۷ آماده کردن NUMA و مقایسه پذیری بالاتر

در سال های اخیر اکثر مدل های منتشر شده برای پشتیبانی از پردازنده های چند هسته ای و چند پردازنده ای متقارن[2] برای معماری non-uniform memory access[3] می باشد . یکی از مشکلات بزرگ SMP ها (چند پردازش متقارن) ، محدودیت گذرگاه[4] پردازنده در مشخصات اندازه آن است زیرا هر پردازنده دسترسی برابری را برای حافظه و ورودی خروجی داراست . با NUMA دستیابی به حافظه نزدیکتر و سریعتر از حافظه های خارجی و دورتر است (حافظه های خارج از سیستم). NUMA را هنگامی بهتر می توان مقایسه نمود که تعداد پردازنده ها بیشتر از چهار عدد باشد. در ویندوز اصطلاحاً Scale-Up-

---

[1] Common Languge Runtime
[2] Symmetric MultiProcessor (SMP)
[3] NUMA
[4] Bus

Technology و ضوابط NUMA در زیر شرح داده شده است (در شکل ۱–۱۷ می توانید نمودار این سازماندهی را ببینید) :

- ✓ یک کامپیوتر یا ماشین ، یک گروه[1] یا بیشتر دارد .
- ✓ هر گروه یک گره[2] یا بیشتر دارد .
- ✓ هر گره NUMA یک (یا بیشتر) پردازشگر فیزیکی یا سوکت دارد (یک ریز پردازنده واقعی) .
- ✓ ریز پردازنده های گوناگون کار ترکیب گره های NUMA ها را به منظوردستیابی به حافظه محلی یا ورودی/ خروجی[3] انجام می دهند .
- ✓ هر پردازنده یا سوکت یک هسته فیزیکی یا بیشتر را دارد ، زیرا بطور معمول یک ریز پردازنده چند هسته ای می باشد. هر هسته فیزیکی حداقل یک پردازشگر منطقی یا نخ سخت افزاری و یا بیشتر را می تواند دارا باشد .

شکل ۱–۱۸ یک کامپیوتر را با یک گره نشان می دهد که شامل دو گره (Node) NUMA هست. هر NUMA دارای دو ریزپردازنده است که به حافظه محلی و ورودی خروجی خود دسترسی دارند .اگر نخی بر روی هسته فیزیکی شماره صفر از پردازنده شماره صفر نیاز به دستیابی به داده های اختصاص داده شده در حافظه محلی برای گره NUMA داشته باشد، نخ مجبور است از گذرگاه مشترک بین گره های NUMA استفاده کند که نسبت به زمان دسترسی به حافظه محلی کندتر است . با NUMA ها کامپیوتر ها بیش از یک گذرگاه سیستم[4] دارند . یک مجموعه مطمئن از پردازنده ها هر گذرگاه سیستم در دسترس را مورد استفاده قرار می دهد .به این دلیل هر مجموعه ای از پردازنده ها می تواند به کانال های حافظه و ورودی خروجی خودشان دسترسی داشته باشند، بطوریکه قبلا شرح داده شد، آنها توانایی دسترسی به حافظه شان را به وسیله مجموعه ای از پردازنده ها و با طرح هایی هماهنگ و مناسب دارا هستند. به هر حال آشکارا دستیابی

---

[1] Group
[2] Node
[3] I/O
[4] System Bus

به حافظه شان بوسیله گره های NUMA خارجی نسبت به کار با حافظه ای که بوسیله گذرگاه سیستم محلی به آن دسترسی دارد پر هزینه تر است (حافظه گره های NUMA) .

سخت افزار های NUMA بهینه سازی خاصی را نیاز دارند. برنامه های کاربردی باید از سخت افزاری های NUMA و پیکره بندی آن آگاه باشند .به همین دلیل می توانند Task ها و نخ های را همزمان اجرا کنند که دستیابی ای یکسان به مکان های حافظه در گروه NUMA یکسان را مجبورند .برنامه های اجرایی از دسترسی به حافظه پرهزینه دوری می کنند و مجبور به توجه به همزمانی ای هستند که در حساب حافظه نیاز دارند .

Windows 7 و Wimdows Server 2008 R2 مفهوم گفته شده در بالا را در یک گروه پردازشی معرفی کرده اند .یک نخ[1] ، فرآیند[2] یا وقفه[3] می توانند یک الویت برای یک عمل در هسته ، فرآیند، گره یا گروه خاصی باشند . به هر حال برای این نوع سطح پایین تعریف شده در TPL یا C# پشتیبانی نمی شود .TPL برا کار با NUMA بهینه سازی شده و سعی در پشتیبانی نخ های موازی شده دارد. تا در مناسب ترین هسته ای اجرا کند که از حافظه محلی به اندازه ممکن استفاده کند. بنابراین کد های موازی خود را، برای NUMA آماده کنید چرا که با برخی از مشکلات عملکردی پیشبینی نشده در هنگامی روبرو می شوید که نخ هایی که اجرای موازی کد را پشتیبانی می کنند مجبور باشند به حافظه ای در گره NUMA خارجی دسترسی داشته باشند .

API ویندوز توابع بسیاری را برای کار با معماری NUMA ارائه می کند اما با نخ های مدیریت نشده سازگاری ای ندارد .بنابراین در هنگام کار با C# و TPL لازم نیست از آن استفاده کنید.

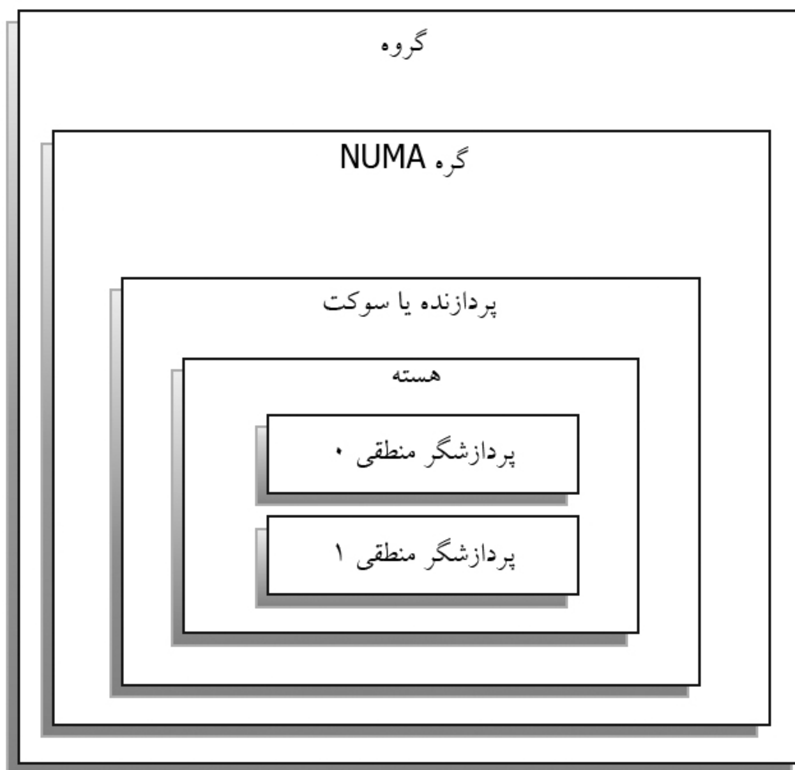
CoreInfo دستور خط فرمان قدرتمندی است که که در عین سادگی دارایی کارایی بالایت و اطلاعات بسیار مفیدی را درباره پردازشگر ها، سازماندهی ، توپولوژی ها و حافظه نهان نشان می دهد و اطلاعات را درباره نگاشت بین پردازنده های منطقی یا نخ های سخت افزاری و هسته های سخت افزاری و هسته های فیزیکی را نشان می دهد. بعلاوه ، اطلاعاتی درباره گره های NUMA، گره های و Socet ها و تمامی سطوح حافظه

---

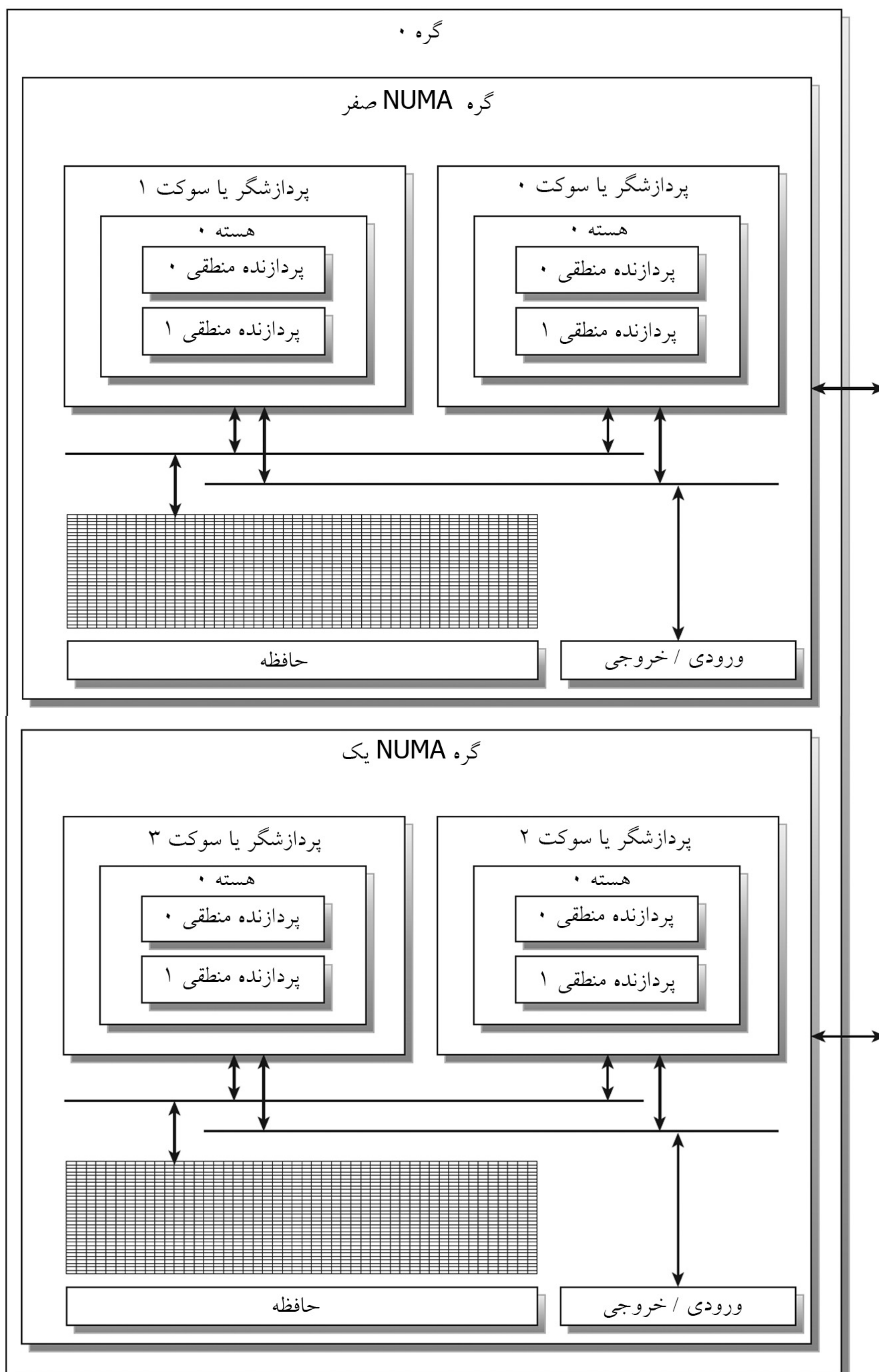
[1] Thread
[2] Process
[3] Interrupt

پنهان را نشان می دهد .شما به سادگی اطلاعاتی درباره سخت افزار های اساسی را قبل از اجرای تست کردن عملکرد برنامه هتان ذخیره کنید و نیز اگر مشکلاتی در عملکرد برنامه باعث شود ، یک معماری NUMA را مشخص کنید. CoreInfo را از مسیر زیر می توانید دانلود کنید(http://technet.microsoft.com/en-us/sysinternals/cc835722.aspx) . سپس فایل را از حالت فشرده خارج کنید و آنر ا در خط فرمان اجرا کنید (Start → All → Programs → Accessories → Command Prompt) .

کاربرد استفاده از تابع API ویندوز GetLogicalProcessorInformation () مشاهده تمامی اطلاعات بر روی صفحه نمایش است. لیست ۱-۱ نتایج اجرای Coreinfo نسخه ۲.۰ را روی کامپیوتری با فقط یک هسته ریزپردازنده i7 را نشان می دهد. تنها یک سوکت با چهار هسته فیزیکی در مشخصات مشاهده می شود که جزئیات منطقی آن در پردازنده های فیزیکی نگاشته شده است . به هر حال به این دلیل که در این CPU از تکنولوژیHyper-Threading ارائه شده ، Coreinfo خروجی .Hyperthreaded را نیز می دهد. Coreinfo از یک عدد نشان ستاره (*) برای ارائه یک نگاشت استفاده می کند .در این صورت چهار هسته فیزیکی با دو نخ سخت افزاری برای هر کدام وجود دارد.



شکل ۱-۱۷



شکل ۱–۱۸

بنابراین ، آنها با دو ستاره (**) نشان داده شده است . بعلاوه اینکه 8 MB حافظه نهان یک تکه از نوع Level 3 هست و نیز هشت نخ سخت افزاری در استفاده از این حافظه نهان سهیم اند. بنابراین Coreinfo با هشت ستاره (********) به سمت چپ آخرین خط زری حافظه منطقی تا نگاشت حافظه نهان نشان داده شده است. و این بدین معناست که حافظه نهان برای تمام نخ های سخت افزاری و هسته های فیزیکی نگاشت شده است .

لیست ۱-۱ : اطلاعات نمایش داده شده بوسیله Coreinfo نسخه ۲.۰ برای یک پردازنده Intel مدل i7

Logical to Physical Processor Map:
****------ Physical Processor 0 (Hyperthreaded)**
------ Physical Processor 1 (Hyperthreaded)**
------ Physical Processor 2 (Hyperthreaded)**
------ Physical Processor 3 (Hyperthreaded)**

Logical Processor to Socket Map:
******** Socket 0

Logical Processor to NUMA Node Map:
******** NUMA Node 0

Logical Processor to Cache Map:

```
**------ Data Cache        0, Level 1, 32 KB,  Assoc 8,  LineSize 64
**------ Instruction Cache 0, Level 1, 32 KB,  Assoc 4,  LineSize 64
**------ Unified Cache     0, Level 2, 256 KB, Assoc 8,  LineSize 64
--**---- Data Cache        1, Level 1, 32 KB,  Assoc 8,  LineSize 64
--**---- Instruction Cache 1, Level 1, 32 KB,  Assoc 4,  LineSize 64
--**---- Unified Cache     1, Level 2, 256 KB, Assoc 8,  LineSize 64
----**-- Data Cache        2, Level 1, 32 KB,  Assoc 8,  LineSize 64
----**-- Instruction Cache2, Level 1, 32 KB,  Assoc 4,  LineSize 64
----**-- Unified Cache     2, Level 2, 256 KB, Assoc 8,  LineSize 64
------** Data Cache        3, Level 1, 32 KB,  Assoc 8,  LineSize 64
------** Instruction Cache 3, Level 1, 32 KB,  Assoc 4,  LineSize 64
------** Unified Cache 3, Level 2, 256 KB, Assoc 8,  LineSize 64
******** Unified Cache 4, Level 3, 8 MB,   Assoc 16, LineSize 64
```

Logical Processor to Group Map:
******** Group 0

لیست ۱-۲ نتایج اجرای Coreinfo نسخه ۲.۰ بر روی کامپیوتری با ۲ ریز پردازنده intel Core i7 در دو گره NUMA نشان می دهد . هر گره NUMA یک سوکت با هشت نخ سخت افزاری دارد .

لیست ۱-۲ : اطلاعاتی بوسیله وسیله Coreinfo نسخه ۲.۰ در دو گره NUMa که هر کدامشان یک هسته Intel Core i7 دارند، نمایش داده شده .

Logical to Physical Processor Map:
**-------------- Physical Processor 0 (Hyperthreaded)
--**------------ Physical Processor 1 (Hyperthreaded)
----**---------- Physical Processor 2 (Hyperthreaded)
------**-------- Physical Processor 3 (Hyperthreaded)
-------------- Physical Processor 4 (Hyperthreaded)**
-------------- Physical Processor 5 (Hyperthreaded)**
-------------- Physical Processor 6 (Hyperthreaded)**
-------------- Physical Processor 7 (Hyperthreaded)**

Logical Processor to Socket Map:
********-------- Socket 0
--------****** Socket 1**

Logical Processor to NUMA Node Map:
********-------- NUMA Node 0
--------****** NUMA Node 1**

Logical Processor to Cache Map:
**-------------- Data Cache 0, Level 1, 32 KB, Assoc 8, LineSize 64
**-------------- Instruction Cache 0, Level 1, 32 KB, Assoc 4, LineSize 64
**-------------- Unified Cache 0, Level 2, 256 KB, Assoc8, LineSize 64
--**------------ Data Cache 1, Level 1, 32 KB, Assoc 8, LineSize 64
--**------------ Instruction Cache 1, Level 1, 32 KB, Assoc 4, LineSize 64
--**------------ Unified Cache 1, Level 2, 256 KB, Assoc 8, LineSize 64
----**---------- Data Cache 2, Level 1, 32 KB, Assoc 8, LineSize 64
----**---------- Instruction Cache 2, Level 1, 32 KB, Assoc 4, LineSize 64
----**---------- Unified Cache 2, Level 2, 256 KB, Assoc 8, LineSize 64
------**-------- Data Cache 3, Level 1, 32 KB, Assoc 8, LineSize 64
------**-------- Instruction Cache 3, Level 1, 32 KB, Assoc 4, LineSize 64
------**-------- Unified Cache 3, Level 2, 256 KB, Assoc 8, LineSize 64
********-------- Unified Cache 4, Level 3, 8 MB, Assoc 16, LineSize 64
-------------- Data Cache 5, Level 1, 32 KB, Assoc 8, LineSize 64**
-------------- Instruction Cache 5, Level 1, 32 KB, Assoc 4, LineSize 64**
-------------- Unified Cache 5, Level 2, 256 KB, Assoc 8, LineSize 64**

```
----------**----  Data Cache           6, Level 1,    32 KB, Assoc   8, LineSize    64
----------**----  Instruction Cache    6, Level 1,    32 KB, Assoc   4, LineSize    64
----------**----  Unified Cache        6, Level 2,    256 KB, Assoc  8, LineSize    64
------------**--  Data Cache           7, Level 1,    32 KB, Assoc   8, LineSize    64
------------**--  Instruction Cache    7, Level 1,    32 KB, Assoc   4, LineSize    64
------------**--  Unified Cache        7, Level 2,    256 KB, Assoc  8, LineSize    64
--------------**  Data Cache           8, Level 1,    32 KB, Assoc   8, LineSize    64
--------------**  Instruction Cache    8, Level 1,    32 KB, Assoc   4, LineSize    64
--------------**  Unified Cache        8, Level 2,    256 KB, Assoc  8, LineSize    64
--------********  Unified Cache        9, Level 3,    8 MB, Assoc    16, LineSize   64

Logical Processor to Group Map:
**************** Group 0
```

هنگام کار با معماری NUMA تست کردن اختلاف تکنیکی بین اجزاء دارای اهمیت زیادی می باشد. بدین ترتیب از استفاده فراوان به حافظه در گره های NUMA خارجی اجتناب کنید . حرف گفته شده برای زمان بیکاری است .

۱-۸ تصمیم راحت موازی سازی

گاهی اوقات موازی سازی بهترین انتخاب برای بهینه سازی الگریتم نمی باشد . استفاده از موازی سازی زمانی دارای معناست و به نتیجه می رسد که به یک افزایش کارایی در مقایسه با کد ترتیبی برسیم. در اینجا هیچ گلوله نقره ای[۱] برای تعیین موازی سازی درست یا غلط و وابستگی به ویژگی ها و نیازمندی های عملکردی برای داده های خاص وجود ندارد .برای مثال ، وقتیکه نسخه ترتیبی کد نیاز به کسری از ثانیه برای اجرا دارد این مقدار ناچیزی است که الگریتم موازی سازی زمان مورد نیاز برای تکمیل کاری را ۳۰ درصد کاهش دهد به هر حال ، اگر شما به این نتیجه برسید که برای بهبود کارایی یک فرآیند دسته ای نیاز به ۱۸ ساعت برای پردازش است ، می توانید با موازی سازی کد در کمتری از ۱۳ ساعت اجرایش کنید. همچنین در اجرای موازی ویژگی های اضافی برای بهبود کارایی برنامه های کاربردی موجود تصور کنید ضمنا شما می توانید راه حل هایی را طراحی کنید که در آن توجه بیشتری به استفاده از وظایف ناهمگام[2] و

---

[۱] از فلز نقره در قدیم برای امتهان سمی نبودن غذا استفاده می کردند . سنگ محکی برای صدق گفتار آشپز.

