مجله علمی ترویجی انجمن مهندسان مکانیک ایران
ISSN: 1605-9719
سال ۲۶، شماره ۱۱۷، صفحات ۱۱–۲۱

افزایش کارایی کدهای سریال محاسباتی به روش پردازش موازی با رویکرد OpenMP

محمد روح اللهی[۱ و *]، فرهاد قدک[۲]

[۱] کارشناس ارشد مهندسی مکانیک، دانشگاه آزاد اسلامی واحد تهران جنوب، تهران،
[۲] دانشیار گروه هوافضا، دانشکده فنی و مهندسی، دانشگاه جامع امام حسین(ع)، تهران
*مسئول مکاتبات: m.a_rouhollahi@yahoo.com

◄ چکیده

امروزه شاهد گسترش روزافزون و اجتناب ناپذیر استفاده از رایانه‌ها در علوم و صنایع مختلف به منظور پردازش داده‌ها و انجام محاسبات هستیم. در این بین حجم محاسبات علمی در برخی از صنایع همچون صنعت هوافضا به دلیل سنگینی و پیچیدگی محاسبات میزان بالایی از پردازش را می‌طلبد و همچنین مقابله به موقع با چالش‌های فراوان پیش رو، نیاز به سرعت را نیز افزایش می‌دهد. دستیابی به روش‌های نوین، که پاسخگوی نیازها بوده و مشکلات ابر رایانه‌ها را تا حد زیادی مرتفع سازد به شدت احساس می‌شود. در این مقاله، به بررسی عملکرد کدهای محاسباتی در حالت سریال و موازی به روش حافظه اشتراکی پرداخته می‌شود. و در نهایت کد دو بعدی لاپلاس موازی خواهد شد. نکته متمایز کننده این تحقیق تلاش برای افزایش کارایی و عملکرد اجرای برنامه‌ها از طریق دست یافتن به الگوریتم‌هایی برای بهبود عملکرد کدهای سریال می‌باشد. آنچه حاصل گردید عملکرد مناسب پردازش موازی با افزایش تعداد هسته‌های سیستم بر روی کدهای محاسباتی بود.

◄ واژگان کلیدی

پردازش موازی
حافظه اشتراکی
افزایش کارایی
ابر رایانه‌ها
معادله لاپلاس
OpenMP

◄ تاریخچه مقاله

تاریخ دریافت ۱۳۹۵/۱۱/۱۱
تاریخ پذیرش ۱۳۹۷/۰۴/۱۶

۱ مقدمه

تا چند دهه قبل دانش بشری بر پایه دو ستون آزمایش‌های تجربی و آنالیزهای تحلیلی استوار بوده است. محدودیت‌های موجود در استفاده از این دو روش باعث شده است که دانشمندان و متخصصین به بهره‌برداری از روش‌های عددی برای حل معادلات حاکم بر پدیده‌های فیزیکی رو آورند. بدین ترتیب شبیه‌سازی‌های رایانه‌ای و روش‌های عددی به عنوان ستون سوم علم و دانش بشری وارد عرصه شده و امروزه از جایگاه ویژه‌ای برخوردار هستند. با توجه به این واقعیت که ابزار اصلی در انجام محاسبات و شبیه‌سازی‌های عددی، ماشین‌های حساب و رایانه‌های دیجیتالی می‌باشند، همواره توان پردازشی سخت افزاری یکی از عوامل اصلی محدود کننده در بهره‌برداری از روش‌های عددی در حل مسائل علمی و صنعتی بوده است. بدین جهت ارتقای توان محاسباتی و حجم سامانه‌های سخت افزاری همواره مد نظر بوده است. با پیشرفت تکنولوژی و دستیابی محققین به رایانه‌های پر سرعت و با حافظه بیشتر، رفته رفته امکان محاسبات حجیم[۱] ارتقا یافته است. از طرفی تمایل و نیاز محققین در علوم و صنایع به حل مسائل پیچیده و با لحاظ نمودن فیزیک واقعی‌تر، محرک اصلی برای ساخت رایانه‌های پر قدرت‌تر بوده است. بدین صورت در یک حلقه افزاینده، از یک طرف بر پیچیدگی محاسبات عددی افزوده شده و دانش بشری در روش‌های عددی و شبیه سازی‌های رایانه‌ای ارتقا یافته و از طرف دیگر تکنولوژی سخت افزار رایانه‌ای مرتبا در حال تکامل و ارتقا به سمت طراحی و ساخت ابر رایانه‌های پر قدرت‌تر حرکت نموده است. دانشمندان علوم رایانه از همان ابتدا متوجه محدودیت‌های مختلف برای ساخت یک پردازشگر بسیار سریع بوده و بدین جهت، استفاده همزمان از چندین پردازشگر برای انجام محاسبات مورد توجه بوده است. امروزه با نزدیک شدن تکنولوژی ساخت پردازشگرها به برخی محدودیت‌های فیزیکی، استفاده از پردازش موازی تقریبا در تمامی انواع پردازشگرها تبدیل به یک ضرورت نموده است. در نتیجه برنامه نویسی و تدوین روش‌های عددی موازی برای حل مسائل علمی یک ضرورت می‌باشد. صرفه اقتصادی و استفاده از فناوری‌های روز از جمله برتری‌هایی است که برای پردازش موازی نسبت به پردازش سریال (روش سنتی تر پردازش اطلاعات) برشمرده می‌شود. یکی از دلایل اصلی استفاده از برنامه‌نویسی موازی، بحث افزایش سرعت اجرای برنامه می‌باشد، که این امر در پردازنده‌های تک هسته‌ای دارای محدودیت افزایش توان مصرفی می‌باشد که اگر تعداد این ترانزیستورها زیاد باشد، گرمای ایجاد شده ناشی از توان مصرفی پردازنده باعث ذوب شدن آن خواهد شد [۱۰].

پردازش موازی[۲] یا محاسبات موازی به اجرای یک فرآیند بطور همزمان، عموما با تقسیم عملیات پردازش بر روی چندین پردازنده به منظور افزایش کارایی و در نهایت سرعت بخشیدن به رسیدن جواب مسئله است. گاهی استفاده از تکنیک‌های اشتراک زمان را در یک پردازنده، به اشتباه پردازش موازی محسوب می‌شود(چند پروسه به طور موازی روی یک پردازنده اجرا می‌شوند). ایده این کار بر این مبنا است که هر مسئله بطور معمول قابل تقسیم به چندین مسئله با اندازه کوچک‌تر است که این مسئله‌های کوچک‌تر می‌توانند به‌صورت همزمان حل شده و در نهایت ادغام شوند تا نتیجه نهایی سریع‌تر بدست آید. کاهش زمان محاسبه، امکان حل مسائل بزرگتر، غلبه بر محدودیت‌های حافظه، صرفه اقتصادی[۳] و استفاده از فناوری‌های روز از جمله

[1] Massive computing [2] Parallel Processing [3] economic efficiency

برتری‌هایی است که برای پردازش موازی نسبت به پردازش سریال (روش سنتی پردازش اطلاعات) بر شمرده می‌شود.

با توجه به شکل ۱، یک مسئله محاسباتی از چندین دستورات تشکیل شده است و این دستورات بطور متوالی یکی پس از دیگری در اختیار پردازنده مرکزی قرار می‌گیرند و پردازنده مرکزی تنها از یک هسته (که عملا از هسته اصلی خود استفاده می‌کند) برای پردازش دستورات استفاده می‌نماید [۱، ۱۱].

شکل ۲، اجرای برنامه در حالت موازی را نشان می‌دهد. در پردازش موازی مسئله به چند بخش مجزا که می‌توانند بطور همزمان اجرا شوند تقسیم می‌شود. سپس هر قسمت به یکسری دستورات شکسته شده و این دستورات بطور همزمان در پردازنده‌های مختلف اجرا می‌شوند [۲، ۱۰، ۱۱].

۲ مزایای استفاده از پردازش موازی

- نسبت هزینه به‌کارایی بسیار پایین
- سخت‌افزار و نرم‌افزار ارزان و در دسترس
- تعمیر و نگهداری ساده
- قابلیت توسعة سیستم متناسب با افزایش نیاز
- زمان بالای در اختیار بودن و سرویس‌دهی سیستم
- کاهش زمان اجرا در شبیه سازی‌ها و حل مسائل کاربردی
- امکان حل مسائل بزرگتر و پیچیده تر در زمان کوتاه‌تر [۱۱].

۳ انواع روش‌های موازی سازی

روش‌های موازی سازی کدهای محاسباتی به دو دسته کلی تقسیم می‌شوند.

الف) موازی سازی با CPU[1]

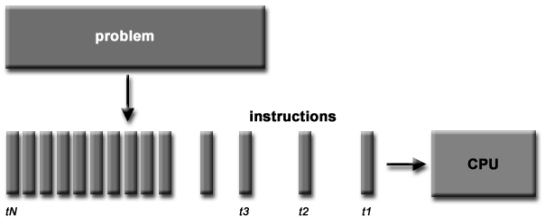
ب) موازی سازی با GPU[2]

موازی سازی با CPU می‌تواند با دو روش OpenMP[3] و MPI[4] بر روی کدهای محاسباتی اعمال می‌شود. در این تحقیق به بررسی و عملکرد روش OpenMP پرداخته شده است [۴، ۵].