[2] Synchronous Tasks

نخ ها کرده باشد و همچنین از مزیت های موازی سازی استفاده کنید .برنامه نویسی موازی پیچیدگی زیادی را نسبت به برنامه نویسی کلاسیک دارد به هر حالا ، طرح های Task-Based را ایجاد می کنید و کد های موازی را می نویسید. دوری از تفکر مجدد موازی سازی سخت است. مابقی این کتاب توضیحات عمیقی را در مورد امکانات گوناگونی که توسط visual C# 2010 و NET Framework. ارائه می کند. تا طراحی Taske-Based در کد های موازی و حل کردن تمام مشکلاتی که در یک فرآیند ظاهر می شوند را ترجمه کنند .

۱-۹ خلاصه

در این فصل چند پردازنده ای حافظه-مشترک و معماری NUMA را معرفی کردیم و جزئیاتی که درباره موارد زیر شرح می دهد ، همزمانی سبک جدید ومدل های موازی سازی و تا هنگامی که راه حلی طراحی شود وقبل از نوشتن کدی در این فصل درباره قوانین کلاسیک مرتبط با بهینه سازی موازی برنامه و محدودیت های مقیاس پذیری بحث شده است . خلاصه مطالب این فصل :

- ✓ شما الگریتم تان را به Taskهای موازی تقسیم کنید تا از مزیت نخ های سخت افزاری استفاده کنید .
- ✓ شما میتوانید با مدل های ساده تر و کارآمدتر همزمانی کار کنید .
- ✓ شما با احتناب از محدودیت های قانون Amdhel می توانید طرح هایتان را بهبود ببخشید .
- ✓ ممکن است قانون Gustafson را مد نظر قرار دهید .
- ✓ شما مجبورید نواحی بحرانی را حداقل کنید که باعث کاهش اندازه پذیری می شود .
- ✓ شما مجبورید در ذهنتان همزمانی، همزمانی تکه تکه شده و موازی سازی را طراحی کنید .
- ✓ شما باید سربار مورد نیاز کدهای موازی را نیز در نظر بگیرید .
- ✓ شما در کار با معماری NUMA با مشکلات کارایی پیش بینی نشده ای روبرو می شوید .
- ✓ شما هنگامی که قصد کد نویسی موازی را دارید هر مطلبی را درباره معماری سخت افزاری های پیشرفته فرا بگیرید .

۲ فصل دوم

موازی سازی اجباری داده ها

مطالبی که در این فصل فرا می گیرید :

- ✓ درک کامل مسئله موازی سازی داده
- ✓ شروع موازی سازی task ها
- ✓ درک اختلاف بین موازی سازی و همزمانی
- ✓ طرز کار حلقه ها در اجرای موازی با دستور parallel.foreach و parallel.for
- ✓ کار با تکنین های مختلف بخش داده های ورودی در حلقه های موازی
- ✓ کنترل حلقه های موازی
- ✓ تبدیل کد های ترتیبی موجود به کد های موازی
- ✓ اندازه گیری SpeedUP ها و مقیاس پذیری که بوسیله کد موازی ارائه شده است .
- ✓ کار با درجه های مختلف موازی سازی

Visual C# 2010 (C# 4.0) و NET Framework 4. ویژگی هایی هیجان انگیزی را در طراحی موجود عرضه کرده اند تا با پردازنده های چند هسته پیچیده پیوندی ناگسستنی را ایجاد کنند. به هر حال به دلیل شامل شده کامل ویژگی های جدید در این طراحی ها، برنامه نویس ها و طراح ها باید مدل برنامه نویسی جدیدی را فرا بگیرند .

این فصل پیرامون برخی کلاس ها، ساختار ها و داده های شمارشی جدیدی است که به شما اجازه کار با سناریو های موازی سازی داده را می دهد. بجای تمرکز بر روی مشکلات پیچیده مرتبط با برنامه نویسی موازی، این فصل چگونگی ایجاد کد های موازی و توصیف مفاهیم جدید مرتبط با هر سناریو را به شما نشان می دهد. در این شیوه، شما کاملا بهبود هایی را در عملکر درک می کنید.

۱-۲ شروع task های موازی

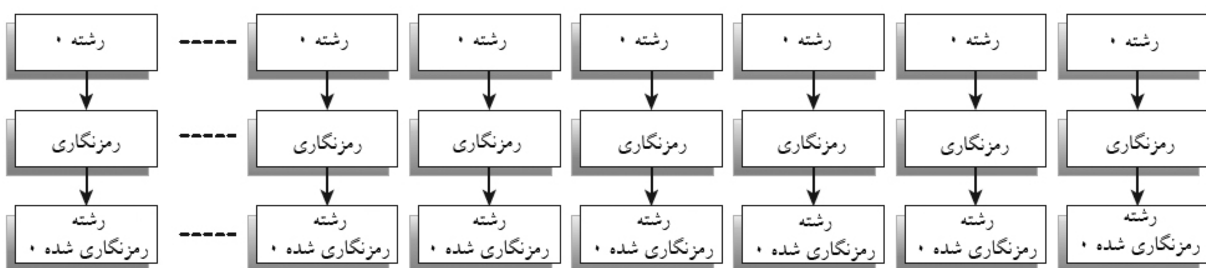
با نسخه های قبلی NET Framework. ، توانایی توسعه برنامه های کاربردی برای استفاده کامل از مزیت های پردازنده های چند هسته مشکل بود. بنابراین لازم بود نخ های متعدد ایجاد، کنترل، مدیریت و هماهنگ شوند این کار با استفاده از ساختار های پیچیده ای که توانایی مدیریت مقداری از همزمانی را داشتند صورت می گرفته اما برای سیستم های چند هسته ای پیشرفته آماده نشده بود.

NET Framework 4. شامل کتابخانه موازی سازی task ها[1] می باشد که برای کار با مدل های همزمانی سبک وزن که در فصل یک با نام " برنامه نویسی Task-Based " شرح داده شد آمده شده است .

پشتیبانی از موازی سازی داده، موازی سازی Taskها و ایجاد مجرای ارتباطی[2] TPL که در چارچوب همزمانی سبک وزن فراهم شده این توانایی را به توسعه دهندگان می دهد تا با سناریو های موازی سازی گوناگون و اجرای طرح های Task-Based که شامل کار با همزمانی سنگین وزن و نخ های پیچیده است کار کنند.

این سناریو ها شامل موارد زیر می باشد :

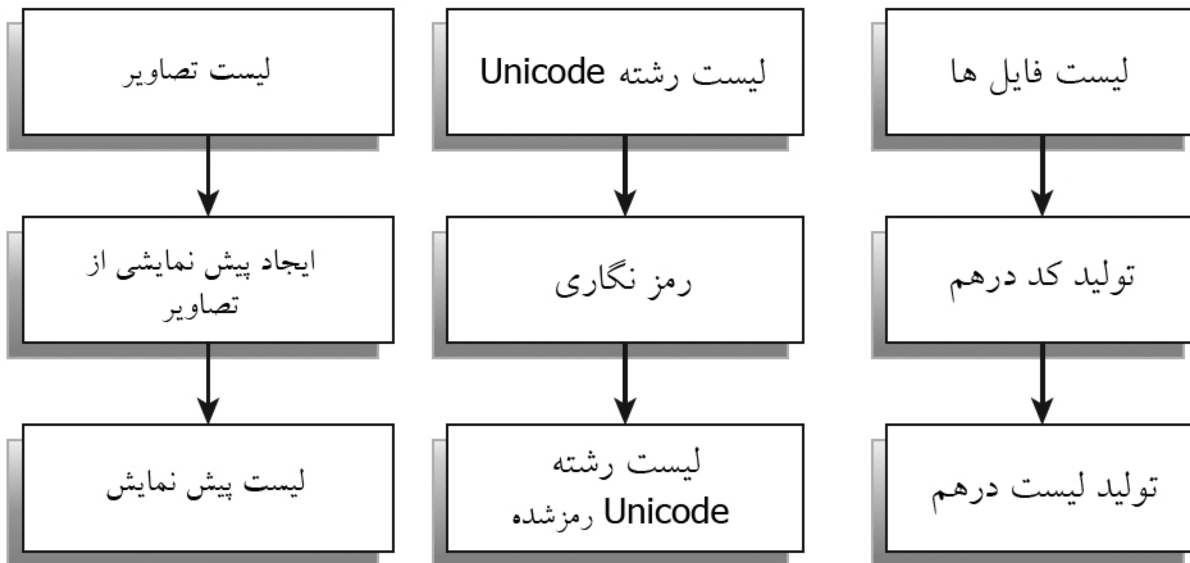
- ✓ موازی سازی داده : حجم بسیاری از داده های و مقدار کمی از عملیاد باید روی هر قطعه انجام شود به طوری که در شکل ۲-۱ می بینید. برای مثال رمزنگاری ۱۰۰ رشته Unicode با استفاده از الگریتم AES[3] و کلید ۲۵٦ بیتی صورت می گیرد.



شکل ۲-۱

[1] Task Parallel Library (TPL)
[2] PipeLining
[3] Advanced Encryption Standard

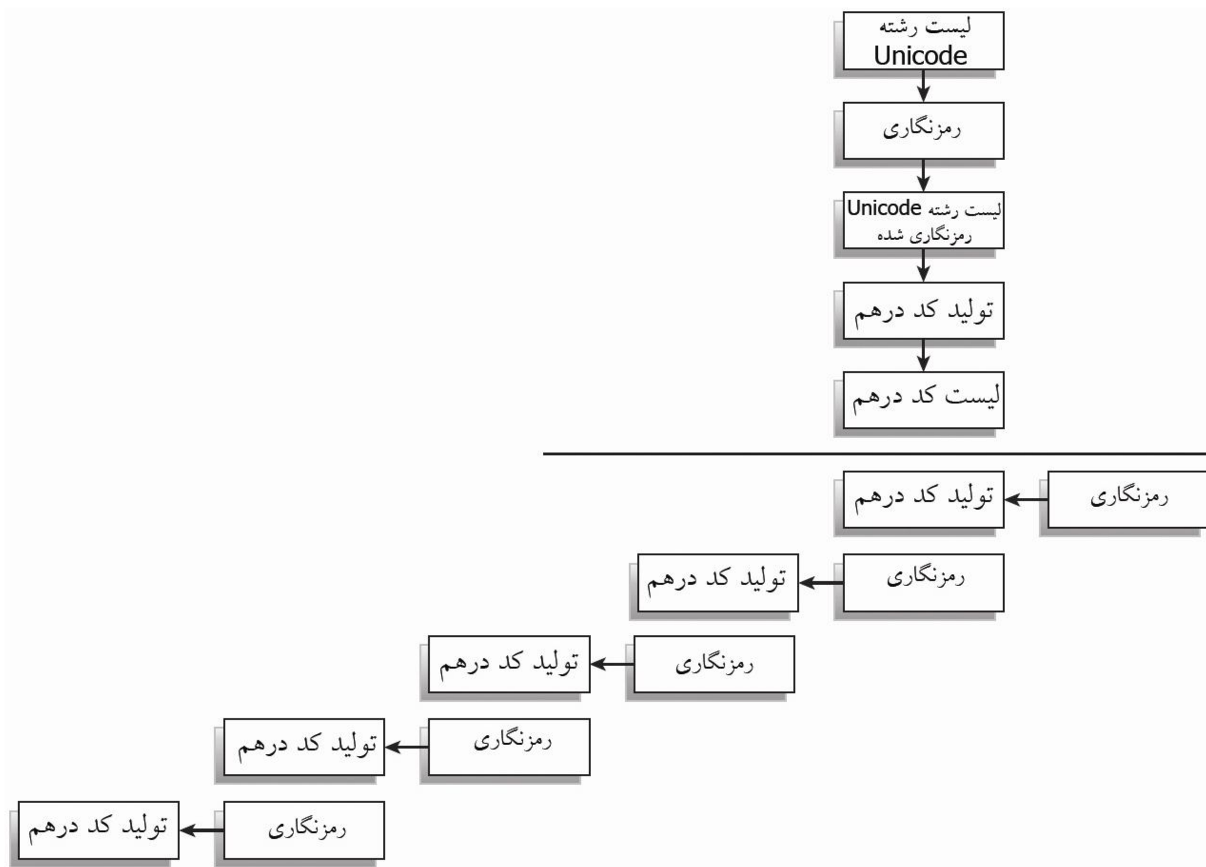
- ✓ موازی سازی Task : عملیات های مختلف بسیاری که بطور همزمان اجرا می شوند از مزیت موازی سازی همانگونه بصوریتی که شکل ۲-۲ می بینید استفاده می کنند. برای مثال تولید کد های درهم برای فایل ها،رمزنگاری رشته های Unicode و ایجاد پیشنمایشی از تصاویر .



شکل ۲-۲

- ✓ ایجاد مجاری ارتباطی : بطوری که در شکل ۲-۳ واضح است، مخلوطی از موازی سازی Task ها و داده هاست. این حالت اکثر سناریو های پیچیده را شامل می شود، زیرا همیشه نیازمند هماهنگی بین همزمانی چندگانه کار های تخصیص یافته است. برای مثال رمز نگاری ۱۰۰ رشته Unicode با کلید های ۲۵٦ بیتی و سپس تبدیل به کدی درهم برای تمامی رشته های رمزنگاری شده. این خط-لوله می تواند اجرای دو Task را همزان تکمیل کند : رمز نگاری و تولید کد درهم. هر رشته Unicode رمز شده به ترتیبی که توسط الگریتم hash-code-generation پردازش شده وارد صف می شود.

همچنین سناریو های ترکیب شده ای هستند که یکی از موارد ذکر شده را ترکیب می کند. ساده ترین شیوه برای درک چگونگی کار با Taskهای موازی را در حین کار کردن با آنها می توانید یاد بگیرید. سناریو های رایج بیشتری با جزئیات مثال هایشان در فصل های بعدی توضیح داده می شود.



شکل ۲-۳

۲-۱-۱ کلاس System.Threading.Tasks.Parallel

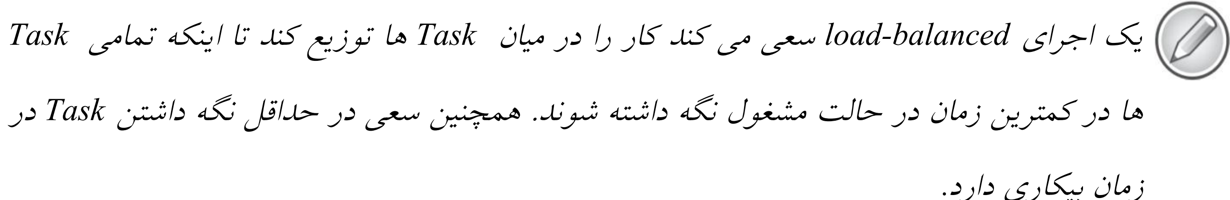
TPL فضای نام جدید System.Threading.Tasks را معرفی می کند . این فضای نام، دسترسی به کلاس ها، ساختار ها و داده های شمارشی که توسط NET Framework 4. معرفی شده را ارائه می کند. بنابراین استفاده از این فضای نام هنگامی که می خواهید با Task ها کار کنید ایده خوبی است .

```
using System.Threading.Tasks;
```

در این شیوه ، از مرجع های زیادی می توانید اجتناب کنید. برای مثال به جای نوشتن System.Threading Tasks.Parallel.Invoke. می توانید از Parallel.Invoke استفاده کنید. بدین ترتیب برنامه ساده شده است، فضای نام Parallel.Invoke را در کدهای این فصل در ابتدای برنامه قرار دهید.

کلاس اصلی Task است، که عملیات همزمانی و ناهمگام[1] را ارائه می دهد. بهر حال، برای ایجاد کد های موازی لازم نیست بطور مستقیم با Taskهای نمونه کار کنید. برخی اوقات بهترین انتخاب ایجاد حلقه ها یا ناحیه های موازی است. در این حالت بجای کار با نمونه Taskهای سطح پایین با متد های که در ادامه شرح داده شده و توسط parallel از کلاس پویای System.Threading.Tasks.Parallel ارائه شده است استفاده کنید :

- ✓ Parallel.For : ارائه load-balanced ، که بطور بلقوه اجرای موازی تعدا ثابت تکرار در حلقه های مستقل For را پشتیبانی می کند.



یک اجرای load-balanced سعی می کند کار را در میان Task ها توزیع کند تا اینکه تمامی Task ها در کمترین زمان در حالت مشغول نگه داشته شوند. همچنین سعی در حداقل نگه داشتن Task در زمان بیکاری دارد.

- ✓ Parallel.ForEach : load-balanced ی را ارائه می دهد که بطور بلقوه اجرای موازی تعداد ثابتی از تکرا حلقه های مستقل ForEachرا برعهده دارد. این متد از جدا کننده های سفارشی ای پشتیبانی می کند که شما را قادر به کنترل نهایی در توزیع داده می سازد.
- ✓ Parallel.Invoke : بطور بلقوه اجرای موازی ای را ارائه می دهد که توسط فعالیت های مستقلی فراهم شده است .

زمانی که کد موجود بخواهد از مزیت های موازی سازی استفاده کند، این متد ها بسیار مفید واقع می شود. فهمیدن این نکته بسیار مهم است که جایگزینی یک عبارت For بجای Parallel.For به این سادگی ها نیست.

۲-۱-۲ Parallel.Invoke

ساده ترین شیوه برای اجرای موازی تعدادی تابع استفاده از تابع Invoke جدیدی است که توسط کلاس Parallel فراهم شده است. برای مثال، فرض کنید چهار تابع مستقل که در ادامه ذکر شده است را دارید که

[1] asynchronous

کار تبدیل فرمت را انجام می دهند و شما قصد دارید از امنیت اجرای آنها هنگامی که بصورت موازی اجرای می شوند را دارید :

- ✓ ConvertEllipses
- ✓ ConvertRectangles
- ✓ ConvertLines
- ✓ ConvertText

با استفاده از کد زیر می توانید از مزیت های موازی سازی بهره ببرید و توجه داشته باشید که ابتدا از تابع هایی استفاده کنید که پارامتر های کمتری برگشت می دهد یا مقداری برگشتی آن Voidباشد :

```
Parallel.Invoke(ConvertEllipses,
        ConvertRectangles,
        ConvertLines,
        ConvertText);
```

تکه کد ۲-۱

در این حالت، کد Deligate ی که اشاره به هر متد دارد را ایجاد می کند. در تعریف متد Invoke مقادیر دریافتی آرایه های از از نوع Action (System.Action [ ]) تعریف شده تا بصورت موازی اجرا شود .

کد زیر نتایج یکسانی را بوسیله تعریف عبارات Lambda و اجرای تابع ها انجام می دهد :

```
Parallel.Invoke(() => ConvertEllipses(),
        () => ConvertRectangles(),
        () => ConvertLines(),
        () => ConvertText());
```

تکه کد ۲-۲

به طوری که در مثال زیر می بینید، با استفاده از عبارات Lambda، Delegate های بدون نام تابع ها را اجرا کنید .

```
Parallel.Invoke(
        () =>
        {
                ConvertEllipses();
                // Do something else adding more lines
        },
        () =>
        {
                ConvertRectangles();
                // Do something else adding more lines
        },
        delegate()
```

```
{
        ConvertLines();
        // Do something else adding more lines
},
delegate()
{
        ConvertText();
});
```

از مزیت های بزرگ استفاده از عبارات Lambda و Delegate های بدون نام، تعریف تابع پیچیده در چندین خط برای اجرای موازی است بدون اینکه نیازی به تعریف تابع ای اضافی باشد. اگر قصد دارید با TPL برنامه ای موازی بنویسید، این نکته بسیار مهم است که از Delegate های اصلی و عبارات Lambda در جاهای مناسب استفاده کنید .

۲-۱-۲-۱ بدون ترتیبی خاص، در اجرا

در ادامه هر یک از کد های که قبلا درج شده بود شرح داده می شود. در کد های که مشاهده نمودید تا زمانیکه تمام چهار تابع کارشان کامل نشود تابع Parallel.Invoke هیچ مقدار را برگشت نمی دهد. در طول فرآیند تکمیل توابع ممکن است استثنائاتی[1] نیز رخ دهد.

تابع ها سعی می کنند تا بطور همزمان هر چهار تابع اجرا شوند تا از مزیت های هسته های منطقی که توسط یک یا چند ریز پردازنده ارائه می شود سود ببرند. اجرای همزمانی واقعی به فاکتور های زیادی بستگی دارد. در این حالت، چهار متد در اینجا داریم. این مطلب بدین معنی است که Parallel.Invoke برای اجرای همزمان چهار تابع به کمتر از هسته منطقی در دسترس نیاز دارد. هسته های منطقی معرف به نخ های سخت افزاری اند که در فصل یک شرح داده شده است. بخاطر بیاورید که یک موضوع پر اهمیت این بود که درک کنید سخت افزاری های چند هسته ای جدید کد های موازی را چگونه اجرا می کنند.

داشتن چهار هسته منطقی ضمانتی بر اجرای چهار تابع به طور همزمان نیست. اگر یک هسته یا بیشتر مشغول باشد، زمانبند منطقی سیستم عامل در اجرای اولیه برخی از توابع تاخیری می تواند اضافه کند.

[1] Exceptions

حقیقتاً پیشگویی درست ترتیب اجرای تابع ها خیلی مشکل است، زیرا سعی منطق سیستم عامل این است که مناسب ترین طرح اجرا، بر اساس منابع در دسترس در زمان اجرا را انتخاب کند .