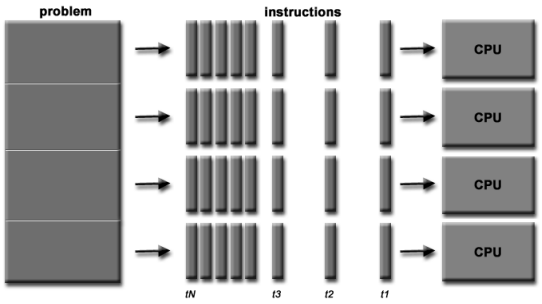
۴ تعریف OpenMP

OpenMP یک واسط برنامه نویسی کاربردی[5] برای برنامه نویسی موازی نخ‌ها در سیستم‌های حافظه اشتراکی با یکی از سه زبان C/C++/Fortran است. و از معماری‌های مختلفی از جمله پلتفرم‌های ویندوز و یونیکس پشتیبانی می‌کند.

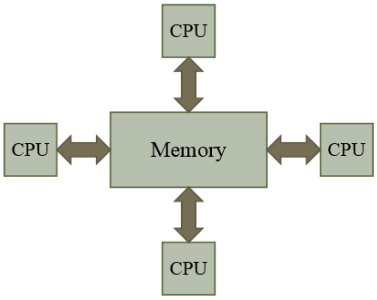
OpenMP قابلیت انجام دو سطح از موازی سازی یعنی موازی سازی سطح داده‌ها[6] و موازی سازی سطح دستورات[7] را دارا می‌باشد و همچنین ترکیب نواحی سریال و موازی در یک سورس کد را دارد. OpenMP از یکسری راهنماهای کامپایلر[8]، توابع کتابخانه‌ای[9] و متغیرهای محیطی[10] تشکیل شده است که بر روند اجرای برنامه تاثیر می‌گذارند. OpenMP تضمین نمی‌کند که از حافظه اشتراکی استفاده بهینه خواهد کرد. همچنین مواردی مانند وابستگی داد[11]، شرایط مسابقه[12] یا بن بست‌ها[13] ۱۶ باید توسط برنامه نویس در کد برنامه کنترل شود. همزمان سازی ورودی و خروجی هنگام دسترسی موازی و چک کردن ترتیب اجرای کد برنامه نیز از جمله وظایف برنامه نویس می‌باشد و از عهده OpenMP خارج است [۳، ۶، ۷، ۱۱].



شکل ۱: چگونگی عملکرد یک برنامه سریال [۱۱]



شکل ۲: چگونگی عملکرد یک برنامه موازی [۱۱]



شکل ۳: حافظه اشتراکی [۱۱]

۵ مزایای استفاده از OpenMP

- سادگی انجام موازی سازی برنامه سریال با برچسب زنی[14] کد برنامه که موازات را نشان می‌دهد.
- مقیاس پذیری[15] و کارایی بسیار بالا در صورت استفاده صحیح
- قابل حمل بودن[16] برنامه نوشته شده با OpenMP، به دلیل پشتیبانی بسیاری از کامپایلرها از OpenMP
- عدم نیاز به برنامه نویسی‌های پیچیده توسط برنامه نویس
- اجرا شدن هر نخ برنامه در OpenMP توسط نخ‌های سخت افزاری [۳، ۱۱].

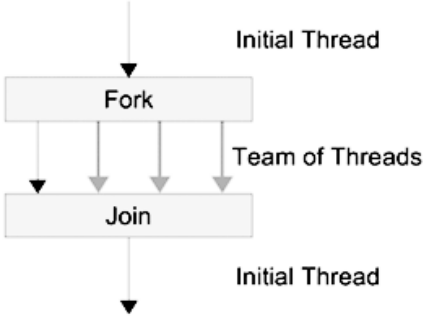
[1]Center processing Unit [2]Graghic Processing Unit [3]Open Multi Processing [4]Message Passing Interface [5]Application Programming interface [6]Data Level Parallelism [7]Instruction Level Parallelism [8]Compiler Directive [9]Library Routines [10]Enviroment Variable [11]Data Dependency [12]Race Condition [13]Dead Lock [14]Annotation [15]Scability [16]Portable

۶ حافظه اشتراکی

تمامی پردازنده‌ها دسترسی به حافظه به‌عنوان یک فضای یکپارچه و آدرس‌دهی یکسان دارند. پردازنده‌های مختلف مستقلا عمل کرده ولی از حافظه مشترک استفاده می‌کنند. هر تغییر در حافظه توسط یک پردازنده، بلافاصله توسط تمامی پردازنده‌ها قابل رویت می‌باشد [۵، ۱۱].

۷ طرز کار OpenMP

برنامه های چند نخی[1] می‌توانند به روش‌های مختلفی نوشته شوند که در برخی از آنها برهمکنش بین نخ‌ها بسیار پیچیده است. اما OpenMP به دنبال فراهم کردن سهولت در برنامه نویسی است و با استفاده از یک نگرش ساختار یافته در برنامه نویسی چند نخی، به کاربر کمک می‌کند تا از خطاهای برنامه نویسی اجتناب کند. این روش از یک مدل برنامه نویسی به نام Fork-join استفاده می‌کند که در شکل‌های ۴ و ۷ نشان داده شده است. بر اساس این نگرش، برنامه با یک نخ اجرایی همچون یک برنامه سریال آغاز می‌شود. به این نخ که برنامه با آن اجرا می‌شود نخ اولیه یا نخ اصلی گفته می‌شود. به محض مواجه شدن ساختارموازی OpenMP با نخ ذکر شده در هنگام اجرای برنامه، یک گروه از نخ ها ایجاد می‌شوند (مرحله fork). در این لحظه نخ اصلی با سایر اعضا همکاری کرده و کد مربوطه به صورت دینامیکی اجرا می‌شود. در انتهای ساختار موازی، تنها نخ اولیه یا همان master thread ادامه یافته و مابقی منقضی می‌شوند. (مرحله join) هر قسمت از کد که درون ساختار موازی قرار می‌گیرد ناحیه موازی نامیده می‌شود که این نواحی در شکل ۷ کاملا مشهود است. و اگر دوباره به یک ناحیه موازی دیگری برسد یکسری نخ مجددا ایجاد می‌شوند که تعداد آنها با ناحیه موازی اول می‌تواند متفاوت باشد [۴، ۵، ۸، ۱۱].



شکل ۴: مدل برنامه نویسی Fork-join در OpenMP [۱۰]

۸ ترکیب[2] اصلی OpenMP

در موازی سازی به روش OpenMP تغییرات در سورس کد اولیه تنها در نقاط کمی اتفاق می افتد و برای بهبود عملکرد برنامه نیاز به نوشتن قسمت‌های مختلف کد نیست.

در برنامه های C/C++ برای کنترل موازی سازی از pragma استفاده می‌شود که به آن راهنما[3] می‌گویند. این راهنما همیشه با #pragma omp آغاز می‌شوند سپس یک کلمه کلیدی برای راهنما قرار داده می‌شود و پس از آن از یک یا چند عبارت[4] به منظور کنترل بیشتر پردازش موازی استفاده می‌شود.

در برنامه Fortran از راهنمای $!omp استفاده می‌شود. سپس باید سرآیند[5] مربوط به OpenMP که شامل پرتوتایپ توابع و سایر اطلاعات اضافه گردد [۱۰، ۱۱].

در زبان C/C++ دستور سرآیند برابر با:

```
#include <omp.h>
```

در زبان Fortran دستور سرآیند برابر با:

```
Use omp_lib
```

قطعه کد زیر در برنامه Fortran اجرا شده است.

```
Program name
 Use omp_Lib
   .
   .
 !$omp parallel
Do i=1,N
 Tnew(i)=T(i-1)-T(i)+T(i+1)
End do
 !$omp end parallel
     .
     .
End program name
```

شکل ۵: نمونه قطعه کد موازی شده در برنامه فرترن

در موازی سازی دو نوع متغیر تعریف می‌شود:

متغیر اشتراکی[6]: به تمامی متغیرهایی که بیرون از ناحیه موازی قرار دارند اطلاق می‌شود.

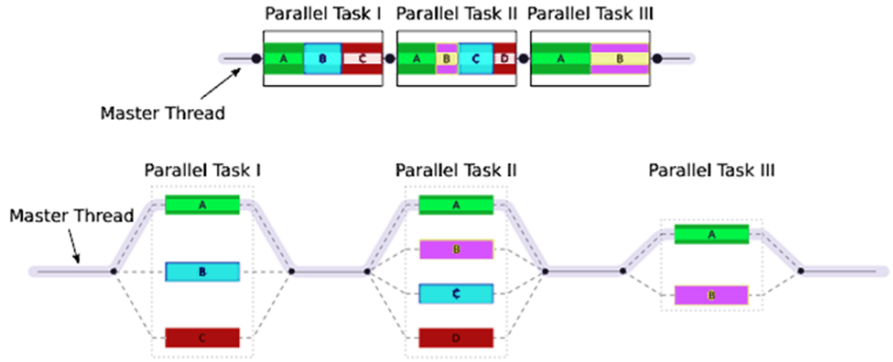
متغیر اختصاصی[7]: به تمامی متغیرهایی که درون ناحیه موازی قرار دارند اطلاق می‌شود.