در شکل ۲-٤ فقط سه سناریوی ممکن اجرای موازی را نشان می دهد که بر اساس پیکره بندی سخت افزاری یا بارهای کاری مختلف می توان رخ دهد. نکته ای این است که برخی کد ها نیازی به اجرا در زمان ثابتی ندارند بنابراین، تابع ConvertText نسبت به ConvertLines زمان بیشتری نیاز دارد حتی با پیکره بندی سخت افزاری یکسان و جریان داده های ورودی مساوی .

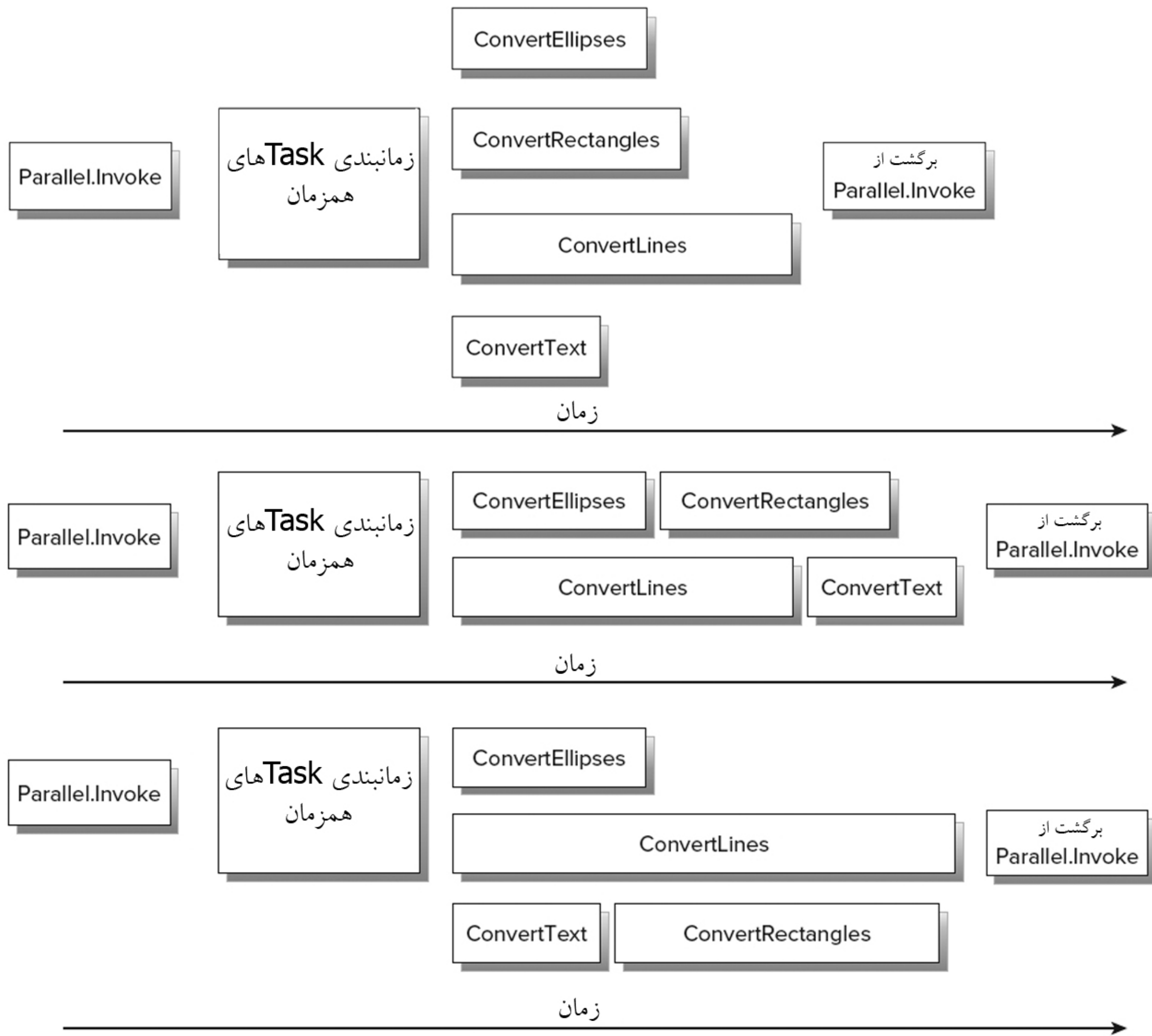
اولین نمودار وضعیت تقریبا دلخواهی را ارائه می دهد : چهار تابع موازی اجرا می شوند. در نظر گرفتن زمان مورد نیاز برای زمانبندی Taskهای همزمان و اضافه کردن سربار جدید به زمان کل اهمیت بسیاری دارد .

نمودار دوم سناریو های گوناگونی را نشان می دهد و در آن تنها دو مسیر همزمان و چهار تابع برای اجرا وجود دارد. در یکی از مسیر ها زمانی که ConvertEllipses به پایان می رسد ConvertRectangles شروع می شود و در مسیر دیگر زمانی که ConvertLines کارش تمام شد، ConvertText شروع می شود Parallel.Invoke نسبت به سناریوی قبلی برای اجرای تمام تابع ها زمان بیشتری می گیرد.

نمودار سوم سناریوی دیگری را با سه مسیر همزمان نشان می دهد که تقریبا زمان یکسانی با نمودار دوم صرف می کند زیرا، در این حالت ConvertLines زمان بیشتری برای اجرا نیاز دارد. بنابراین، تقریبا Parallel.Invoke در زمان برابری نسبت به سناریوی قبلی تمام توابع را اجرا می کند حتی اگر مسیر موازی ای اضافه شود.

کد که بصورت موازی با استفاده از Parallel.Invoke برای اجرا شدن نوشته شده نباید به ترتیب اجرای خاصی تکیه کند. اگر شما کدی داشته باشد که نیاز به ترتیب اجرا موازی خاصی دارد را داشته باشید ، باید با روش های موازی سازی دیگری که توسط TPL ارئه شده است را کار کنید. در این باره در ادامه فصل با موضوعات پیشرفته ای آشنا می شوید.

درک مشکل موجود در ترتیب اجرا بسیار ساده است، اگر برنامه های کنسولی که در لیست ۲-۱ آمده است را اجرا کنید مشاهده می کنید که زمان زیادی روی یک کامپیوتر با دسته کم دو هسته فیزیکی نیاز دارد .



شکل ۲-٤

لیست ۲-۱ : برنامه کنسول ساده ای است که مشکل ترتیب اجرا را در همزمانی کد در Parallel.Invoke را نشان می دهد.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Listing2-1
{
	class Program
	{
		static void Main(string[] args)
```

```
        {
            Parallel.Invoke(
            () => ConvertEllipses(),
            () => ConvertRectangles(),
            () => ConvertLines(),
            () => ConvertText());
            System.Console.ReadLine();
        }
        static void ConvertEllipses()
        {
            System.Console.WriteLine("Ellipses converted.");
        }
        static void ConvertRectangles()
        {
            System.Console.WriteLine("Rectangles converted.");
        }
        static void ConvertLines()
        {
            System.Console.WriteLine("Lines converted.");
        }
        static void ConvertText()
        {
            System.Console.WriteLine("Text converted.");
        }
    }
}
```

برای مثال، نسخه کد ترتیبی ای که از Parallel.Invoke در زیر هست در نظر بگیرید :

```
ConvertEllipses();
ConvertRectangles();
ConvertLines();
ConvertText();
```

نتیجه خروجی نوشته شده در Console در کامپیوتر های چند هسته ای مختلف یکسان است زیرا ترتیب اجرا دقیقا یکی است. در زیر خروجی را مشاهده می کنید :

```
Ellipses converted.
Rectangles converted.
Lines converted.
Text converted.
```

اگر Parallel.Invoke را حتی با سخت افزاری یکسان اجرا کنید، نتایج متفاوتی بدست می آورید. لیست ۲-۲ شامل نتایج نشان داده شده در برنامه Console ای است که در سه زمان در همان کامپیوتر چند هسته ای اجرا شد.

لیست ۲-۲ :نتایج اجرای کد موازی یکسانی که زمان های مختلفی برای اجرا نیاز دارد و در لیست ۲-۱ نشان داده شده است .

FIRST TIME

Ellipses converted.
Rectangles converted.
Lines converted.
Text converted.

SECOND TIME

Ellipses converted.
Lines converted.
Rectangles converted.
Text converted.

THIRD TIME

Ellipses converted.
Lines converted.
Text converted.
Rectangles converted.

در وحله اول کد همزمان خروجی یکسانی را نسبت به اجرای موازی تولید می کند. در زمان های دوم و سوم اجرا شدن توابع ترتیب های متفاوتی دارد. برای مثال در زمان سوم، بخاطر شروع در اولین ثانیه rectangles آخرین خروجی است. در این مثال ساده یکی از اختلاف ها بین کد های ترتیبی و موازی اثبات شد.

۲-۱-۲-۲ برتری ها و سبک و سنگین کردن ها[1]

برتری کلیدی استفاده از Parallel.Invoke در روش ساده اجرای موازی تابع های بسیاری بدون نگرانی درباره نخ ها و Taskهاست. البته با همه سناریوها مناسب نیست. در ادامه مزایا و معایب Parallel.Invoke سبک سنگین شده است :

[1] Trade-Offs

- ✓ اگر شما از Parallel.Invoke برای به کار بردن توابع ای که نیاز به زمان های اجرایی بسیار گوناگونی دارند استفاده کنید، شما به زمان بسیار بیشتری نیاز دارید تا کنترل به کدتان برگردد. هسته های منطقی زیادی که برای مدت طولانی به حالت بیکار رفته اند توسط این شیوه رهبری می شوند. بنابراین، این موارد از اهمیت زیادی برخودار است: میزان نتایجی که مورد استفاده قرار می گیرد، SpeedUP های بدست آمده و استفاده از هسته های منطقی.
- ✓ این روش محدودیت های را به موازی سازی تحمیل می کند. دلیل آن هم فراخوانی تعداد ثابتی از Delegate هاست. اگر مثال قبلی را در کامپیوتری با ١٦ هسته اجرا کنید تنها چهار تابع بصورت موازی اجرا می شوند بنابراین ١٢ هسته دیگر در حالت بیکار می مانند.
- ✓ قبل از اجرای موازی تابع ها فراخوانی شان سربار اضافی ای را ایجاد می کند .
- ✓ در تمام کد های موازی شده، وابستگی ها و تعامل های کنترل نشده بین تابع های مختلف هدایت می شود تا خطا های همزمانی گوناگون و اثرات مختلف پیشبینی نشده تشخیص داده شود. این سبک سنگین کردن بر کد های موازی اثر می گذارد نه فقط بر مشکلات استفاده از Parallel.Invoke. این کتاب به شما در یافتن مکانیزم هایی برای حل بهینه تر مشکل ها کمک می کند.
- ✓ در اینجا ضمانتی برای ترتیب اجرا کد ها نیست، بنابراین Parallel.Invoke انتخاب مناسبی برای برای اجرا الگریتم های پیچیده ای که نیاز به طرح اجرای خاصی در تابع های همزمان دارند نمی باشد .
- ✓ استثنائات بوسیله استفاده Delegate های که از طرح های مختلف اجرای موازی شروع می شوند از رخ می دهند. بنابراین کد های که دارای قسمت Catch اند و وقوع استثنائات را مدیریت می کنند نسبت به کد های ترتیبی سنتی دارای پیچیدگی بیشتری اند.

همانطور که ذکر شد استفاده از Parallel.Invoke دارای مزیت ها و معایبی بود که در مثال هایی شرح داده شد. اگر چه مکانیزم های گوناگونی ترکیب می شوند تا تعدای زیادی از مشکلات را حل کند، ولی شما می توانید مکانیزم های زیادی که در این فصل و ادامه کتاب آورده شده است را یاد بگیرید .

۲-۱-۲-۳ همزمانی و همزمانی تکه تکه شده

چنانچه در مثال قبلی و شکل ۲-۵ مشاهده نموده اید، همزمانی و همزمانی تکه تکه شده با هم اختلاف دارند.

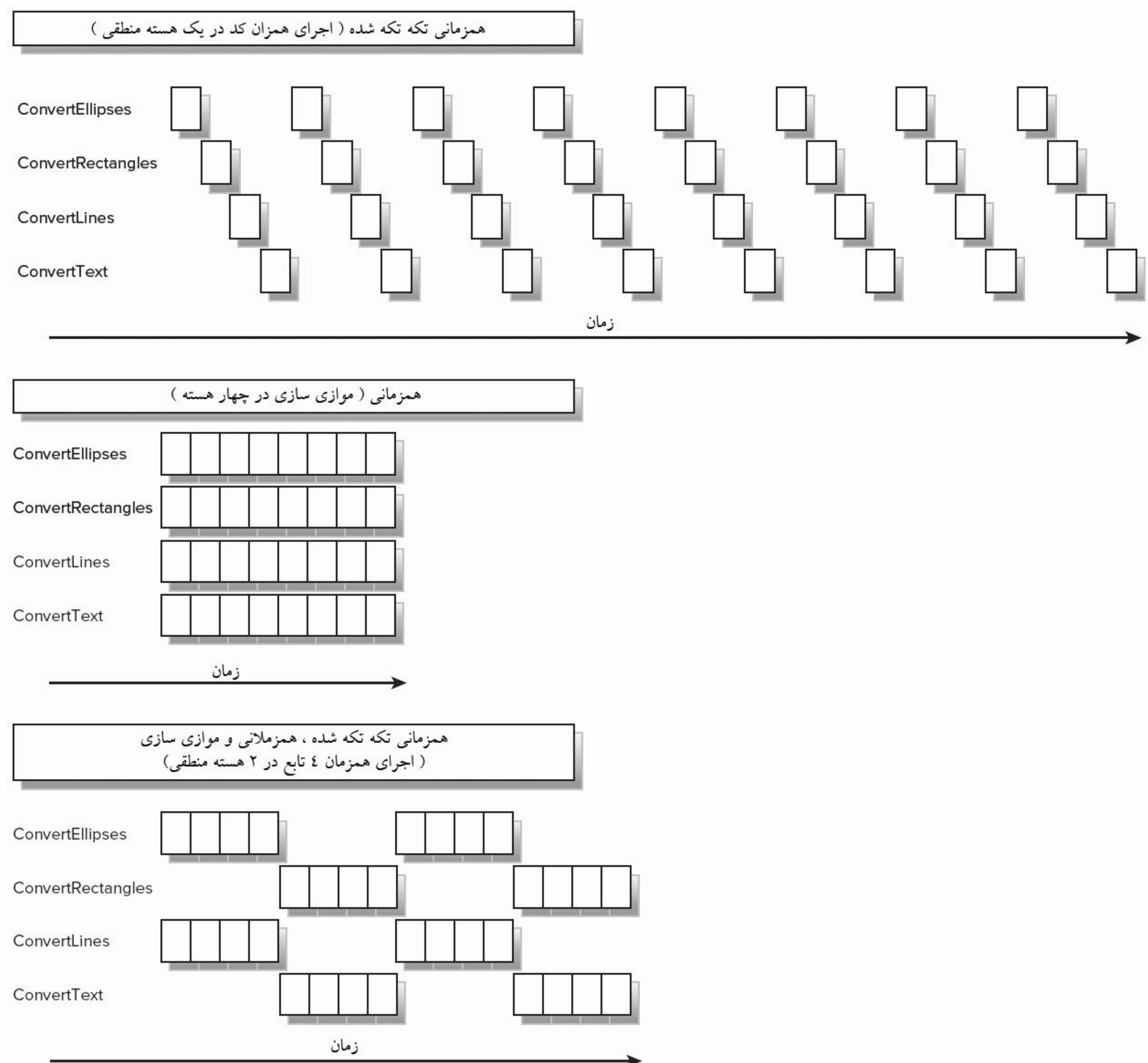
همزمانی تکه تکه شده می تواند اجزاء مختلفی از کد را شروع، اجرا و در دوره های تداخلی و متناوب از زمان کامل کند. این همزمانی حتی در کامپیوتر های با یک هسته منطقی نیز رخ می دهد. در همزمانی تکه تکه شده، زمانیکه قطعه های زیادی از کد در کامپیوتر هایی با یک هسته منطقی اجرا می شوند، مکانیسم time slicing[1] و سرعت سوئیچ برنامه ها، برداشت اجرای موازی را تلقین می کند. در این سخت افزار ها زمان زیادی در همزمانی تکه تکه شده برای اجرای تکه های زیادی از کد ها به نسبت اجرای تنها یک تکه کد صرف می شود. زیرا چنانچه در نمودار قبلی نشان داده شد کد های همزمان برای دستیابی به منابع سخت افزاری رقابت می کنند. مثالی برای فهم بهتر همزمانی تکه تکه شده این است که، ماشین های زیادی در یک مسیر باریک شریک اند. از همزمانی تکه تکه شده به عنوان موازی سازی مجازی نیز تعبیر می شود . همزمانی به این معنی است که تکه های بسیاری از کد بصورت پیوسته می توانند اجرا شوند. همانطور که در فصل یک توضیح داده شد استفاده از قابلیت های واقعی پردازش موازی در سخت افزار نهفته است. همزمانی در کامپیوتری با یک هسته منطقی اتفاق نمی افتد. برای اجرای موازی کد حداقل به دو هسته منطقی نیاز هست. زمانیکه از موازی سازی وقعی استفاده می کنید به SpeedUP ای بالاتری دست می یابید، زیرا تکه های زیادی از کد که بصورت موازی اجرا می شوند که زمان کل برای کامل شدن الگریتمی خاص را کاهش می دهند. نمودار پیشین سناریو های موازی زیر را ارائه کرد :

- ✓ همزمانی : انجام موازی سازی در چهار هسته منطقی (چهار کوچه) ـــ در وضعیت ایده آل، دستور العمل های هر کدام از این چهار تابع در هسته منطقی متفاوتی اجرا می شوند .
- ✓ ترکیب همزمانی تکه تکه شده و همزمانی : موازی سازی ناقص با چهار تابع باعث استفاده از مزیت دو هسته منطقی می شود (دو کوچه) ـــ گهگداری دستور العمل های هر چهار تابع بصورت موازی

---

[1] برش زمانی

در چهار هسته منطقی متفاوت اجرا می شود و در برخی زمان ها مجبور به صبر کردن برای یک Time-Slice اند. در چنین موردی همزمانی با موازی سازی ترکیب شده است. این رایج ترین وضعیت است، زیرا حقیقتاً دستیابی کامل به موازی سازی خیلی سخت است، حتی در یک سیستم عامل بلادرنگ[1].



شکل ۲-۵

---

[1] Real Time operating system (RTOS)

زمانیکه برخی از اجزاء کد عیناً در زمان یکسانی اجرا می شوند، اشکال[1] های جدید ممکن است ظاهر شود. خوشبختانه TPL ویژگی های بسیاری در ساختار ها و اشکال زدایی[2] ارائه نموده که برای اجتناب از کابوس های موازی سازی به ما کمک می کند. شما قادر به ترکیب هر آنچه که درباره Task موازی الزامی در این فصل با اضافه کردن مکانیزم های که در فصل های آینده شرح داده شده فرامی گیرید تا ارائه برنامه های موازی را بچالش بکشانید .

۲-۳ تبدیل کد های ترتیبی به کد های موازی

در دهه قبلی، اکثر کد های C# به شیوه ترتیبی و اجرای همگام نوشته می شد. بنابراین اکثر الگریتم با استفاده از تکنیک های همزمانی یا موازی سازی در ذهن طراحی نمی شد. در اکثر زمان ها، شما الگریتمی که بطور کامل به کد های دقیقا موازی تبدیل شده باشند را نمی توانید پیدا کنید. (این امکان وجود دارد که چنین الگریتمی را بیابید اما نه در سناریو های رایج.)

زمانی که کدی ترتیبی دارید و می خواهید از مزیت های بالقوه موازی سازی برای دستیابی به SpeedUP بالاتر استفاده کنید. مجبورید تا hotspot قابل موازی سازی بیابید. سپس شما آن را به کدی موازی برای SpeedUP بالاتر و تشخیص اندازه بالقوه تبدیل کنید تا مطمئن شوید اشکالات جدیدی وجود ندارد تا زمانی که کد های ترتیبی را به کد های موازی تبدیل می کنید .

Hotspot جزئی از کد می باشد و عمده زمان اجرا را میگیرد و تنگنایی برای عملکرد الگریتم به حساب می آید. یک Hotspot قابل موازی سازی اگر به تکه های زیادی تقسیم و اجرا شود به SpeedUP بالاتری دست می یابد. Hotspot ها اجباراً کد ها را به تکه های زیادی(حداقل دو) تقسیم می کنند تا بدین ترتیب از منفعت های موازی سازی بهرمند شوند. اگر قسمتی از کد عمده زمان اجرا را بخود اختصاص ندهد، سرباری که توسط TPL شرح داده شد در اکثر زمان باعث کاهش SpeedUP، حتی تا 0x می شود در این صورت کد موازی شده کندتر از نسخه ترتیبی اجرا

---

[1] Bug
[2] Debugging

می شود. هنگامی که شروع به کار با امکانات گوناگون TPL می کنید قادر به تشخیص Hotspot قابل موازی سازی در کد ترتیبی هستید.

۲-۲-۱ تشخیص Hotspot قابل موازی سازی

لیست ۲-۳ مثالی ساده از یک برنامه کاربردی Console می باشد که با دو کد ترتیبی در زیر اجرا شده است:

- ✓ GenerateAESKeys : حلقه ای For را برای تولید اعدادِ کلید های AES خاص توسط ثابت NUM_AES_KEYS اجرا می کند. تابع GenerateKey توسط کلاس System.Security. Cryptography.AesManaged در دسترس است. هنگامی که کلید تولید شد نتایج تبدیل آرایه Byte به رشته Hexadecimal[1] در متغیر محلی hexString قرار داده می شود (ConvertToHexString).
- ✓ GenerateMD5Hashes : حلقه Forی را برای محاسبه تعداد لیست ها اجرا می کند، با استفاده از الگریتم ADE و توسط ثابت خاص NUM_MD5_HASHES. از نام کاربری به عنوان پارامتی برای فراخوانی تابع ComputeHash استفاده می کند و از کلاسی که در مسیر است قابل دسترسی می باشد. زمانیکه لیست تولید شد نتایج تبدیل آرایه Byte به رشته ۱٦ بیتی (ConvertToHexString) ذخیره کرده و در متغیر محلی (hexString) قرار داد .

لیست ۲-۳ : سری ساده ای از کلید های AES و لیست مولد های MD5

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

        // Added for the Stopwatch
using System.Diagnostics;
        // Added for the cryptography classes
using System.Security.Cryptography;
        // This namespace will be used later to run code in parallel
using System.Threading.Tasks;
```