تمامی متغیرهایی که درون ناحیه موازی قرار دارند بعد از ناحیه موازی از بین خواهند رفت. بنابراین اگر در ادامه کد به این متغیرها نیاز باشد باید بصورت اشتراکی تعریف شوند. در شکل ۶ به متغیرهای T، T_{new} در ادامه کد نیاز است لذا بصورت اشتراکی تعریف شده‌اند.

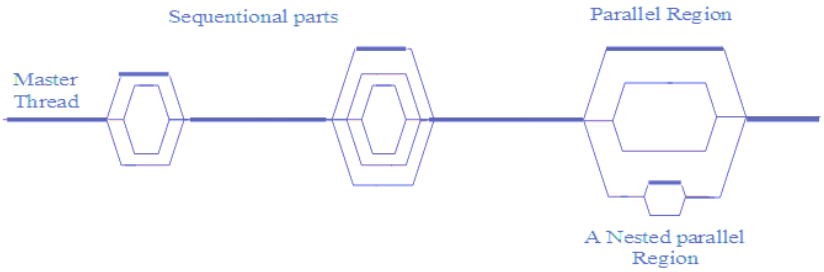
```
program name
 use omp_lib
   .
   .
!$omp parallel shared(T,Tnew)
  Do i=1,N
    Tnew(i)=T(i-1)-T(i)+T(i-1)
  End do
!$omp end parallel
  Do i=1,N
      T(i)=Tnew(i)
      End do
     .
     .
End program name
```

شکل ۶: قطعه کد نمونه همراه با تعریف متغیر اشتراکی

[1]Thread [2]Syntax [3]Directive [4]Clause [5]Header [6]Shared Variable [7]Private Variable



شکل ۷: نواحی موازی و همچنین Master Thread [۱۱]



شکل ۸: تولید نخ و ناحیه موازی تودرتو و نواحی سریال

۹ شرایط مسابقه و همگام سازی[۱]

در سیستم‌های حافظه مشترک ارتباط پردازنده‌ها با یکدیگر از طریق حافظه مشترک صورت می‌گیرد. بنابراین اگر پردازنده‌های توسط حافظه مشترک بر روی داده‌ای که قرار است پردازنده دیگر آنرا فراخوانی کند، اثر بگذارد باعث ایجاد شرایط مسابقه می‌شود. این عمل باعث می‌شود که در اجراهای مختلف خروجی‌های متفاوت ایجاد شود.

یکی از راه‌های رفع شرایط مسابقه استفاده از ابزارهای همگام سازی می‌باشد. این ابزارها هزینه بر هستند و تا آنجایی که ممکن است باید استفاده از آنها را به حداقل رساند. از این ابزارها برای سازماندهی و ایجاد ترتیب و همچنین حفاظت از دسترسی به داده‌های اشتراکی استفاده می‌شود.

ابزارهای همگام سازی به دو دسته سطح بالا[۲] و سطح پایین[۳] تقسیم می‌شوند. در ادامه به بررسی چند نوع از ابزارهای همگام سازی خواهیم پرداخت [۱۰، ۱۱].

ابزارهای همگام سازی سطح بالا عبارتند از:

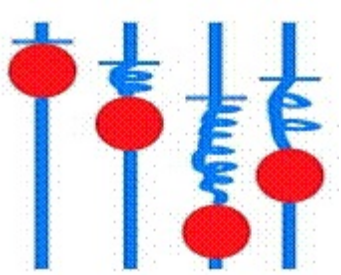
Barrier-atomic-critical-ordered

ابزارهای همگام سازی سطح پایین عبارتند از:

Flush-locks

۱.۹ ساختار barrier

barrier یکی از ابزارهای همگام سازی می‌باشد. در محلی که از barrier استفاده می‌شود، هر نخ منتظر می‌ماند تا سایر نخ ها به آن محل برسند، سپس به کار خود ادامه می‌دهند.



شکل ۹: ساختار Barrier [۱۱]

در واقع Barrier باعث توقف بعضی از نخ‌ها بصورت موقت می‌باشد. دستور اجرای آن در زبان Fortran بصورت زیر است:

```
!$omp barrier
```

و در زبان C/C++ بصورت زیر است:

```
#pragma omp barrier
```

```
#pragma omp parallel
{
   int id = omp_get_thread_num();
   A[id] = big_calc1(id);
#pragma omp barrier
   B[id] = big_calc2(id, A);
}
```

شکل ۱۰: کد نمونه ابزار همگام سازی barrier در زبان C

با توجه به شکل بالا، متغیر A یک آرایه به صورت اشتراکی است. هر کدام از نخ‌ها تابع big_calc1 را که شامل یکسری محاسبات طولانی

[1]Synchronization [2]High Level [3]Low Level

می‌باشد برای id مربوط به خود اجرا می‌کنند و نتیجه را در یکی از اعضای A که مربوط به آن نخ است انتساب می‌دهند. هر کدام از نخ‌ها با سرعت متفاوتی این محاسبات را انجام می‌دهند. همه اعضای آرایه A به صورت همزمان به روز نمی‌شوند. به دلیل آنکه قرار است در ادامه کد از متغیر A استفاده شود، از دستور barrier برای به روز رسانی استفاده شده است [۱۱].

```
!Somp parallel
integer:: id
id=omp_get_Thread_num()
A(id)=big_calc1(id)
!Somp barrier
B(id)=big_calc2(id,A)
!Somp end parallel
```

شکل ۱۱: کد نمونه ابزار همگام سازی barrier در زبان فرترن

۲.۹ ساختار Critical

در این ساختار فقط یک نخ در یک زمان مشخص می‌تواند وارد ناحیه Critical شود. به عبارتی این ساختار گارانتی می‌کند که آن قسمت از کد که در ناحیه Critical قرار دارد را فقط یکی از نخ‌ها در یک زمان اجرا کند. هنگامی که اجرای آن نخ تمام شد، نخ دیگری شروع به اجرای آن قسمت می‌کند و این روند ادامه میابد. این کار به نوعی موجب اجرای برنامه بصورت سریال می‌شود و در نتیجه شاهد افت عملکرد برنامه خواهیم بود.

دستور اجرای آن در زبان Fortran بصورت زیر است:

```
!$omp critical
```

و دستور اجرای آن در زبان‌های C/C++ بصورت:

```
#pragma omp critical
```

با توجه به قطعه کد شکل ۱۲ در این کد یک حلقه for وجود دارد که در هر تکرار آن کار زیادی توسط تابع big_job توسط نخ‌های مختلف انجام می‌شود. سپس برای جلوگیری از ایجاد شرایط مسابقه، هر نخ وارد ناحیه critical شده و آن قسمت را به ترتیب اجرا می‌کند. به عبارتی نخ‌ها به نوبت تابع consume را اجرا می‌کنند [۹–۱۱].

```
float res;
#pragma omp parallel
{  floatB; int i, id, nthrds;
id = omp_get_thread_num();
nthrds = omp_get_num_threads();
for(i = id; i < niters; i += nthrds){
   B = big_job(i);
#pragma omp critical
   res += consume(B);
}
}
```

شکل ۱۲: کد نمونه ابزار همگام سازی critical در زبان C

```
real:: res
!Somp parallel
Real:: B
Integer::i, id, nthrds
id=omp_get_thread_num()
nthrds=omp_get_num_threads()
DO i=id,(niters-1),(i+nthrds)
   B=big_job(i)
   !Somp critical
   res=res+consume(B)
End do
!Somp end parallel
```

شکل ۱۳: کد نمونه ابزار همگام سازی critical در زبان فرترن

۳.۹ ساختار atomic

ساختارهای مشخصی در سخت افزار وجود دارند که برای بروز رسانی مقادیر در حافظه از آنها استفاده می‌شود. ساختار atomic به این صورت عمل می‌کند که اگر این ساختارهای سخت افزاری در دسترس باشند، از آن‌ها استفاده می‌کند. در غیر اینصورت مانند ساختار critical عمل خواهد کرد. یعنی اجرای یک بلوک از کد فقط توسط یک نخ در یک زمان را ایجاد می‌کند. با این تفاوت که فقط برای بروز رسانی یک محل از حافظه بکار می‌رود [۱۱].

دستور اجرای atomic در زبان Fortran بصورت:

```
!$omp atomic
```

و در زبان‌های C/C++ بصورت:

```
#pragma omp atomic
```

```
#pragma omp parallel
{
   double tmp, B;
   B = DOIT();
   tmp = big_ugly(B);
#pragma omp atomic
   X + = tmp;
}
```

شکل ۱۴: کد نمونه ابزار همگام سازی atomic در زبان C

در قطعه کد شکل بالا برای بروزرسانی x از ساختار atomic استفاده شده است.

ضمنا متغیر x باید از نوع اسکالر باشد.

```
!Somp parallel
Double precision tmp,B
B=DOIT()
tmp=big_ugly(B)
!Somp atomic
X=X+tmp
!Somp end parallel
```