[1] مبنای ۱٦

```
namespace Listing2_3
{
	class Program
	{
		private const int NUM_AES_KEYS = 800000;
		private const int NUM_MD5_HASHES = 100000;

		private static string ConvertToHexString(Byte[] byteArray)
		{
			// Convert the byte array to hexadecimal string
			var sb = new StringBuilder(byteArray.Length);
			for (int i = 0; i < byteArray.Length; i++)
			{
				sb.Append(byteArray[i].ToString("X2"));
			}
			return sb.ToString();
		}

		private static void GenerateAESKeys()
		{
			var sw = Stopwatch.StartNew();
			var aesM = new AesManaged();
			for (int i = 1; i <= NUM_AES_KEYS; i++)
			{
			aesM.GenerateKey();
			byte[] result = aesM.Key;
			string hexString = ConvertToHexString(result);
			// Console.WriteLine("AES KEY: {0} ", hexString);
			}
			Debug.WriteLine("AES: " + sw.Elapsed.ToString());
		}

		private static void GenerateMD5Hashes()
		{
			var sw = Stopwatch.StartNew();
			var md5M = MD5.Create();

			for (int i = 1; i <= NUM_MD5_HASHES; i++)
			{
				byte[] data =
				Encoding.Unicode.GetBytes(
				Environment.UserName + i.ToString());
				byte[] result = md5M.ComputeHash(data);
				string hexString = ConvertToHexString(result);
```

```
                    // Console.WriteLine("MD5 HASH: {0}", hexString);
                }
                Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
            }

            static void Main(string[] args)
            {
                var sw = Stopwatch.StartNew();
                GenerateAESKeys();
                GenerateMD5Hashes();
                Debug.WriteLine(sw.Elapsed.ToString());
                //    Display the results and wait for the user to press a key
                Console.ReadLine();
            }
        }
}
```

حلقه For در تابع GenerateAESKeys از متغیر i برای کنترل استفاده نمی کند، زیرا فقط تعداد مکان های که کلید تصادفی AES تولید می شود را کنترل می کند. حلقه For در تابع GenerateMD5Hashes از متغیر i برای کنترل در اضافه کردن عددی به نام کاربری کامپیوتر مورد استفاده قرار گرفته است. سپس این رشته بعنوان ورودی داده در فراخوانی تابع که شامل لیستی از کامپیوتر هاست مورد استفاده قرار می گیرد که در کد زیر نشان داده شده است .

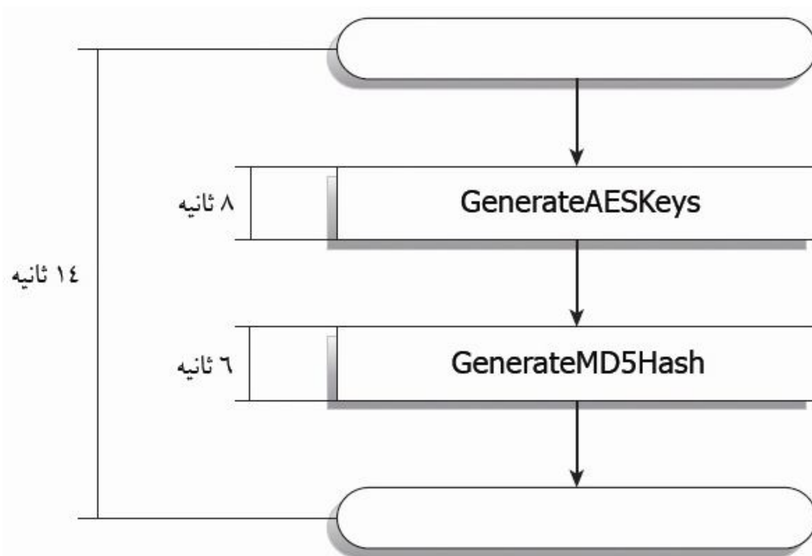
```
for (int i = 1; i <= NUM_MD5_HASHES; i++)
{
        byte[] data = Encoding.Unicode.GetBytes(Environment.UserName + i.ToString());
        byte[] result = md5M.ComputeHash(data);
        string hexString = ConvertToHexString(result);

        // Console.WriteLine(hexString);
}
```

تکه کدی از لیست ۲-۳

خطوط پررنگ شده بالا به لیست ۲-۳ اضافه شده است تا زمان اجرای هر تابع و زمان سپری شده کل را اندازه گیری کند. با فراخوانی StartNew در ابتدای هر تابع، Stopwatch جدیدی را آغاز می شود و سپس زمان سپری شده کل را در خروجی Debug درج می شود.

همچنین در لیست ۲-۳ خطوطی نوشته شده که کلید های را تولید کرده و لیست از کد های که به توضیحاتی(// در ابتدای خط) تبدیل شده اند درج شده است. عملیات ارسال رشته ها به Console باعث تولید تنگنایی می شود که ممکن است مسبب اختلال در دقت اندازه گیری زمان شود .
شکل ۲-٦ روند اجرای ترتیبی را برای این برنامه کاربردی نشان می دهد زمان اجرای هر دو تابع ذکر شدن در کامپیوتری با ریزپردازنده dual-core را می گیرد .

GenerateMD5Hashes و GenerateAESKeys تقریبا به ١٤ ثانیه برای اجرا نیاز دارد.اولی هشت ثانیه و آخری شش ثانیه برای اجرا زمان میگرد. بدیهی است که، این زمان ها موضوع تغییرات عظیمی برطبق پیکره بندی نهفته سخت افزاری می باشد .در اینجا دو تابع تعاملی با یکدیگر ندارند. بنابراین، بطور کامل از یکدیگر مستقل اند. توابعی که بطور ترتیبی یکی پس از دیگری اجرا می شوند ، از مزیت های که توسط پردازش موازی و بوسیله هسته های اضافه شده در اختیارشان قرار می گیرد استفاده ای نمی کنند. بدین دلیل، این دو تابع hotspot واضحی را ارائه می دهند در سورتی که موازی سازی به دستیابی به SpeedUP بالاتری نسبت به اجرای ترتیبی کمک می کرد . برای مثال می توانید هر دو تابع را با استفاده از Parallel.Invoke اجرا کنید .



شکل ۲-۶

۲-۲-۲ اندازه گیری موفق SpeedUP توسط اجرای موازی

تابع اصلی که در لیست ۲-۳ نشان داده شده با نسخه جدید زیر جایگزین کنید، که از Parallel.Invoke برای شروع موازی دو تابع GenerateAESKeys و GenerateMD5Hashes استفاده می شود .

```
static void Main(string[] args)
{
        var sw = Stopwatch.StartNew();

        Parallel.Invoke(
                () => GenerateAESKeys(),
                () => GenerateMD5Hashes());

        Debug.WriteLine(sw.Elapsed.ToString());
        //  Display the results and wait for the user to press a key
        Console.WriteLine("Finished!");
        Console.ReadLine();
}
```

تکه کدی از لیست ۲-۳ ـ با استفاده از Parallel.Invoke

شکل ۲-۷ روند اجرای موازی را برای نسخه جدیدی از برنامه کاربردی و همچنین زمان اجرای هر دو تابع در کامپیوتری با یک ریزپردازنده Dual-Core را نشان می دهد.

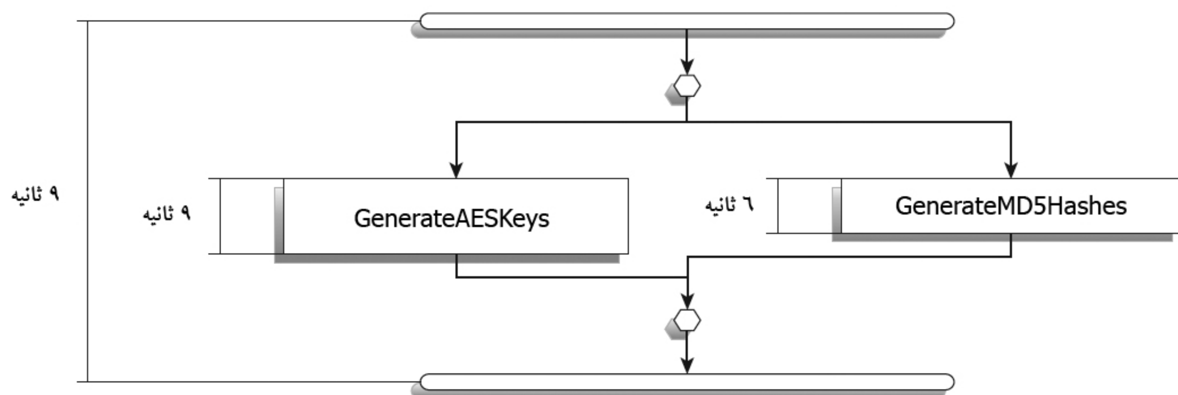
حالا GenerateAESKeys و GenerateMD5Hashes تقریباً به نه ثانیه برای اجرا احتیاج دارند، زیرا آنها از هر دوهسته پردازنده استفاده کرده اند. بنابراین SpeedUP را می توانید از فرمول زیر محاسبه کنید :

$$\text{Speedup} = \frac{\text{زمان اجرای ترتیبی}}{\text{زمان اجرای موازی}}$$

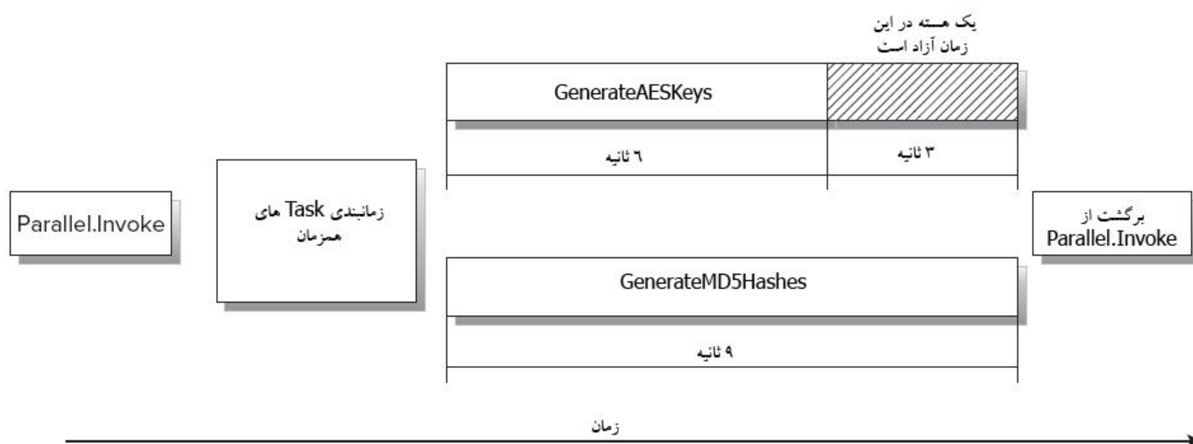
که در این حالت عدد زیر بدست می آید:

$$\frac{14}{9} = 1.56\text{x} \text{ بیشتر از کد ترتیبی}$$

بطوریکه می بینید، GenerateAESKeys زمان بیشتری نسبت به GenerateMD5Hashes برای اجرا می گیرد نه ثانیه در مقابل شش ثانیه. با توجه به شکل ۲-۸ تا تمام Delegate ها اجرایشان پایان نیابد تا Parallel.Invoke مقداری را برگشت دهد .بنابرایت در طی سه ثانیه برنامه از یک هسته که دارای مشکل load-imbalance است نمی تواند استفاده کند.



شکل ۲-۷



شکل ۲-۸

اگر این برنامه در رایانه با ریزپردازنده quad-cor اجرا شود، SpeedUP آن تقریبا برابر نسخه ترتیبی خواهد بود زیرا از مزیت دو هسته اضافه شده در سخت افزار نهفته استفاده ای نمی کند.

در این مثال، مجموع hotspot های مقداری کدی قابل موازی سازی را برای اندازه گیری زمان باقی مانده برای اجرای قطعی تابع تشخیص داده اید. سپس به تغییراتی در جهت بهبود SpeedUP با استفاده از چند خط کد دست یافتید. حالا لازم است در ساختارهای TPL موازی سازی داده های ضروری را بشناسید تا نتایج بهتری به ثمر بنشیند و زمانیکه تعداد هسته های در دسترس در سناریوهای موازی داده افزایش می یابد، اندازه بهتری را ارائه دهید

در شروع کار با Class ها، Struct ها و توابع مقدار دهی اولیه TPL لازم نیست. TPL خیلی مخفیانه کار می کند وشامل بهترین بهینه سازی است همچنین از مکانیزم های زمانبندی برای

کسب منفعت سخت افزار های اساسی به بصورت بلادرنگ استفاده می کند. انتخاب ساختار صحیح موازی سازی برای یک hotspot یک وظیفه مهم می باشد.

۲-۲-۳ درک اجرای همزمان

بعداً، به خطوطی غیر وابسته نیاز پیدا می کنید که در هر دو تابع GenerateAESKeys و GenerateMD5Hashes خروجی را به Console بفرستد :

```
Console.WriteLine("AES KEY: {0} ", hexString);
Console.WriteLine("MD5 HASH: {0}", hexString);
```

نوشتن در Console باعث ایجاد تنگنا[1]یی در اجرای موازی است. فعلا به اندازه دقیق زمان نیازی نداریم. درعوض، خروجی هر دو تابع را بصورت موازی می توانید تصور کنید. لیست ۲-٤ مثالی را از خروجی Console که توسط این برنامه اجرایی تولید شده را نمایش می دهد. کوتاهترین رشته ۱٦ بیتی (حالت Bold در لیست) با لیست MD5 برابرند. رشته های مبنای ۱٦ کلید های AES را ارائه می کنند. هر کلید AES زمان کمتری نسبت به جدول MD5 می گیرد. بخاطر آورید که کد ۸۰۰/۰۰۰ کلید های AES (NUM_AES_KEYS) و ۱۰۰/۰۰۰ لیست MD5 (NUM_MD5_HASHES) را تولید می کند.

<u>لیست ۲-۵ : مثالی از تولید خروجی توسط تولید کننده کلید ها و لیست های MD5 که بصورت موازی اجرا می شوند</u>

```
AES KEY:    296ADFE332D8AED2423E8CB820624900D78E79495674F1CBBBDA411CB2E23DC5
MD5 HASH:   CEE3F2C283F460B668FC97A53368A9E5
MD5 HASH:   BFCDA3087E249C7DE1D305870AB399CC
MD5 HASH:   164E54003C1CBD3B4AFBF45F8FBAE35C
AES KEY:    A11B7FC47629C03E0A7F9B46A9C65857B4101E5E84744AAB261B58EED546556C
AES KEY:    30C4DC5168D4816F2D264EA9231C77C1EDC9BB022B673FC6F665A6B38A7316FC
AES KEY:    5898F745937BC5989515FF55E982DAA67FA445C4F60AF71BBE14B6641F8B64CB
AES KEY:    F5D925B92021E5F21A943AC5CCAD89E8E929FC1A7407B789A794DFC68D63E220
AES KEY:    DB6A3D03644E363A8950C69A32C724E623CB18B86BC9CB5D9D92F54891C85864
AES KEY:    41473B9A27124F4D9D50B8536F0C8E39ACF28F5A11E4DB06B80F9BBF0561549E
AES KEY:    26C2EB68720236CC15C2426378108BDAB0F6C29931930C864A692AE4D6BE62FC
AES KEY:    02FBD19C8509F9D4AF389D808FE40FEA2C1C89811C345A908EAA58AB4818A7B8
MD5 HASH:   5A041EF4132CCB7AA4EBBD6B92C80FB9
MD5 HASH:   DEB81F14E7F7DD55367C15D45359D984
MD5 HASH:   1FCD39BB3BD2C1115623D51BF2C917AD
```

[1] bottleneck

```
AES KEY:      EE42C8B44A828BB79F11FB54D9F7202315D6BACCAEC3B413755C0C2C798B1F67
AES KEY:      862423B344622829A9CA54FE8C8619FBAD11694C849753996CD3599973C0C36C
AES KEY:      5927AFF242E24241BA1CFD2364E6B96303D8366A0AE6BE1FC6E6511819F305DA
AES KEY:      D586E06A9623ADE1FD44827AE6AFAFC0274718A4EE4A98B3892879A0D7324A85
AES KEY:      CF00B0268B9E1B19E9E9C0B5A9F1E5B8795F24A0FDEB862FAC886F5B6C7A5A39
AES KEY:      4ED2B64C4F34A00F7880AAF10E3CFED0CCD02D7F1BCCF18A891062C94556C7E9
AES KEY:      751206B6E98D8C7832CBEB441A8EE48AC1F5C16500DA89C93D34104B83A76FC2
AES KEY:      5865588BE6E8832EF1C8056AD3B9926456778A52FA83E355FF66F4B6295742B0
MD5 HASH:     EDD39FC8108492B6BB4D9ABD806D4927
```

حالا، مجدد توضیح داده می شود که خط های که خروجی را به Console می فرستد در دو تابع GenerateAESKeys و GenerateMD5Hashes, وجود دارد.

۲-۳ حلقه های موازی

چه تابع GenerateAESKeys وچه تابع GenerateMD5Hashes از فرصت های برابری برای تکرار های موازی را اجرا کنند. آنها داده های ورودی را تولید می سازند و اجرای عملی یکسان را برای هر تکه در مثال ها ساده کند. این مثالی ساده از سناریوی موازی سازی داده است. با ایجاد حلقه هایی می توانید اجرای موازی را در اختیار داشته باشید. در این شیوه برخلاف اجرای تابع ها بصورت موازی، هر کدام می توانند بطور خودکار از مزیت های کامل موازی سازی و افزایش یا کاهش تعدا هسته های منطقی موجود بهرمند شوند.

در این مثال، عملیاتی مانند تولید لیست و کلید هست مکررا بکار برده می شود. بنابراین، Task های یکسانی را برای هر جزء می توانید بکار ببرید. انواع مشکلاتی که باعث نکرانی در موازی سازی است شناخته شده، زیرا Task ها به سادگی موازی سازی می شوند .

۲-۳-۱ Parallel.For

با جایگزینی حلقه های For با Parallel.For و تغییر پارامتر های برای تابع جدید از ویژگی های جدید موازی سازی بهرمند می شوید .

لیست های ۲-۵ و۲-٦ کد حلقه های قبلی را نشان می دهد و کد های جدید را با حلقه های refactore استفاده کند تا ترکیب نحوی ضروری را برای موازی سازی داده های ارائه ارائه شده بوسیله Parallel.For به انجام برساند. تابع جدید ParallelGenerateAESKeys و ParallelGenerateAESKeys، سعی در

استفاده کامل از تمامی هسته ها دارند، باستناد کار انجام شده توسط حلقه Parallel.For، این حلقه رفتارخود را براساس سخت افزار های موجود به صورت بلادرنگ بهینه سازی می کند .

لیست ۲-۵ : تابع اصلی GenerateAESKeys با حلقه For ترتیبی و نسخه موازی خودش .

ORIGINAL SEQUENTIAL FOR VERSION

```
private static void GenerateAESKeys()
{
        var sw = Stopwatch.StartNew();
        var aesM = new AesManaged();
        for (int i = 1; i <= NUM_AES_KEYS; i++)
        {
                aesM.GenerateKey();
                byte[] result = aesM.Key;
                string hexString = ConvertToHexString(result);
                // Console.WriteLine("AES KEY: {0} ", hexString);
        }
        Debug.WriteLine("AES: " + sw.Elapsed.ToString());
        }
```

PARALLELIZED VERSION USING PARALLEL.FOR

```
private static void ParallelGenerateAESKeys()
        {
                var sw = Stopwatch.StartNew();
                Parallel.For(1, NUM_AES_KEYS + 1, (int i) =>
                {
                        var aesM = new AesManaged();
                        byte[] result = aesM.Key;
                        string hexString = ConvertToHexString(result);
                        // Console.WriteLine("AES KEY: {0} ", hexString);
                });
                Debug.WriteLine("AES: " + sw.Elapsed.ToString());
        }
```

لیست ۲-٦ : تابع اصلی GenerateMD5Hashes با حلقه For ترتیبی و نسخه موازی اش.

ORIGINAL SEQUENTIAL FOR VERSION

```
private static void GenerateMD5Hashes()
        {
                var sw = Stopwatch.StartNew();
                var md5M = MD5.Create();
                for (int i = 1; i <= NUM_MD5_HASHES; i++)
                {
                        byte[] data =
                        Encoding.Unicode.GetBytes(
```

```
                Environment.UserName + i.ToString());
                byte[] result = md5M.ComputeHash(data);
                string hexString = ConvertToHexString(result);
                // Console.WriteLine("MD5 HASH: {0}", hexString);
            }
            Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
        }
```

PARALLELIZED VERSION USING PARALLEL.FOR

```
private static void ParallelGenerateMD5Hashes()
{
        var sw = Stopwatch.StartNew();
        Parallel.For(1, NUM_MD5_HASHES + 1, (int i) =>
        {
                var md5M = MD5.Create();
                byte[] data =
                Encoding.Unicode.GetBytes(
                Environment.UserName + i.ToString());
                byte[] result = md5M.ComputeHash(data);
                string hexString = ConvertToHexString(result);
                // Console.WriteLine("MD5 HASH: {0}", hexString);
        });
        Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
}
```

اکثر نسخه های اصلی تابع کلاس Parallel.For دارای پارامتر های زیر است :

- ✓ fromInclusive : این شماره ابتدایی برای محدود تکرار است. از نوع Int (Inet32) یا Long (Int64) می باشد .
- ✓ toExclusive : این عدد، کران بالای انحصاری را مشخص می کند. از نوع Int (Inet32) یا Long (Int64) می باشد و محدوده از عدد (fromInclusive) تا (۱ – toExclusive) می تواند باشد.

توجه به این پارامتر های بسیار مهم است زیر برخی حلقه های For محدوده تکراری با استفاده از یک کلان بالا توسط عملگر (=<) یا (<) تعریف می شود. بنابراین، هنگامی که حلقه For با استفاده از عملگرها به حلقه Parallel.For تبدیل می شود، اجباراً کران بالای اصلی به کران بالای اصلی منهای یک تبدیل می شود .

- ✓ Delegate : Body ی است که تعریف شده، بدون طرح اجرای از پیش تعریف شده یکبار بوسیله تکرار اجرا می شود. Body را از نوع Action<Int32> یا Action<Int64> بر طبق نوع استفاده شده در تعریف محدوده تکرار، می توان تعریف کرد .

Parallel.For از ممیز شناور برای پرش هایش پشتیبانی نمی کند، حتی استفاده از نوع در حلقه For نیز خطر زیادی دارد و دلیل آن افزایش های غیر دقیق در هر گِرد کردن می باشد. Parallel.For با مقدار های Int32 و Int64 کار می کند که بوسیله جمع کردن یک واحد در هر تکرار اجر می شود. دلیل استفاده از Body در Parallel، قسمت بندی محدوده تکرار بر اساس منابع سخت افزاری در دسترس می باشد. در اینجا ضمانتی برای ترتیب اجرای تکرارها وجود ندارد. برای مثال، در تکرای از ۱ الی ۱۰۱ (۱۰۰ تکرار)، شماره تکرار ۵۰ می تواند اجرایی قبل از شکاره تکرار ۲ داشته باشد که هر کدام موازی اجرا می شوند زیرا زمانی که هر تکرار اجرا می شود، متغیر و نامعلوم است. بنابراین روش های زیادی برای پیشبینی ترتیب اجرا وجود ندارد زیرا حلقه ها به تکرار های موازی زیادی تقسیم می شوند. کد ها را برای اجرای موازی آماده کرده و از تولید اثرات جانبی موازی سازی و همزمانی دوری گزینید.

Parallel.For نتیجه حلقه موازی را برگشت می دهد، زیرا مانند هر کد موازی دیگری، حلقه های موازی نسبت به کد های ترتیبی پیچیده ترند. در اینجا اجرایی متوالی نداریم، بنابراین نمی توان به متغیری دسترسی داشت که کارش تعیین پایان اجرای حلقه بعد از تعداد اجرای خاصی باشد. در واقع، تکه های بسیاری موازی اجرا می شوند.

۲-۳-۱-۱ خلاصه کردن یک حلقه ترتیبی

تابع اصلی GenerateAESKey با حلقه ایFor ی ترتیبی در لیست ۲-۵ نشان داده شده است. هنگامی که کدی ترتیبی را خلاصه می کنید تا نسخه ای موازی از آن ایجاد کنید، یک تمرین خوب ایجاد تابعی جدید با نامی متفاوت است. در این حالت، ParallelGenerateAESKeys تابعی جدید است. در زیر محدوده تکرار در حلقه های For اصلی تعریف شده است :

```
for (int i = 1; i <= NUM_AES_KEYS; i++)
```

تکه کدی از لیست ۲-۳

بدین منظور که NUM_AES_KEYS از یک تا خود NUM_AES_KEYS را تکرار می کند. ترجمه این تعریف به Parallel.For امری ضروری است همانطور که در ادامه مشاهده می کنید NUM_AES_KEYS با ۱ جمع می شود زیرا یک کران بالای انحصای می باشد .

```
Parallel.For(1, NUM_AES_KEYS + 1,
```

تکه کدی از لیست ۲-۷

اما پارامتر سوم Delegate است در این حالت حلقه از متغیر های تکراری استفاده نمی کند. همانگونه که در ادامه مشاهده می کنید، کد از عبارات lambda استفاده می کند تا تابعی با متغیری از نوع Int (Int32) پارامتر (i) را تعریف کند این عبارت برای نگهداری مقداری جاری بکار برده می شود.

```
Parallel.For(1, NUM_AES_KEYS + 1, (int i) =>
```

تکه کدی از لیست ۲-۵

کد قبل را به تنهای نیز می توان **اجرا** کرد ، شاید با اجرای موازی توابع دیگر .هر تکرار برای اجرای موازی با تکرار های در بدنه همان حلقه طراحی نشده بود. کاربرد Parallel.For در تغییر قوانین است. کد های که مشکلی را دارا اند نیاز به حل شدن دارند. تکرای های پی در پی با متغیر محلی aesM شریک اند.بدنه حلقه دارای کدی است که ارزش های آن برای هر تکرار تغییر می کند. برای مثال، کد زیر را در نظر بگیرید:

```
aesM.GenerateKey();
byte[] result = aesM.Key;
string hexString = ConvertToHexString(result);
```

تکه کدی از لیست ۲-۵

خط اول تابع GenerateKey را برای AesManaged که نمونه ای از aesM است فراخوانی می کند. کلید درProPerty ، aesM.Key ذخیره می شود. سپس کد با دلالت بر ارزش ذخیره شده در این Property متغیر را برگشت می دهد. سرانجام در خط آخر نتیجه به مبنای ١٦ تبدیل می شود و به رشته hexString داده می شود. واقعاً دشوار است که نتایج اجرای همزمان یا موازی این کد را تصور کنیم. زیرا شلوغ کاری زیادی دارد. برای مثال ، بخشی از کد کلید جدیدی تولید می کند که در propertyی ، aesM.Key ذخیره می شود تا در بخش دیگری که در حال اجرای موازی است خوانده شود. بنابراین، ارزش خوانده شده از propertyی ، aesM.Key معیوب می شود .

یک راه حل ممکن در ساختار های همگام سازی برای محافظت از هر مقدار و حالت در حال تغییر، مورد استفاده می باشد. در این صورت نیازی نیست که، کد های و همزمان سازی های بیشتری را جمع کنیم چرا که باعث سربار می شود. راه حل دیگری با کارایی بالاتری هست : خلاصه سازی بدنه حلقه ها و انتقال متغیر های محلی چنانچه متغیر های محلی درون متد موقتاً به عنوان Delegate کار می کنند. بدین ترتیب نیاز به ایجاد نمونه ای از AesManaged دارید. در این شیوه، بوسیله تمامی تکرار های موازی چیزی تقسیم نمی شود .

این تعویض (جابجایی) باعث اضافه شدن دستور العمل های بسیاری به روند اجرای هر تکرار می شود. اما اثرات جانبی[1] نامطلوب و کد های موازی بدون طراحی امنیت ایجاد شده را از بین می برد. در ادامه قسمت اصلی و جدیدی نشان داده شده ، با متغیرaesM کار جابجایی درونی انجام می شود (حروفی که پر رنگ تر اند) .

```
{
        byte[] result;
        string hexString;

        var aesM = new AesManaged();
        byte[] result = aesM.Key;
        string hexString = ConvertToHexString(result);

        // Console.WriteLine("AES KEY: {0} ", hexString);
});
```

تکه کدی از لیست ۲–۵

یک مسئله خیلی مشابه ای بر این اساس حل شده که به بدنه اصلی حلقه در GenerateMD5Hashes تبدیل شده است. لیست ۲–۶ تابع اصلی را با حلقه For ترتیبی نشان می دهد. در این حالت ParallelGenerateMD5Hashes تابعی جدیدی می باشد. و نیاز به استفاده از تکنیکهای کوتاه و خلاصه گفته شده دارد. به این دلیل که، اگر نمونه MD5 مشکلی را در حالت داخلی تولید کند، شما آن را نمی شناسید. این شیوه ایمنی بیشتر نسبت به ایحاد نمونه مستقل جدید برای هر تکرار دارد. خطوط زیر بدنه

---

[1] side effects

اصلی جدیدی را نشان می دهند، متغیر md5M برای انتقال درونی Delegate استفاده می شود (حروفی که پر رنگ تر اند) .

```
{
        var md5M = MD5.Create();
        byte[] data = Encoding.Unicode.GetBytes(Environment.UserName + i.ToString());
        byte[] result = md5M.ComputeHash(data);
        string hexString = ConvertToHexString(result);
        // Console.WriteLine("MD5 HASH: {0}", hexString);
});
```

تکه کدی از لیست ۲-۵

۲-۳-۱-۲ اندازه گیری مقیاس

تابع اصلی را با نسخه جدید زیر جایگزین کنید، ابتدا تابع ParallelGenerateAESKeys شروع می شود و سپس تابع ParallelGenerateMD5Hashes کار خود را آغاز می کند .

```
static void Main(string[] args)
{
var sw = Stopwatch.StartNew();
ParallelGenerateAESKeys();
ParallelGenerateMD5Hashes();
Debug.WriteLine(sw.Elapsed.ToString());
// Display the results and wait for the user to press a key
Console.WriteLine("Finished!");
Console.ReadLine();
}
```

تکه کدی از لیست ۲-۵

حالا، تابع ParallelGenerateAESKeys و ParallelGenerateMD5Hashes برای اجرا شدن نیاز زمان تقریبی ۷.۵ ثانیه دارند. چرا که هر کدام کاملا از مزیت های دو هسته ارائه شده توسط هسته بهرمند می شوند. بنابراین SpeedUP بدست آمده ۱.۸۷ x = (۷.۵ / ۱٤) بیشتر از کد ترتیبی می باشد. که نسبت به SpeedUP بدست آمده توسط Parallel.Invoke (۱.٥٦x) بهتره است زیرا این حلقه ها با تکه های موازی اجرا می شوند .حلقه های موازی سعی در متعادل نگه داشتن بار کاری هر هسته دارند . ParallelGenerateAESKeys در ٤.۲ ثانیه و ParallelGenerateMD5Hashes در ۳.۳ ثانیه اجرا می شود.

مزیت های دیگری موازی سازی با استفاده از Parallel.For : زمانی که بیش دو هسته استفاده می کنید، کد یکسانی می تواند مقایسه شود. نسخه ترتیبی این برنامه روی کامپیوتری با ریزپردازنده quad-core برای اجرا شدن نیاز به تقریباً ۱۱ ثانیه زمان دارد. زمان اجرای کد ترتیبی را باید مجدد اندازه گیری کرد زیرا هر پیکره بندی سخت افزاری ای نتایج مختلفی برای کد های ترتیبی و موازی دارد. به همین خاطر SpeedUp دستیافته را اندازه گیری کنید همیشه نیاز به محاسبه خط مبنایی در پیکره بندی سخت افزاری دارید. نسخه بهینه شده با استفاده از Parallel.For نوشته شده و نیاز به ٤.١ ثانیه برای اجرا دارد. هر تابع بطور کامل از مزیت های که توسط چهار هسته ارائه شده است استفاده می کند. بنابراین SpeedUP بدست آمده ۲.٦۸ x = (٤.١ / ۱۱) بیشتر از کد ترتیبی می باشد. ParallelGenerateAESKeys ۲.۱۲ ثانیه و arallelGenerateMD5Hashes ۱.۹۸ ثانیه برای اجرا نیاز دارند. این حلقه ها در زمان نسبتا مناسبی کار را به انجام می رسانند. و بنابراین ایستگاه کاری پیش فرض جمع آوری پس مانده ها[1] (GC) ممکن است باعث کاهش میزان مقایسه شود. در فصل ۱۰ چگونگی فعال سازی سرویس GC را در هنگامی که نیاز به تخصیص دهنده های بیشتری دارید را فرا می گیرید .

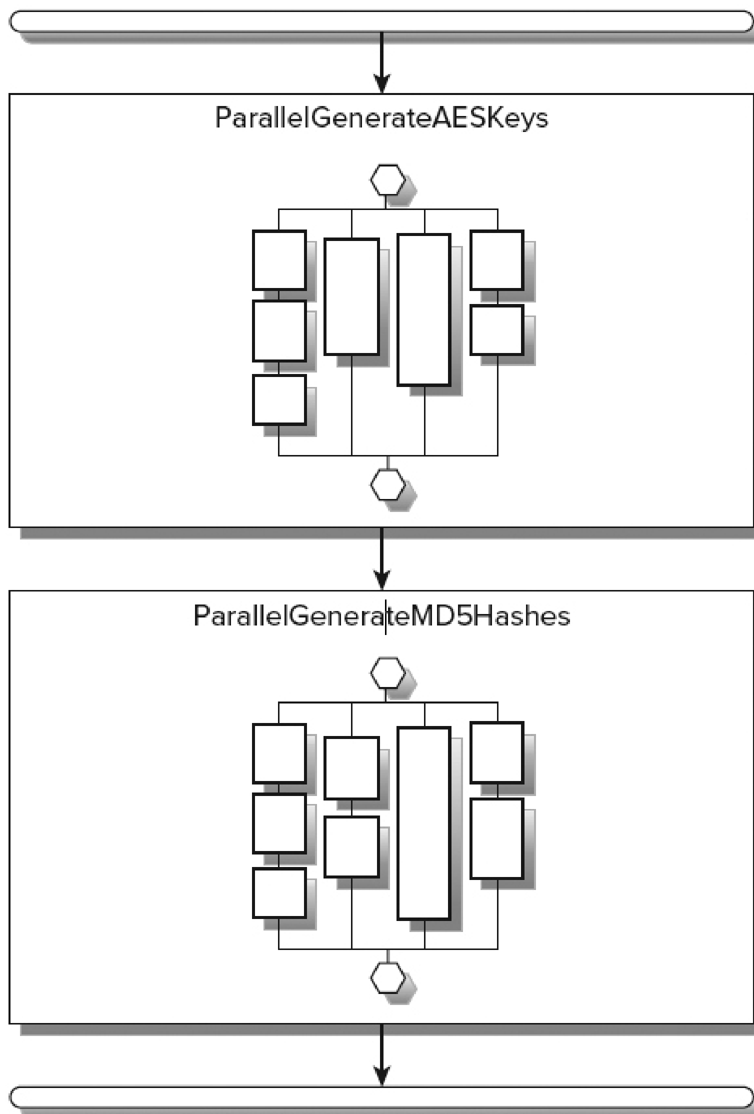
کد های موازی توانایی مقایسه تعداد هسته افزایش یافته را دارند که در نسخه Parallel.Invok این امکان در دسترس نیست. یعنی کد های موازی SpeedUP ای خطی را ارائه می دهند . در اکثر موارد محدودیتی های مقایسه پذیری وجود دارد. زمانی که کد های موازی به تعداد خاصی از هسته دسترسی می یابند. الگریتم های موازی به موفقیتی با SpeedUP بالاتر نمی توانند دست بیابند.

در مثال قبلی، نیاز بود تا بدنه حلقه ها برای استفاده در هر تکرار تغییر یابد. بنابراین، سرباری به هر تکرار اضافه می شود که جزئی از ترار ترتیبی نیست و فراخوانی Delegate بیشتر از فراخوانی مستقیم تابع گران تمام می شود. همچنین، Parallel.For و کار های اساسی باعث می شود در اجرای تکه های مختلف با تکرار های موازی به کار توضیع و هماهنگ کردن سربار زیادی اضافه شود. به همین دلیل SpeedUP به ٤x نزدیک نمی شود اما در اجرا با چهار هسته تقریباً برابر ۲.٦۸x است .در غالب موارد الگریتم های موازی

[1] garbage collection

SpeedUP خطی ای را ارائه نمی دهند. همچنین گذرگاه[1] های مرتبط با اجرای سریال[2] و معماری های سخت افزاری، واقعاً اختلافی بیش از حدی را در برتری تعداد هسته ایجاد می کند اگر مزیت عملکرد بار سنگینی به سربار حاصل از کد موازی اضافه کند، باید اندازه SpeedUP را تعیین کنید.

شکل ۲-۹ یکی از جریان های اجرای ممکن را نشان می دهد که از مزیت وجود چهار هسته بهرمند می شود.



شکل ۲-۹

[1] bottleneck
[2] serial execution

۲-۳-۱-۳ کار با مشکلات موازی نگران کننده

این نگران کننده هست که همه مشکلات موازی نیستند. تعداد زیادی از الگریتم های موازی به طور نگران کننده ای وجود دارند چنانچه در مثال قبل شرح داده شد. ساده ترین راه برای فهمیدن مهمترین ویژگی هایی که توسط TPL ارائه شده، کار با مشکلات موازی نگران کننده است. معمولاً، مشکلات موازی نگران کننده نیازی به مکانیسم های هماهنگی پیچیده ندارند. در برنامه های پیچیده موازی نیاز به اضافه کردن کد های برای هماهنگی ارتباط بین Task های موازی است. در قسمت بعد مهمترین ساختار های TPL توضیح داده می شود و فصل ۵ ساختار های هماهنگی که در مهمترین ساختار های TPL بکار رفته را پوشش می دهد .

۲-۳-۲ Parallel.Foreach

برخی اوقات خلاصه کردن حلقه For موجود به نحوی که در قبل شرح داده شد کاری پیچیده است و تغییراتی که در کد ایجاد می کند باعث می شود در هر تکرار (درون حلقه) سربار زیادی ایجاد شود که عملکرد کلی را کاهش می دهد. یکی از تناوب های مفید این است که تمام داده ها را در داخل قسمت های کوچکتر بخشبندی شوند تا حلقه های ترتیبی مستقل به صورت موازی اجرا شوند. با استفاده از Parallel.ForEach کار بخشبندی را می توانید بصورت خصوصی انجام داد، بدین ترتیب ایجاد نسخه جدید از حلقه های ترتیبی را با خلاصه سازی ساده تر پردازش می کند .

Parallel.ForEach مکانیسم کلی ای را برای پردازش مجموعه ای از داده های فراهم می کند. شما از محدوده عدد صحیح برای مجموعه ای از داده ها استفاده کنید و تقسیم کننده ای خصوصی را برای تبدیل این محدود به تکه های کوچکی بکار ببرید. هر تکه ای که در حلقه خودش است، بصورت موازی پردازش شده است. لیست ۲-۷ حاوی کد های جدید است که از حلقه های خلاصه شده ای سود می برند که در آن فقط از قسمت های ضروری شامل ابزار موازی سازی ارائه شده توسط Parallel.ForEach نوشته شده اند. این تابع با حلقه For ترتیبی و قسمت کننده خصوصی از فضای نام System.Collections.Concurrent. Partitioner ایجاد شده است. تابع عای جدید ParallelPartitionGenerateAESKeys و ParallelPartitionGenerateMD5Hashes ، همچنین سعی در استفاده مفید از تمامی هسته ها دارند و روی انجام شده کار های توسط Parallel.ForEach اعتماد کرده اند. و در محدوده انجام بخشبندی کار توزیع

حلقه های ترتیبی کوچکتر را در حلقه های موازی بسیاری با استفاده از هسته های در دسترسی انجام می دهند. همچنین کد براساس سخت افزار موجود در زمان اجرا رفتارش را بهینه سازی می کند .