شکل ۱۵: کد نمونه ابزار همگام سازی atomic در زبان فرترن

۴.۹ ساختار Ordered

در این ساختار تکرارها در یک حلقه توسط یک نخ بصورت سریال در یک زمان انجام خواهد شد. زمانی از این ساختار استفاده می‌شود که تکرارهای یک حلقه به یکدیگر وابسته باشند. این ساختار مورد استفاده در حلقه‌های Do/for می‌باشد. ordered نشان می‌دهد که حلقه باید به همان ترتیبی که در حالت سریال اجرا می‌شود، اجرا گردد. البته این به معنای غیرموازی بودن کار نیست. درواقع قرار گرفتن ordered در بدنه for نشان می‌دهد که ما یک بلاک ordered در درون کد برنامه داریم [۱۰، ۱۱].

```
#include
#include
int main();
{
   int i;
#pragma omp parallel
   {
#pragma omp for ordered
      for (i = 0; i < 25; i++){
#pragma omp ordered
         printf("i=%d\n", i);
      }
   }
}
```

شکل ۱۶: کد نمونه ابزار همگام سازی ordered در زبان C

```
integer::i
!Somp parallel
   !Somp do ordered
Do i=1 , 24
   !Somp ordered
   print " i=mod(d,n) "
End do
!Somp end parallel
```

شکل ۱۷: کد نمونه ابزار همگام سازی ordered در زبان فرترن

۵.۹ ساختار Flush

این ساختار در نقطه‌ای از کد که نیاز به ابزار همگام سازی می‌باشد مورد استفاده قرار میگیرد. در نقطه‌ای از کد که این دستور به کار برده می‌شود فقط متغیرهای نقطه مزبور برای نخ‌ها قابل رویت می‌باشند. متغیرهایی که توسط نخ‌ها قابل مشاهده هستند عبارتند از:

۱. تمامی متغیرهای جهانی
۲. آرگومان‌های تکراری
۳. همه اشاره‌گرها
۴. متغیرهای محلی موجود در نقطه مزبور

دستور مربوطه در زبان C بصورت زیر است [۱۱].

```
#pragma omp flush [(variable-list)]
```

در زبان فرترن بصورت زیر است.

```
!$omp flush [(variable-list)]
```

```
            if (tid==0) then
               do j=1,M
                  if(c(j)==9) stop=j
               end do
            end if
            done(tid)=1
            !Somp flush(done)
            do while(done(neigh).eq.0)
            !Somp flush(done)
            end do
         if (tid==1) then
            sum=0
            do j=1,stop-1
               sum=sum+c(j)
            end do
         end if
         !Somp end parallel end
```

شکل ۱۸: کد نمونه ابزار همگام سازی flush در زبان فرترن

```
#include <omp.h>
void main()
{
   omp_lock_t lock;
   int myid;
   omp_init_lock(&lock);
#pragma omp parallel shared(lock) private(myid)
   {
      myid = omp_get_thread_num();
      omp_set_lock(&lock);
      printf("Hello from thread %d\n", myid);
      omp_unset_lock(&lock);
      while (!omp_test_lock(&lock)) {
         skip(myid);
      }
      do_ work(myid);
      omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
}
```

شکل ۱۹: کد نمونه ابزار همگام سازی flush در زبان C

۶.۹ ساختار lock

این ساختار از ابزارهای توابع کتابخانه‌ای lock استفاده می‌کند. این توابع انعطاف پذیری بیشتری برای همگام سازی درمقایسه با ساختارهای critical و atomic دارند. ساختار lock یک حصار حافظه برای همه متغیرهایی که توسط همه نخ‌ها قابل رویت هستند می‌کشد. مجموعه توابع lock عبارتند از:

۱. متغیرهای lock را تعریف می‌کنیم.
۲. استفاده از تابع omp_init_lock آن را مقداردهی اولیه می‌کنیم.

۳. استفاده از تابع omp_set_lock یا omp_test_lock، lock را قرار می‌دهیم.
۴. پس از اتمام کار با استفاده از تابع omp_unset_lock، lock را برمی‌داریم.
۵. پیوستگی lock را با استفاده از تابع omp_destroy_lock از بین برده و حافظه مربوط به lock را آزاد می‌کنیم [۱۱].

۱۰ توزیع کار در OpenMP

توزیع کار در OpenMP یکی از مهم‌ترین ویژگی‌های آن می‌باشد. OpenMP قابلیت تقسیم کار به دو صورت را داراست بصورت دستی و بصورت اتوماتیک (استفاده از متغیرهای محیطی). در این پژوهش به چند تقسیم کار بصورت اتوماتیک خواهیم پرداخت.