لیست ۲-۷ : نسخه موازی دیگری از حلقه ترتیبی اصلی با استفاد از Parallel.ForEach با یک تقسیم کننده خصوصی

```
private static void ParallelPartitionGenerateAESKeys()
{
        var sw = Stopwatch.StartNew();
        Parallel.ForEach(Partitioner.Create(1, NUM_AES_KEYS + 1), range =>
        {
                var aesM = new AesManaged();
                Debug.WriteLine(
                        "AES Range ({0}, {1}. TimeOfDay before inner loop starts: {2})",
                        range.Item1, range.Item2,
                        DateTime.Now.TimeOfDay);
                for (int i = range.Item1; i < range.Item2; i++)
                {
                        aesM.GenerateKey();
                        byte[] result = aesM.Key;
                        string hexString = ConvertToHexString(result);
                        // Console.WriteLine("AES KEY: {0} ", hexString);
                }
        });
        Debug.WriteLine("AES: " + sw.Elapsed.ToString());
}

private static void ParallelPartitionGenerateMD5Hashes()
{
        var sw = Stopwatch.StartNew();
        Parallel.ForEach(Partitioner.Create(1, NUM_MD5_HASHES + 1),
        range =>
        {
        var md5M = MD5.Create();
        Debug.WriteLine(
                "MD5 Range ({0}, {1}. TimeOfDay before inner loop starts: {2})",
                range.Item1, range.Item2,
                DateTime.Now.TimeOfDay);
        for (int i = range.Item1; i < range.Item2; i++)
                {
                        byte[] data =
                        Encoding.Unicode.GetBytes(
                        Environment.UserName + i.ToString());
```

```
                byte[] result = md5M.ComputeHash(data);
                string hexString = ConvertToHexString(result);
                // Console.WriteLine("MD5 HASH: {0}", hexString);
            }
        }
        Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
}
```

در کد از فضای نام مهم دیگری نیز برای TPL می توان استفاده کرد ، System.Collections.Concurrent ، که شما را برای دسترسی مفیدی از مجموعه های محیا شده برای همزمانی و تقسیم کننده های خصوصی که در NET Framework 4. معرفی شده است قادر می سازد. همانطوری که در مثال زیر می بینید می توانید این فضای را بکار ببرید.

```
using System.Collections.Concurrent;
```

تکه کدی از لیست ۲-۷

تابع کلاس Parallel.ForEach دارای ۲۰ حالت مختلف برای فراخوانیست. تعریفی که در لیست ۲-۷ استفاده شده دارای پارامتر های زیر است :

- ✓ source : تقسیم کننده ای است که منابع داده ای را برای تکه تکه کردن در تعدادی زیادی فراهم می کند .
- ✓ Body : Delegate ایست که فعال شده، یکبار در تکرار و بدون تعریف طرح اجرایی از قبل.به هر بخش تعریف شده به عنوان پارامتر دسترسی دارد، در این حالت Tuple<int, int> می باشد .

بعلاوه ، Parallel.ForEach مقدار ParallelLoopResult را برگشت می دهد .

۲-۳-۲-۱ کار با بخش ها در حلقه ای موازی

در لیست ۲-۵ تابع اصلی GenerateAESKey با حلقه For ی ترتیبی نمایش داده شد. خطوط پررنگ شده از کد در لیست ۲-۷ حلقه For ی ترتیبی را ارئه می دهند. تنها خطی که در تعریف For تغییر کرده به حساب حد پایین گذاشته می شود و کرا بالای بخش بوسیله range.Item1 و range.Item2 به شرح ذیل نسب داده می شود :

```
for (int i = range.Item1; i < range.Item2; i++)
```

تکه کدی از لیست ۲-۷

در این حالت، خلاصه سازی کد های ترتیبی ساده تر است زیرا نیازی به جابجای متغیر های محلی نیست. تنها اختلاف این است که بجای کار با منابع داده ای کامل، Parallel.ForEach داده ها را به تعداد زیادی همزمانی بخش های مستقل تکه تکه می کند. هر بخش با یک حلقه داخلی پشت سر هم کار می کند. کد زیر بخش ها را به عنوان پارامتر Parallel.ForEach تعریف می کند :

```
Partitioner.Create(1, NUM_AES_KEYS + 1)
```

تکه کدی از لیست ۲-۷

این خط محدوده ای از خود یک تا NUM_AES_KEYS + ۱ را تکه تکه می کند و این کار را در بخش های زیادی با یک کران بالا و یک کران پایین با ایجاد یک (Tuple<Int32, Int32>) Tuple<int, int> ، انجام می دهد که تعیین کننده تعداد بخش ها برای ایجاد نمی باشد. بنابراین، از Partitioner.Create برای ایجاد با مقادیر پیش فرض استفاده کنید. ParallelPartitionGenerateAESKeys شامل کران بالا و پایین هر بخش تولید می باشد و زمان واقعی (DataTime.Now.TimeOfDay) که حلقه ترتیبی در این محدوده اجرایش شروع می شود ، به طریقه زیر نوشته می شود :

```
Debug.WriteLine("AES Range ({0}, {1}. TimeOfDay before inner loop starts: {2})",
range.Item1, range.Item2,
DateTime.Now.TimeOfDay);
```

تکه کدی از لیست ۲-۷

جایگزینی تابع اصلی با نسخه جدید زیر، ابتدا آغاز ParallelPartitionGenerateAESKeys و سپس ParallelParallelGenerateMD5Hashes :

```
static void Main(string[] args)
{
        var sw = Stopwatch.StartNew();

        ParallelPartitionGenerateAESKeys();
        ParallelPartitionGenerateMD5Hashes();

        Debug.WriteLine(sw.Elapsed.ToString());
        // Display the results and wait for the user to press a key
        Console.WriteLine("Finished!");
        Console.ReadLine();
}
```

تکه کدی از لیست ۲-۷

چنانچه در لیست ۲-۸ نشان داده شده است، تقسیم کننده محدوده را به ۱۳ ناحیه در یک ماشین خاص تقسیم می کند. تعداد نواحی پیش فرض ایجادی بستگی به تعداد هسته های منطقی، اندازه هر ناحیه و فاکتور های دیگر دارد. بنابراین Parallel.ForEach، ۱۳ حلقه تکرار پیشت سر هم را در محدوه های خودشان اجرا می کند. این حلقه ها در زمان یکسانی شروع نمی شوند، چرا که ایده خوبی بود که تنها با چهار هسته در دسترس بایستی کد را به ۱۳ قسمت پخش شود. سعی حلقه های موازی در این است که بار کاری حاصل را در حالت متعادلی نگه دارند و این کار را با در نظر گرفتن منابع سخت افزاری در دسترس به انجام برسانند. در خط برجسته شد پیچیدگی ای که بوسیله موازی سازی ایجاد می شود نشان داده می شود. زمان قبل از ورود به حلقه حساب می شود اولین بخشی که در حلقه داخلی پشت سر به آن رسیده می شود (66667, 133333) هست نه (1, 66667). بخاطر بیاورید که مقدار کران بالای در لیست ۲-۸ را شامل می شود، زیرا حلقه For درونی از عملگر کوچکتری استفاده می کند (<).

بعلاوه، ترتیبی که داده ها در خروحی خطایابی نوشته می شوند از ترتیبی که توسط تابع WriteLine فراخوانی شد فرق دارد. تغییری در ترتیب رخ داده زیرا تعداد بسیاری از فراخوانی های تابع WriteLine بصورت همزمان است. در واقع ، زمانی که SpeedUP اندازه گیری می شود ، کامنت کردن (نوشتن قبل از دستوری ، تا کامپایلر آن خط را نادیده بگیرد) خط قبل از شروع حلقه خیلی مهم است. نوشتن اطلاعات در خروجی خطا یابی حادثه ای را در زمان کل بوجود می آورد این حادثه تنگایی در تولید تابع است.

لیست ۲-۸ : خروجی خطا یابی برای ParallelPartitionGenerateAESKeys در سیستمی با یک پردازشگر quad-core

AES Range (133333, 199999. TimeOfDay before inner loop starts: 15:45:38.2205775)
AES Range (66667, 133333. TimeOfDay before inner loop starts: 15:45:38.2049775)
AES Range (266665, 333331. TimeOfDay before inner loop starts: 15:45:38.2361775)
AES Range (199999, 266665. TimeOfDay before inner loop starts: 15:45:38.2205775)
AES Range (1, 66667. TimeOfDay before inner loop starts: 15:45:38.2205775)
AES Range (333331, 399997. TimeOfDay before inner loop starts: 15:45:39.0317789)
AES Range (399997, 466663. TimeOfDay before inner loop starts: 15:45:39.0317789)
AES Range (466663, 533329. TimeOfDay before inner loop starts: 15:45:39.1097790)
AES Range (533329, 599995. TimeOfDay before inner loop starts: 15:45:39.2345793)
AES Range (599995, 666661. TimeOfDay before inner loop starts: 15:45:39.3281794)
AES Range (666661, 733327. TimeOfDay before inner loop starts: 15:45:39.9365805)
AES Range (733327, 799993. TimeOfDay before inner loop starts: 15:45:40.0145806)

AES Range (799993, 800001. TimeOfDay before inner loop starts: 15:45:40.1705809)

نسخه جدید Parallel.ForEach با بخش های خصوصی تقریباً به زمان یکسانی نسبت به نسخه Parallel.For قبلی برای اجرا شدن احتیاج دارد .

۲-۳-۲-۲ بهینه سازی بخش ها با استفاده از تعداد هسته ها

شما می توانید بخش های تولید شده را بر اساس تطبیق شان با تعداد هسته های منطقی در زمان اجرا به ترتیب به چرخش در (حلقه در) آورید. پیش فرض ساخته شده سعی دارد تا مقدار سطح بار کاری را در حالت تعادل نگه دارد. System.Environment.ProcessorCount تعداد هسته های منطقی یا پردازشگر های که توسط سیستم عامل تشخیص داده می شود را فراهم می کند. اگر حجم بار کاری را در حالت متعادل بشناسید، می توانید با استفاده از این مقدار محدوده مطلوب هر بخش را محاسبه کنید. از سه پارامتر برای فراخوانی Partitioner.Create به صورت زیر می توانید استفاده کنید:

((numberOfElements / numberOfLogicalCores) + 1) : int or long

از کد زیر برای ایجاد بخش های با ParallelPartitionGenerateAESKeys می توانید استفاده کنید :

```
Partitioner.Create(
        1,
        NUM_AES_KEYS,
        ((int)(NUM_AES_KEYS / Environment.ProcessorCount) + 1))
```

تکه کدی از لیست ۲-۹

از خطوط بسیار مشابه ای برای بهبود ParallelPartitionGenerateMD5Hashes می تواند بهره ببرید :

```
Partitioner.Create(
        1,
        NUM_MD5_HASHES,
        ((int)( NUM_MD5_HASHES / Environment.ProcessorCount) + 1))
```

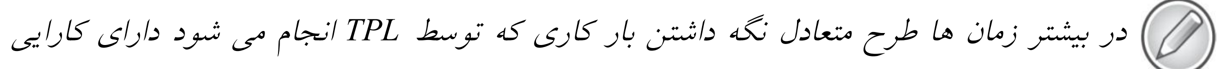
تکه کدی از لیست ۲-۹

بطوریکه در لیست ۲-۹ نشان داده شده است، تقسیم کننده چهار بخش را ایجاد می کند، زیرا اندازه مطلوب محدوده ۲۰۰۰۰۱ = (۱ + (۸۰۰۰۰۰/٤) (int)) هست. بنابراین، Parallel.ForEach چهار کد ترتیبی درون حلقه های For را با استفاده از محدوده هایشان بر روی تعداد هسته های منطقی در دسترس اجرا می کند .

لیست ۲-۹ : عیب یابی خروجی تولید شده در زمان اجرای نسخه بهینه سازی شده ParallelPartitionGenerateAESKeys با یک پردازشگر quad-core .

Range (1, 200002. TimeOfDay before inner loop starts: 16:32:51.3754528)
Range (600004, 800000. TimeOfDay before inner loop starts: 16:32:51.3754528)
Range (400003, 600004. TimeOfDay before inner loop starts: 16:32:51.3754528)
Range (200002, 400003. TimeOfDay before inner loop starts: 16:32:51.3754528)

حالا، دو تابه ParallelPartitionGenerateAESKeys و ParallelPartitionGenerateMD5Hashes تقریباً نیاز به ۳.٤۰ ثانیه زمان برای اجرا دارند. زیرا هر کدارم بخش های زیادی را در هسته های در دسترس تولید می کنند و از حلقه های ترتیبی در هر Delegate استفاده می کنند. بنابراین مقدار SpeedUP بدست آمده، ۲.۳۳x = (۳.٤ / ۱۱) بیشتر از کد ترتیبی است و سربار را کاهش می دهد و زمان را از ٤.۱ به ۳.٤ کاهش می دهد .



در بیشتر زمان ها طرح متعادل نگه داشتن بار کاری که توسط TPL انجام می شود دارای کارایی بالایی است. به هر حال شما طرح ها، کد ها و الگریتم هایتان را بهتر از TPL می شناسید. بنابراین با مد نظر قرار دادن توانایی هایی که در معماری های سخت افزار های مدرن ارائه شده می توانید کار تحلیل بار کاری الگریتم تان را بهتر از TPL می توانید تحلیل کنید. سپس از این اطلاعات می توانید برای استفاده در ویژگی های بسیاری که در TPL قرار داده شده، مصرف کنید تا عملگر کل را بالا ببرید. سربار غیرضروری معرفی شده توسط اولین زمان موازی سازی را کاهش دهید .

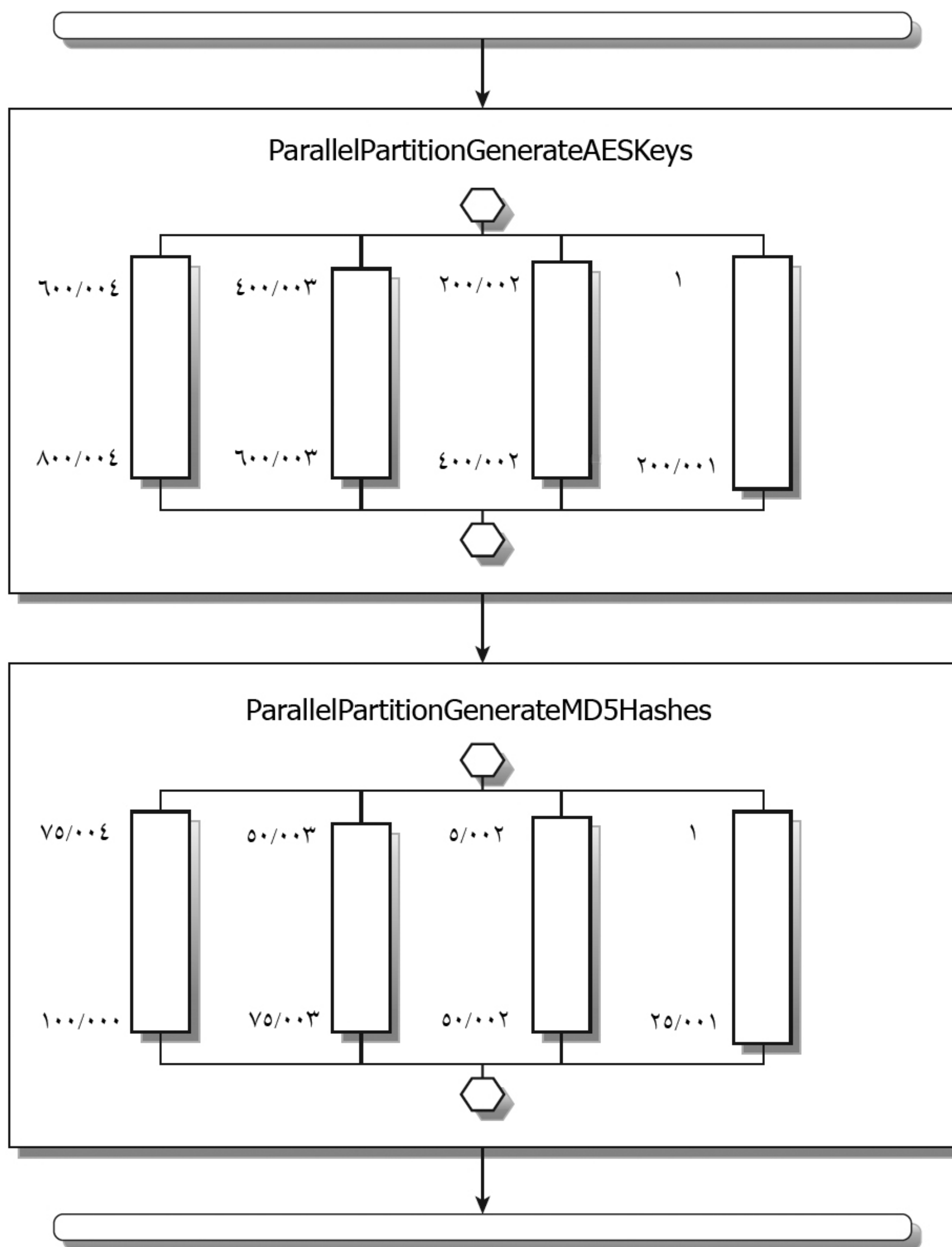
یک روند اجرای ممکن در شکل ۲-۱۰ نشان داده شده است و از چهار هسته با طرح قسمت بندی بهینه[1] سود می برد .

لیست ۲-۱۰ نتایج کار تقسیم کننده را در ایجاد هشت محدوده با هشت پردازشگر منطقی نشان می دهد. زیرا اندازه ناحیه های طراحی شده برابر است با ۱۰۰۰۰۱= (۱+ (۸ / ۸۰۰۰۰۰) (int)) . در این حالت Parallel.ForEach هشت کد ترتیبی را پشت سر هم در حلقه For اجرا می کند .

لیست ۲-۱۰: تولید خروجی رفع عیب شده هنگامی که نگارش بخش های بهینه ParallelPartitionGenerateAESKeys در پردازنده ای با هشت هسته منطقی اجرا می شود .

[1] optimized partitioning

AES Range (1, 100002. TimeOfDay before inner loop starts: 00:07:21.4111496)
AES Range (500006, 600007. TimeOfDay before inner loop starts: 00:07:21.4601524)
AES Range (100002, 200003. TimeOfDay before inner loop starts: 00:07:21.4111496)
AES Range (200003, 300004. TimeOfDay before inner loop starts: 00:07:21.4111496)
AES Range (400005, 500006. TimeOfDay before inner loop starts: 00:07:21.4381511)
AES Range (300004, 400005. TimeOfDay before inner loop starts: 00:07:21.4361510)
AES Range (700008, 800000. TimeOfDay before inner loop starts: 00:07:21.5181557)
AES Range (600007, 700008. TimeOfDay before inner loop starts: 00:07:21.5181557)



کران بالا مقادیر موجود در نمودار می باشد

شکل ۲-۱۰

۲-۳-۲-۳ کار با منابع داده IEnumerable

Parallel.ForEach برای خلاصه سازی حلقه های ForEach موجود و تکرار یک مجموعه از IEnumerable که دارای امنیت کمی است استفاده می شود .

لیست ۲-۱۱ کدی برای تابع جدید GenerateMD5InputData نشان می دهد که ترتیب اعداد Int در آن از یک تا خود NUM_AES_KEYS می باشد. پارامتر دوم Enumerable.Range شماره عدد integer های پشت سر هم است که یک کران بالا را تولید می کند. بجای استفاده از حلقه اعدد را برای تکرار ها کنترل کنید، کد تابع ParallelForEachGenerateMD5Hashes ، ترتیب را در متغیر محلی inputData ذخیر می کند.

<u>لیست ۲-۱۱ : نسخه موازی از تابع GenerateMD5Hashes با استفاده از Parallel.ForEach با منبع داده ای IEnumerable</u>

```
private static IEnumerable<int> GenerateMD5InputData()
{
	// Generate a sequence of NUM_AES_KEYS integral numbers
	// The value of the first integer in the sequence (start) is 1
	// The number of sequential integers to generate (count)
	// is NUM_AES_KEYS
	return Enumerable.Range(1, NUM_AES_KEYS);
}
private static void ParallelForEachGenerateMD5Hashes()
{
	var sw = Stopwatch.StartNew();
	var inputData = GenerateMD5InputData();
	Parallel.ForEach(inputData, (int number) =>
	{
		var md5M = MD5.Create();
		byte[] data =
			Encoding.Unicode.GetBytes(
			Environment.UserName + number.ToString());
		byte[] result = md5M.ComputeHash(data);
		string hexString = ConvertToHexString(result);
		// Console.WriteLine("MD5 HASH: {0}", hexString);
	});
	Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
}
```

ساده ترین تعریف تابع Parallel.ForEach در لیست ۲-۱۱ استفاده شده تا نسخه جدیدی از تابع تولید لیست MD5 تولید کند، ParallelForEachGenerateMD5Hashes دارای پارامتر های زیر است :

- ✓ Source : این مجموعه نماینده یک واسط IEnumerable می باشد و منابع داده را فراهم می سازد .
- ✓ Delegate : Bodyی است که برای یک طرح از پیش تعریف شده فراخوانی شده و هر جزئیاتی از نوع int را از مجموعه Source دریافت می کند.

خط زیر Parallel.ForEach را با منبع داده (inputData) فراخوانی می کند و تابع Delegate عددی را برای هر تکرار دریافت می کند :

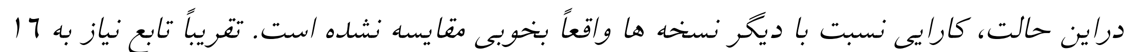
```
Parallel.ForEach(inputData, (int number) =>
```

تکه کدی از لیست ۲-۱۱

خط زیر داده های ورودی را برای تابع محاسبه پراکنده(درهم) و استفاده از ارزش موجود در عدد :

```
data = Encoding.Unicode.GetBytes(Environment.UserName + number.ToString());
```

تکه کدی از لیست ۲-۱۱



دراین حالت، کارایی نسبت با دیگر نسخه ها واقعاً بخوبی مقایسه نشده است. تقریباً تابع نیاز به ۱۶ ثانیه برای اجرا شدن نیاز دارد این تابع در پیکره بندی سخت افزاری یکسان با مثال قبل، اجرا شده است. در اینجا ابزار ها بهینه نیستند چون کد مجبور به تکرار ۱۰۰/۰۰۰ آیتم ترتیبی می باشد. اجرا موازی انجام می شود، اما زمان بیشتر نسبت به اجرای حلقه ها با سرباری اندک را می گیرد. همچنین حافظه زیادی نیز مصرف می کند. این مثال بهترین تمرین برای این حالت نمی باشد اما به فهم فرصت های ارائه شده در تابع های کلاس Parallel و ارزیابی آنها در سناریو های مختلف کمک می کند .

قسمتی از مشکلات تکرار واقعی بصورت Enumerable ترتیبی اتفاق می افتند. تمامی نخ هایی که از اجرای Task ها پشتیبانی می کنند توسط Parallel.ForEach شروع شده اند و مجبور اند تا زمانی که تکرار کننده ها دسترسی دارند به طور سریالی اجرا شوند. می توان یک تقسیم کننده خصوصی ایجاد کرد که توانایی قسمت بندی محدوده های پویا را دارند.

۲-۳-۳ خروج از حلقه های موازی

اگر در حلقه ای ترتیبی می خواهید یک وقفه بزنید ، می توانید از دستور Breack استفاده کنید .زمانی که با حلقه های موازی کار می کنید، خروج از بدنه تابع Delegate تاثیری بر اجرای حلقه موازی ندارد زیرا بدنه تابع Delegate از ساختار حلقه های سنتی جدا بوده و در هر فراخوانی تکرار جدید فراخوانی می شود .

لیست ۲-۱۲ نسخه جدیدی از تابع ParallelForEachGenerateMD5Hashes و فراخوانی ParallelForEachGenerateMD5HashesBreak. را نشان می دهد .

لیست ۲-۱۲ : نسخه جدیدی از تابع ParallelForEachGenerateMD5Hashes با احتمال خروج از حلقه

```
private static void DisplayParallelLoopResult(ParallelLoopResult loopResult)
{
        string text;
        if (loopResult.IsCompleted)
        {
                text = "The loop ran to completion.";
        }
        else
        {
                if (loopResult.LowestBreakIteration.HasValue)
                {
                        text =   "The loop ended by calling the Break statement.";
                }
                else
                {
                        text =  "The loop ended prematurely with a Stop statement.";
                }
        }
        Console.WriteLine(text);
}
private static void ParallelForEachGenerateMD5HashesBreak()
{
        var sw = Stopwatch.StartNew();
        var inputData = GenerateMD5InputData();
        var loopResult = Parallel.ForEach(inputData,
        (int number, ParallelLoopState loopState) =>
        {
                // There is very little value in doing this first thing
                // in the loop, since the loop itself is checking
                // the same condition prior to invoking the body delegate
                // Therefore, the following if statement appears commented
```

```
                //if (loopState.ShouldExitCurrentIteration)
                //{
                // return;
                //}
                var md5M = MD5.Create();
                byte[] data =
                        Encoding.Unicode.GetBytes(
                        Environment.UserName + number.ToString());
                byte[] result = md5M.ComputeHash(data);
                string hexString = ConvertToHexString(result);
                // Console.WriteLine("MD5 HASH: {0}", hexString);
                if (sw.Elapsed.Seconds > 3)
                {
                        loopState.Break();
                        return;
                }
        });
        DisplayParallelLoopResult(loopResult);
        Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
}
```

حالا، متغیر محلی loopResult حاصل فراخوانی تابع Parallel.ForEach را ذخیره می کند. بعلاوه، بدنه تابع Delegate نمونه ای از ParallelLoopState را به عنوان پارامتر های ثانیه دریافت می کند .

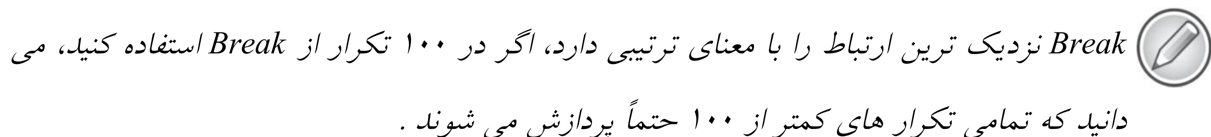
```
var loopResult = Parallel.ForEach(inputData,
(int number, ParallelLoopState loopState) =>
```

تکه کدی از لیست ۲-۱۲

۲-۳-۳-۱ درک ParallelLoopState

loopState نمونه ای از ParallelLoopState را ارائه می کند که دو تابع اجرای Parallel.For یا Parallel.ForEach را متوقف می کند :

- ✓ Break : این ارتباط های که حلقه های موازی مسئول قطع آن اند اجرای تکرار جاری است و این کار را به محض ممکن بودنش به انجام می رساند. اگر در ۱۰۰ تکرار پردازش، زمانی که Break فراخوان می شود، حلقه تمام تکرار های کمتر از ۱۰۰ را پردازش می کند .
- ✓ Stop : این ارتباط های که حلقه های موازی مسئول قطع اجرای به محض ممکن بود آن هستند. اگر ۱۰۰ تکرا آغازشود، هنگام فراخوانی Stop حلقه پردازش مابقی تکرار ها تا ۱۰۰ را ضمانت نمی کند.


Break نزدیک ترین ارتباط را با معنای ترتیبی دارد، اگر در ۱۰۰ تکرار از Break استفاده کنید، می دانید که تمامی تکرار های کمتر از ۱۰۰ حتماً پردازش می شوند .

کد نشان داده شده در لیست ۲-۱۲ تابع Break را فراخوانی می کند، اگر اگر زمان باقی مانده بیشتر از ۳ ثانیه باشد :

```
if (sw.Elapsed.Seconds > 3)
{
        loopState.Break();
        return;
}
```

تکه کدی از لیست ۲-۱۲

کد در بدنه Delegate، Lambda به متغیر sw که در ParallelForEachGenerateMD5HashesBreak تعریف شده دسترسی دارد و ارزش دوم این Property فقط خواندنی هست. هنگامی که دستوری برای توقف اجرای حلقه موازی باشد، به ترتیبی که بخواهید می تواند مقدار قط خواندنی ShouldExitCurrentIteration را چک کنید. این درخواست می تواند توسط تکرار های جاری یا همزمان داده شود. لیست ۲-۱۲ شامل خطوط کامنت شده[1] زیر است که در آن ShouldExitConcurrentIteration مقدارش True می باشد :

```
if (loopState.ShouldExitCurrentIteration)
        {
                return;
        }
```

تکه کدی از لیست ۲-۱۲

اگر شرط ذکر شده True باشد تابع پایان می یابد و از اجرای تکرار های نا مناسب اجتناب می شود. خطوط کامنت شده اند زیرا در این حالت، تکرار های جمع شده مشکلی ندارند. بنابراین، نیازی به جمع کردن دستورالعمل های هر تکرار نداریم. اگر بخواهید حلقه هایتان زودتر پایان بیابند و دیگر حقه ها فوراً خاتمه یابند، فقط نیاز به اجرای آن در میانه اجرای طولانی تکرار ها دارید .

[1] کامنت گذاری خطوط برای کامپایل نشدنشان است . البته برخی کاراکترهای خاص در کامنت ها باعث ایجاد تغییر در روند کامپایل کردن می شود.

نیاز به لغو یک حلقه تکرار می تواند از خارج حلقه صادر شود این کار در غالب cancellation[1] *انجام می پذیرد. فصل ۳ درباره کار با cancellation ها توضیح داده خواهد شد .*

۲-۳-۳-۲ تحلیل نتایج حلقه های موازی

هنگامی که Parallel.ForEach اجرایش پایان می پذیرد loopResult اطلاعات پیرامون نتایج ساختار ParallelLoopResult را ارائه می دهد . تابع DisplayParallelLoopResult در لیست ۲-۱۲ یک ساختار ParallelLoopResult را دریافت و Propertyهای فقط خواندنی آن را ارزیابی و نتایج اجرای حلقه Parallel.ForEach را در Console نمایش می دهد. جدول ۲-۱ سه وضعیت ممکنی که در این مثال رخ دهد را شرح می دهد .

جدول ۲-۱ : Property های فقط خواندنی ParallelLoopResult

شرط	شرح
(IsCompleted)	حلقه بطور کامل اجرا شده
((!IsCompleted) && (!LowestBreakIteration.HasValue))	حلقه بطور پارامتری با دستور Stop به پایان رسیده
((!IsCompleted) && (LowestBreakIteration.HasValue))	حلقه توسط فراخوانی دستور Break به پایان رسیده.ProPertyی LowestBreakIteration ، مقدار کمترین تکرار را درکه دستور Break فراخوانی کرده را در خود نگه می دارد.

اگر در بدنه حلقه ها Break یا Stop فراخوانی شده باشد، تحلیل نتایج اجرای حلقه موازی از اهمیت زیادی برخوردار است. ادامه دستورات بعدی به این معنا نیست که بطور کامل تمام تکرار های بعد از Break یا Stop اجرا نمی شوند. بنابراین لازم است تا مقدار ProPerty ی ParallelLoopResult را چک یا ببینم آیا بدنه های حلقه شامل مکانیسم های کنترل خصوصی شده

[1] عمل متوقفکردن فرآیند شروع شده

هست یا نه. در تمام حالت ها اگر حلقه ای موازی بدون وقوع یک استثناء به پایان بپذیرد، تمامی تکرار ها به پایان پذیرفته است. و اگر استثنائی رخ دهد ParallelLoopResult برای بازبینی وجود ندارد. مجدد، تبدیل کد های ترتیبی به کد های موازی و همزمانی کد فقط جایگزینی چند حلقه نیست. فهم مثال های بسیار گوناگونی از برنامه نویسی و ساختار های از پیش تعریف شده برای سناریو های جدید لازم است. تمام مفاهیمی که در فصل یک توضیح داده شده را، بخاطر آورید.

۲-۳-۳-۳ مدیریت استثنائات درون حلقه های موازی

چنانچه تکرار های زیادی در حالت موازی اجرا شوند، استثنائات بسیاری ممکن است رخ دهد. تکنیک های exceptionmanagement کلاسیک که در کد های ترتیبی با حلقه های موازی استفاده می شد مفید نیست. زمانی که کد درون Delegate با فراخوانی در هر تکرار موازی استثناءی درآن رخ می دهد، آن قسمتی از یک مجموعه ای از استثناء ها می شود. با کلاس جدید System.AggregateException استثناءاتی که در کد ترتیبی رخ می دهد را می توان مدیریت کرد و تقریبا از همان اصول فنی برای کد های موازی نیز می توانید استفاده کنید. تنها تفاوت در زمانی است که یک استثناء در درون بدنه حلقه ای که Delegate است رخ دهد. لیست ۲-۱۳ نسخه جدیدی از تابع درهم ParallelForEachGenerateMD5 که ParallelForEachGenerateMD5HashesException را فراخوانی می کند را نشان می دهد .

لیست ۲-۱۳ : رخ دادن و مدیریت استثناء با تابع ParallelForEachGenerateMD5Hashes

```
private static void ParallelForEachGenerateMD5HashesException()
{
	var sw = Stopwatch.StartNew();
	var inputData = GenerateMD5InputData();
	var loopResult = new ParallelLoopResult();
	try
	{
		loopResult = Parallel.ForEach(inputData,
		(int number, ParallelLoopState loopState) =>
		{
			//if (loopState.ShouldExitCurrentIteration)
			//{
			// return;
			//}
			var md5M = MD5.Create();
			byte[] data =
```

```
                Encoding.Unicode.GetBytes(
                Environment.UserName + number.ToString());
            byte[] result = md5M.ComputeHash(data);
            string hexString = ConvertToHexString(result);
            // Console.WriteLine("MD5 HASH: {0}", hexString);
            if (sw.Elapsed.Seconds > 3)
            {
                throw new TimeoutException(
                  "Parallel.ForEach is taking more than 3 seconds to complete.");
            }
        });
    }
    catch (AggregateException ex)
    {
            foreach (Exception innerEx in ex.InnerExceptions)
            {
                Debug.WriteLine(innerEx.ToString());
                // Do something considering the innerEx Exception
            }
        DisplayParallelLoopResult(loopResult);
        Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
    }
```

تکه کدی از لیست ۲-۱۳

فراخوانی Parallel.ForEach در بلوک Try قرار دارد. بنابراین بجای شرط کلاسیک catch (Exception EX) زیر catch، استثناء ات را در نظر می گیرد .

```
catch (AggregateException ex)
```

تکه کدی از لیست ۲-۱۳

AggregateException شامل یک یا چند استثناء است که در حین اجرای موازی و همزمان کد رخ داده است. بنابراین یکباری که استثنائی رخ می دهد ممکن است شامل استثناءی مجزا در مجموعه فقط خواندنی استثناءهای InnerExceptions باشد. در این حالت، Parallel.ForEach بدون هیچ تقسیم کننده خصوصی ای محتویات استثناء ها را نشان می دهد. نتیجه حلقه به نظر می رسید استفاده از کلمه کلید Stop برای توقف است زیرا ممکن است AggregateException بدست آمده باشد. شما می توانید بر اساس مشکلات که توانایی تکمیل شدن همه تکرار های حلقه را می گیرد، تصمیم بگیرید. در این حالت، حلقه ForEach ترتیبی تمام اطلاعاتی که درباره همه استثناء درInnerExceptions هست را بازیابی می کند.

```
catch (AggregateException ex)
{
	foreach (Exception innerEx in ex.InnerExceptions)
	{
		Debug.WriteLine(innerEx.ToString());
		// Do something considering the innerEx Exception
	}
}
```

تکه کدی از لیست ۲-۱۳

لیست ۲-۱٤ اطلاعاتی درباره دو استثناء اول تبدیل شده به یک رشته و ارسال خروجی عیب یابی را نشان می دهد. دو استثناء نشان داده شده اطلاعات یکسانی را در خروجی خطایابی نشان می دهند. در اکثر زمان ها از مدیریت استثناء های بسیار پیچیده و فنی استفاده می کنید و اطلاعات بیشتری درباره تکراری به معنی تولید مشکل ارائه می دهد .

این مثال بر روی اختلاف بین AggregateException و Exception سنتی تمرکز دارد. تمرین نوشتن اطلاعاتی درباره خطاها که در خروجی خطایابی یک مدیریت فنی استثناء مطرح می شود باعث پیشرفت محسوب نمی شود. در واقع، کار با مثال های پیچیده در فصل های آینده می باشد .

لیست ۲-۱٤ : خروجی خطایابی با دو استثناء یافته شده در مجموعه InnerExceptions

```
System.TimeoutException: Parallel.ForEach is taking more than 3 seconds to complete.
	at Listing2_13.Program.<>c__DisplayClassd.
<ParallelForEachGenerateMD5HashesException>b__b
(Int32 number, ParallelLoopState loopState) in
c:\users\gaston\documents\visual studio 2010\
Projects\Listing2_13\Listing2_13\Program.cs:line 220
	at System.Threading.Tasks.Parallel.<>c__DisplayClass32`2.
<PartitionerForEachWorker>b__30()
	at System.Threading.Tasks.Task.InnerInvoke()
	at System.Threading.Tasks.Task.InnerInvokeWithArg(Task childTask)
	at System.Threading.Tasks.Task.<>c__DisplayClass7.
<ExecuteSelfReplicating>b__6(Object )
System.TimeoutException:
Parallel.ForEach is taking more than 3 seconds to complete.
	at Listing2_13.Program.<>c__DisplayClassd.
<ParallelForEachGenerateMD5HashesException>b__b
(Int32 number, ParallelLoopState loopState) in
c:\users\gaston\documents\visual studio 2010\
Projects\Listing2_13\Listing2_13\Program.cs:line 220
	at System.Threading.Tasks.Parallel.<>c__DisplayClass32`2.
```

```
<PartitionerForEachWorker>b__30()
	at System.Threading.Tasks.Task.InnerInvoke()
	at System.Threading.Tasks.Task.InnerInvokeWithArg(Task childTask)
	at System.Threading.Tasks.Task.<>c__DisplayClass7.
<ExecuteSelfReplicating>b__6(Object )
```

٢-٤ تشخیص درجه موازی سازی مطلوب

گهگداری ، از تمام هسته های در دسترس در حلقه موازی نمی خواهید استفاده کنید، زیرا نیاز های خاص و طرح های بهتری برای باقی هسته های در دسترس دارید. بنابراین، حداکثر درجه موازی سازی را برای حلقه های موازی تعیین می کنید. توابع TPL همیشه سعی در رسیدن به بهترین نتیجه با استفاده از همه هسته های منطقی در دسترس هستند. متاسفانه مواردی از نتایج غیر مستدل داخلی NET Framework. در نخ های زیادی تزریق و مصرف می شوند که واقعاً مناسب نیست .در چنین حالتی یک کار بجا تخصیص حداکثر درجه موازی سازی است. در حالات دیگر، یک هسته آزاد را رها می کنید ، زیرا می خواهید برنامه کاربردی حساسی را ایجاد کنید و این هسته در اجرای بخش دیگری از کد بصورت موازی شما را یاری می رساند .

٢-٤-١ ParallelOptions

TPL اجازه تخصیص حداکثر درجه موازی سازی را توسط ایجاد نمونه ای از کلاس جدید ParallelOptions و تغییر مقدار Prppertyی MaxDegreeOfParallelism می دهد. لیست ٢-١٥ نسخه جدید از ParallelGenerateAESKeysMaxDegree و ParallelGenerateMD5HashesMaxDegree می دهد، که دو تابع معروف در استفاده موازی است .

لیست ٢-١٥ : تخصیص حداکثر درجه مطلوب موازی سازی برای حلقه Parallel.For

```
private static void ParallelGenerateAESKeysMaxDegree(int maxDegree)
{
	var parallelOptions = new ParallelOptions();
	parallelOptions.MaxDegreeOfParallelism = maxDegree;
	var sw = Stopwatch.StartNew();
	Parallel.For(1, NUM_AES_KEYS + 1, parallelOptions, (int i) =>
	{
		var aesM = new AesManaged();
		byte[] result = aesM.Key;
		string hexString = ConvertToHexString(result);
		// Console.WriteLine("AES KEY: {0} ", hexString);
```

```
	});
	Debug.WriteLine("AES: " + sw.Elapsed.ToString());
}
private static void ParallelGenerateMD5HashesMaxDegree(int maxDegree)
{
	var parallelOptions = new ParallelOptions();
	parallelOptions.MaxDegreeOfParallelism = maxDegree;
	var sw = Stopwatch.StartNew();
	Parallel.For(1, NUM_MD5_HASHES + 1, parallelOptions, (int i) =>
	{
		var md5M = MD5.Create();
		byte[] data =
			Encoding.Unicode.GetBytes(
			Environment.UserName + i.ToString());
		byte[] result = md5M.ComputeHash(data);
		string hexString = ConvertToHexString(result);
		// Console.WriteLine("MD5 HASH: {0}", hexString);
	});
	Debug.WriteLine("MD5: " + sw.Elapsed.ToString());
}
```

حالا،دو تابع ParallelGenerateAESKeysMaxDegree و ParallelGenerateMD5HashesMaxDegree مقدار maxDegree از نوع Int به عنوان حداکثر درجه مطلوب موازی سازی (maxDegree) دریافت می کنند. هر تابع نمونه ای محلی از ParallelOptions ایجاد می کند و مقدار پارامتر دریافت شده را به Property، MaxDegreeOfParallelism تخصیص می دهد که پارامتری جدید برای (قبل از بدنه) هر حلقه های موازی می باشد. این شیوه بهینه شده تا از تمامی هسته های در دسترس استفاده کند (1 - MaxDegreeOfParallelism). در عوض، اگر تعداد کل هسته های در دسترس برابر با حداکثر درجه موازی سازی تخصیص داده شده در Property باشد .این مورد بهینه می شود .

```
var parallelOptions = new ParallelOptions();
parallelOptions.MaxDegreeOfParallelism = maxDegree;
```

تکه کدی از لیست ۲-۱۵

کار با ارزش های پویا برای درجه موازی سازی مطلوب محدودیت هایی را در هنگامی که هسته های بیشتری در دسترس اند ایجاد می کند. بنابراین باید از اختیارات به دقت استفاده کنید و با مقدار های مرتبط بر اساس تعداد هسته های منطقی در دسترس کار کنید یا این تعداد را به ترتیب برای ویژگی های قابل مقیاس کد آماده کنید. اغلب سودمندی مجموعه ای از MaxDegreeOfParallelism یا

Environment.ProcessorCount، نتیجه شدن مقدار های مدیریت شده است (Environment.ProcessorCount × 2). به حالت پیش فرض، هنگامی که MaxDegreeOfParallelism تخصیص داده نشود، TPL اجازه کم و زیاد کردن تعداد نخ ها را بصورت ذهنی و بطور بلقوه با ProcessorCount می دهد. بهمین خاطر بهتر است بار کاری، ورودی خروجی و CPU با هم پشتیبانی شود. هر دو تابع ParallelGenerateAESKeysMaxDegree و ParallelGenerateMD5HashesMaxDegree را بر اساس مقداری پویا که تعداد هسته های منطقی در زمان اجراست میتوانید فراخوانی کنید. این کد دست کم به ماشینی با دو هسته منطقی یا استثنایی نیاز دارد.

```
// This code requires Environment.ProcessorCount > 2 to run
ParallelGenerateAESKeysMaxDegree(Environment.ProcessorCount - 1);
ParallelGenerateMD5HashesMaxDegree(Environment.ProcessorCount - 1);
```

تک کدی از لیست ۲-۱۵

حلقه Parallel.For سعی در کار کردن با تعداد کل هسته های منطقی منهای یک دارد. بطور مثال اگر کدی در ریزپردازنده quad-core اجرا شود، آنها سه هسته توسط این حلقه استفاده می شود .

کدی که در ادامه آمده بهترین تمرین برای کد نهایی نمی باشد. برخی اوقات شما می خواهید بدانید چگونه دو تابع موازی را با بهترین کارایی موازی کنید هنگامی که اجرا در زمان یکسانی با استفاده از هسته های بسیاری می باشد. وضعیت کد زیر را می توانید امتهان کنید :

```
Parallel.Invoke(
() => ParallelGenerateAESKeysMaxDegree(2),
() => ParallelGenerateMD5HashesMaxDegree(2));
```

تکه کدی از لیست ۲-٦

دو تابع بصورت موازی شروع شده اند و هر کدام سعی در بهینه سازی اجرا با استفاده از دوتا از چهار هسته موجود در ریز پردازنده quad-core دارند. برگشت آشکار بطور پویا از تعداد هسته ها استفاده می کند. با این وجود این مورد تنها برای آزمایش عملکرد اهداف می باشد .

ParallelOptions دو ProPertyی زیر را برای کنترل اختیارات پیشرفته ارائه شده است :

- ✓ CancellationToken : این پارامتر اجازه تخصیص نمونه ای جدید از System.Threading. CancellationToken را برای آگاهی از گسترش عملیات موازی لغو شده می دهد. در فصل سوم درباره کاربرد عمل متوقف کردن فرآینده شروع شده(Cancellation) توضیح داده می شود.

TaskScheduler اجازه تخصیص نمونه ای خصوصی از System.Threading.Tasks TaskScheduler. را می دهد. معمولا نیازی به تعریف زمانبند Task خصوصی شده نیست تا Task های موازی زمانبندی بشوند، مگر اینکه با تعداد زیادی از الگریتم های خاص کار می کنید. برای مثال اگر حداکثر درجه موازی سازی را در سرتاسر حکم های حلقه های موازی می خواستید، می بایست برای هر حلقه از ParallelOptions استفاده کنید. اما اگر می خواستید تمام حلقه ها را متراکم سازید تا از حداکثر درجه موازی سازی ویژه کمتر استفاده شود، شما به TaskScheduler برای هماهنگ کردن تمامی حلقه ها نیاز دارید. فصل هشت استفاده از کلاس TaskScheduler توضیح داده می شود .

۲-٤-۲ شمارش نخ های سخت افزاری

Environment.ProcessorCount تعدا هسته های منطقی را ارائه می کند. چنانچه در فصل یک آموختید، تعداد هسته های منطقی با تعداد هسته های مجازی فرق دارد.

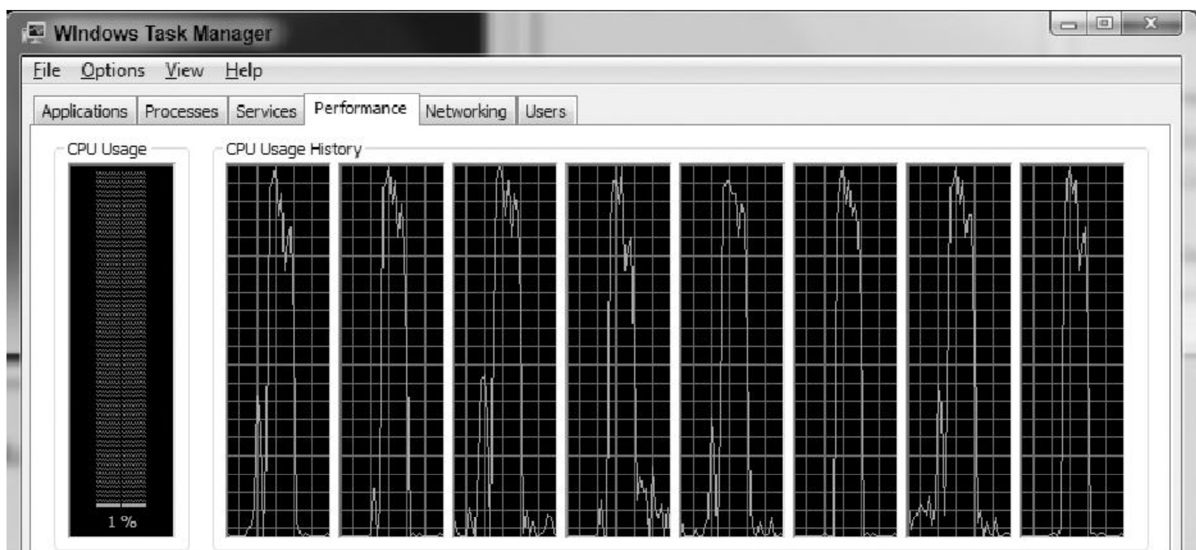
برای مثال ، یک ریزپردازنده Core i7 با چهار هسته فیزیکی دارای فناوری Hyper-Threading دوبرابر می باشد بنابراین دارای هشت هسته منطقی می باشد . در این حالت Environment.ProcessorCount دارای مقدار هشت می باشد نه چهار .همچنین سیستم عامل با هشت پردازشگر منطقی کار می کند.

تمام کدهای که بوسیله TPL نوشته شده اند با استفاده از نخ های نرم افزاری چندگانه اجرا می شوند. نخ ها دارای راه های باریک سطح پایینی برای اجرای بخش های زیادی از کد های موازی هستند و مزیت حضور هسته های چند گانه در سخت افزار ها استفاده می کنند.در اکثر اوقات، کد در روند اجرا برخی نقص هایی دارد. نخ ها زمان بیکاری ای که سیستم منتظر می ماند تا اینکه داده ای از کش های ریزپردازنده های در دسترس یا حافظه سیستم آورده شود را کاهش می دهند. بدین معنی که واحد های اجرایی بیکاری در آن وجود دارد. مثال ساده ای از استفاده از مزیت واحد های بیکار این است که دو نخ کار مختلف ریاضی ای را در حال اجرا هستند چنانچه هر دو تقریباً بطور همزمان با منابع مختلف روی چیپ اجرا می شوند.

Hyper-Threading فناوری است با افزایش سطح دستوالعمل موازی ارائه شده حالت های وابسته به معماری را دوبرابر می کند و نقص اجرای موازی دو نخ را برطرف می کند. این شیوه در ریزپردازنده ای که دست کم دارای دو هسته فیزیکی باشد رخ می دهد. همچنین هسته های منطقی نخ های سخت افزاری اند.
شکل از Windows Task Manager در شکل ۲-۱۱ و Resource Monitor در شکل ۲-۱۲ نشان داده شده است. در این تصاویر نمایشی گرافیکی از میزان استفاده از CPU نشان داده شده است .شکل ها هشت گراف را در پردازشگر quad-core با هشت نخ سخت افزاری که توسط فناوری Hyper-Threading ایجاد شده را نشان می دهند .

۲-٤-۳ هسته های منطقی هسته های فیزیکی نیستند

درک هسته های منطقی خیلی مهم است و اصلا هسته فیزیکی نمی باشد. هنگامی که هر هسته فیزیکی دارای دو نخ سخت افزاری با جریان های از دستوالعمل های مستقل باشد، این تکنیک باعث افزایش کارایی در سطح دستور العمل های موازی می شود. اگر نخ های نرم افزاری دارای وابستگی های نرم افزاری داده ای زیادی نباشند، بهبود هایی را در عملکرد می توان انتظار داشت. میزان این بهبودی به نوع برنامه اجرایی بستگی دارد.



شکل ۲-۱۱

بطور پیش فرض، TPL از تعدادی از نخ های سخت افزاری یا هسته های منطقی برای اجرا بهینه استفاده می کند نه تعداد هسته های فیزیکی .بنابراین ، زمانی ممکن است الگریتم خاصی را بیابید که هسته های زیاد مورد انتظار را ارائه نکنند با اینکه آنها هسته های فیزیکی واقعی هستند.

برای مثال، اگر الگریتمی SpeedUPی برابر ٦.٥ x را در پردازنده ای با هشت هسته منطقی ارئه کند، فناوری Hyper-Threading آن را با ٤.٥ x در ریز پردازنده ای با چهار هسته فیزیک و هشت هسته منطقی اجرا می کند .

برای بهبود کارایی هر دو تابع ParallelGenerateAESKeysMaxDegree و ParallelGenerateMD5HashesMaxDegree را فراخوانی کنید یکبار با تعداد هسته های فیزیکی و بار دیگر با تعداد هسته های منطقی (نخ های سخت افزاری) و سپس SpeedUPی که توسط هسته های منطقی اضافی بدست آمده را اندازه گیری کنید. در رایانه ای با ریزپردازنده quad-core و هشت نخ سخت افزاری، خط زیر را می توانید اجرا کنید :

```
ParallelGenerateAESKeysMaxDegree(4);
ParallelGenerateMD5HashesMaxDegree(4);
```

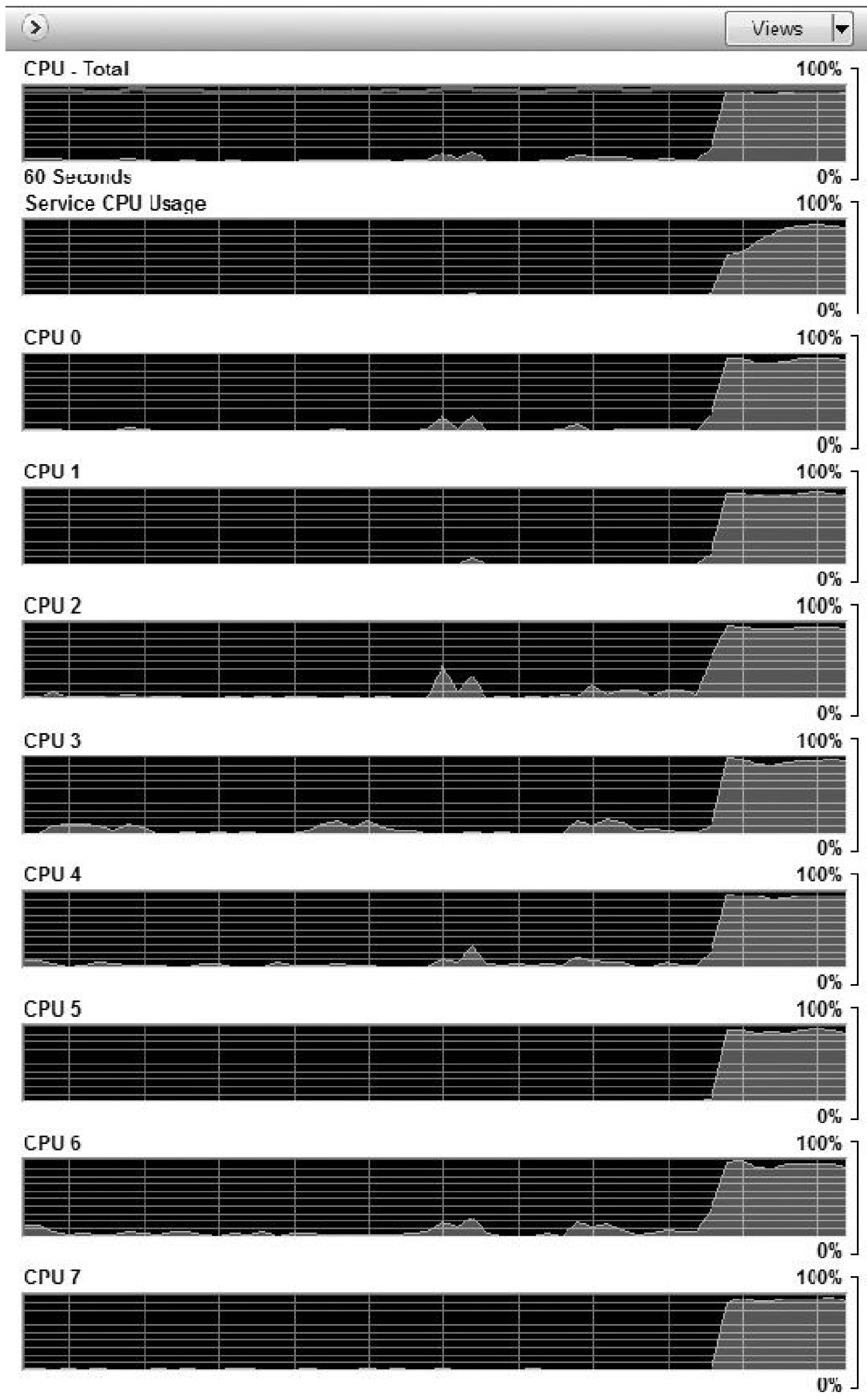
تکه کدی از لیست ٢–٧

و سپس این خط ها را اجرا کنید :

```
ParallelGenerateAESKeysMaxDegree(8);
ParallelGenerateMD5HashesMaxDegree(8);
```

تکه کدی از لیست ٢–٨

اینکه هر دو کد را تحت شرایط یکسانی اجرا کنید ، از اهمیت زیادی برخوردار است. در پیکره بندی های سخت افزاری خاص ،این مثال ساده ٥.٢ ثانیه زمان برای اجرای با حداکثر درجه موازی سازی با ٤ مجموعه و ٤.٦٠ ثانیه با ٨ مجموعه نیاز دارد .SpeedUP اضافی هنگامی که چهار هسته منطقی اضافه شده است برابر برابر ١.١٣ x = ٥.٢٠/٤.٦٠ می باشد. این مثال نشان دهنده اهمیت درک سخت افزارهای نهفته درآزمایش و اجتناب از نتایج اشتباه می باشد.



شکل ۲-۱۲

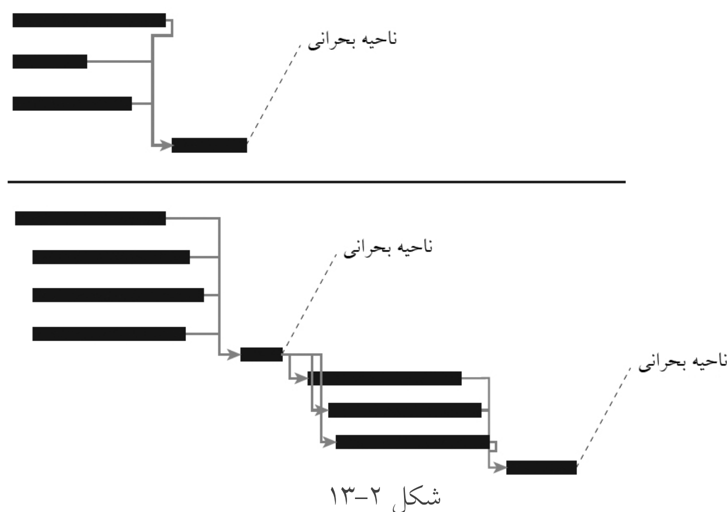
۲-۵ استفاده از نمودار گانت[1] برای تشخیص نواحی بحرانی

هنگامی که کانونی بحرانی که قابل موازی سازی است را در حلقه ای با داده موازی تشخیص داده اید، می توانید از ساختار TPL ی که فراگرفته اید استفاده کنید تا از مزیت هسته های منطقی چندگانه سود ببرید. چنانچه در فصل یک شرح داده شد، نواحی بحرانی همیشه وجود دارند و دستیابی به SpeedUP را در موازی سازی به حداقل می رسانند.

همانند برنامه نویسیِ ترتیبیِ سنتی، برنامه نویسی موازی فقط در مورد نوشتن کد نیست. همیشه یک طراحی خوب کمک می کند. نمودار گانت نمودار نواری ای هست که معمولاً برای تشریح زمانبندی یک پروژه و وابستگی های مرتبط با Task های متصل استفاده می شود. بنابراین، بنابراین نمودار گانت در جاهای که موازی سازی بی نهایت سخت یا تقریباً ناممکن است کمک به تشخیص نواحی بحرانی می کند.

کار با نمودار گانت باعث می شود تا ارتباط وابستگی بین Task ها و Taskهایی که می توانند موازی اجرا شوند و آنهایی که نمی توانند را تشخیص دهید. در این راه حل، نواحی بحرانی به سادگی یافته می شوند. در شکل ۲-۱۳ نمودار گانت برای هدف بیان شده رسم شده است .

هنگامی که راه حلی را طراحی می کنید چیزی که مهم است این مطب می باشد که امکان اجرای همزمان را نیز در نظر داشته باشید. نمودار گانت ابزارهای بزرگی را برای کمک به یافتن امکان موازی سازی در الگریتم های پیچیده می کند.



شکل ۲-۱۳

[1] Gantt chart

۲-۶ خلاصه

در این فصل دیگر مزیت های مرتبط با موازی سازی داده های گفته شد. این فصل تنها خراشی سطحی از مدل برنامه نویسی Task-Based با NET Framework 4. توسط بازدیدی مختصر ازclass ها ، structure ها و enumeration ها را معرفی کرد . همچنین درباره مفاهیم مرتبط با همزمانی و طرح های برنامه نویسی موازی در وسایل موازی سازی داده با استفادهها از یک ترکیب نحوی بحث شد. خلاصه ای از فصل :

- ✓ شما مجبور به طراحی و ترسیم موازی سازی و همزمانی در ذهن تان هستید. TPL ساختار های را فراهم ساخته تا فرایند ایجاد کد های که از مزیت هسته های پردازشگر استفاده می کنند ساده شود .
- ✓ شما نیازی به کامپایل مجدد برای استفاده از امکانات هسته های اضافی ندارید. TPL حلقه های موازی را بهینه می کند و کار توزیع Task ها در نخ های سخت افزاری را با استفاده از زمانبند توزیع بار کاری و بر اساس منابع سخت افزاری دردسترس در زمان اجرا انجام می دهد.
- ✓ حلقه های موازی می توانید اجرا کنید .
- ✓ هنگامی که با حلقه های موازی کار میکنید، مجبورید عدم ترتیب اجرای را نیز به حساب بیاورید.
- ✓ با یک خط کد ساده، کار موازی کردن Task ها را می توانید آغاز کنید. همچنین مجبورید کدهایتان را برای مقیاس های بزرگتر آماده کنید .
- ✓ زمانی که راه حل هایی را طراحی می کنید، مجبورید ناحیه های بحرانی و امکان های موازی سازی را حساب کنید.
- ✓ مجبورید تعداد هسته های فیزیکی و منطقی(نخ های سخت افزاری) را بدانید تا نتایج صحیحی را اندازه گیری و رسم کنید.
- ✓ از عبارات Lambda و delegate های بی نام برای ایجاد کد های موازی ساده می توانید استفاده کنید.

نتیجه‌گیری و پیشنهادات

مطب پیش رو ترجمه ای از کتاب Professional Parallel Programming with CSharp نوشته Gaston C. HILLAR می باشد. دوفصل از ۱۱ فصل کتاب را برای درس پروژه دوره لیسانس آماده کرده ام. منبع فوق کلاً در مورد امکانات محیا شده توسط Microsoft در Visual Studio 2010 می باشد. امکانت زیادی برای برنامه نویسی تا به حال توسط شرکت های مختلف ایجاد شده است که از آن جمله می توان این دو مورد را نام برد :

۱. parallel studio شرکت Intel برای موازی سازی پردازنده های این شرکت در زبان برنامه نویسی Visual C++ .

۲. Multi Pascal که از زبان Pascal برای برنامه نویسی چند پردازنده ای و چند کامپیوتری استفاده می کند .

مواردی این چنین زیاد اند، مانند ابزار های موازی سازی در Matlab .همه این مطالب گویای جهت گیری سریع و با سرعت دنیای سخت افزار و در پی آن دنیای نرم افزار است. Visual Studio با دارا بودن زبان های برنامه نویسی گوناگون و قابلیت افزودن کامپوننت ها ابزار های بسیار مناسبی را برای افزایش سرعت و دقت در برنامه نویسی موازی در اختیار برنامه نویسان قرار می دهد و این امر زمانی محقق می شود که برنامه نویسان دست از برنامه نویسی مبتنی بر پایگاه داده برداشته و به سمت برنامه هایی پیچیده تری بروند تا نرم افزار هایی که در عین پیچیدگی دارای سهولت در موازی سازی نیز هستند بنویسند. در راستای این امر باید سخت افزار های روز را شناخته و طریقه استفاده بهینه آنها را فرا گرفت. چنانچه گفته شد بطور معمول شرکت های بزرگ بسیاری امکانات نرم افزاری را برای استفاده برنامه نویسان از امکانات جدید سخت افزارهای تولیدی فراهم می کنند که خود باعث سهولت در استفاده بهینه این امکانات جدید می شود. امید است که با شناخت و آگاهی از این امکانات، شاهد نوشتن برنامه هایی به روز که از تمامی امکانات سخت افزاری استفاده می کنند باشیم.

محمد زحمتکش کناری

۱۳۹۰ / ۵ / ۱۲

Zahmatkesh8519@gmail.com

مراجع

[۱] کتاب "Professional Parallel Programming with CSharp" از انتشارات Wrox که توسط آقای Gaston C.hillar نوشته شد و در سال ۲۰۱۱ چاپ گردید.