۱.۱۰ ساختار Loop work sharing

تکرارهای یک حلقه را بین گروهی از نخ‌ها تقسیم می‌کند. در برنامه Fortran با دستور !$omp do و در زبان‌های C/C++ با دستور #pragma omp for تقسیم وظایف بین نخ‌ها بصورت اتوماتیک ایجاد می‌شود [۳، ۱۰].

```
Program name
 Use omp_Lib
   .
   .
 !$omp parallel Do
Do i=1,N
 Tnew(i)=T(i-1)-T(i)+T(i+1)
End do
 !$omp end parallel Do
   .
   .
End program name
```

شکل ۲۰: ساختار Loop work sharing در برنامه فرترن

اگر برنامه نویس نحوه توزیع تکرارها بین نخ‌ها را مشخص نکند، کامپایلر باید در مورد استراتژی این کار (کدام تکرار به کدام نخ برسد) تصمیم بگیرد. و احتمال ایجاد شرایط مسابقه در این حالت افزایش میابد.

```
#include <stdio.h>
 #include <omp.h>
.
.
#pragma omp parallel
{
#pragma omp for
   for (i = 0; < N; ++){
Tnew(i) = T(i - 1) - T(i) + T(i+1)
   }
}
```

شکل ۲۱: ساختار Loop work sharing در برنامه C

شایان ذکر است می‌توان دستور for و parallel را مانند شکل ۱۰ در یک خط نوشت.

۱.۱.۱۰ ساختار Schedule

در OpenMP یک ساختاری بنام Schedule وجود دارد که به کاربر اجازه می‌دهد که بر روی توزیع تکرارها تاثیر بگذارد. Schedule clause فقط بر روی ساختار حلقه اعمال می‌شود و از آن برای کنترل نحوه توزیع تکرارهای حلقه در نخ‌ها استفاده می‌شود که تاثیر بسزایی بر عملکرد برنامه دارد.

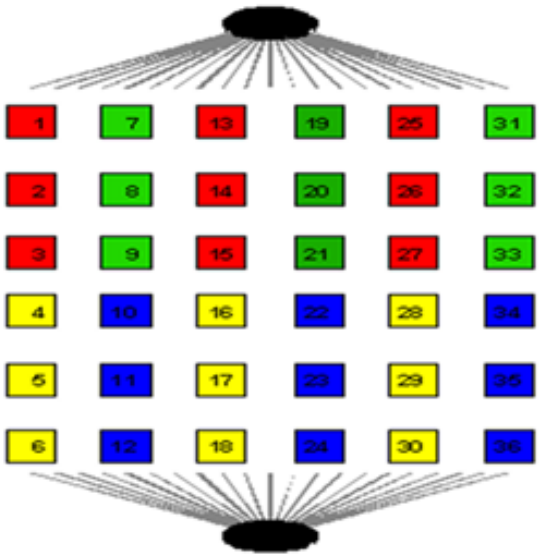
انواع Schedule عبارتند از:

- Static: بلوک‌هایی از تکرارها با سایز مشخص به ترتیب به نخ‌ها اختصاص داده می‌شود.
- Dynamic: هر نخ بلوکی از تکرارها را درون یک صف می‌قاپد تا اینکه همه تکرارها تمام شوند.
- Guided: نخ‌ها بصورت دینامیکی بلوک‌های تکرارها را می‌قاپند. در این حالت سایز بلوک‌ها در شروع بزرگ است و به تدریج کاهش میابد تا به سایز تعیین شده برسد.

نکته قابل توجه این است که از حالت Static زمانی استفاده می‌کنیم که قبل از اجرای برنامه، زمان اجرای هر حلقه قابل پیش بینی باشد. در نتیجه تقسیمات در هنگام کامپایل انجام می‌شود. اما زمانی از حالت dynamic استفاده می‌کنیم که این امر میسر نباشد. در نتیجه توزیع کار در هنگام اجرا انجام می‌گیرد [۲، ۳، ۱۰، ۱۱].

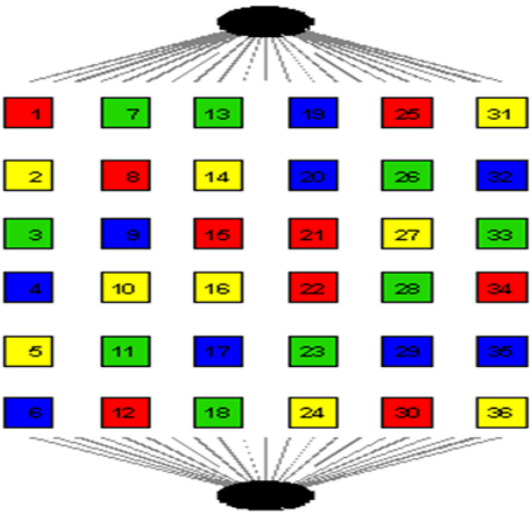
نکته: از حالت static زمانی استفاده می‌کنیم که قبل از اجرای برنامه، زمان اجرای هر قطعه از حلقه قابل پیش بینی باشد. در نتیجه تقسیمات در هنگام کامپایل انجام می‌شود. اما زمانی از حالت dynamic استفاده می‌کنیم که این امر میسر نباشد. در نتیجه توزیع کار در هنگام اجرا انجام می‌گیرد.

```
!Somp parallel do Schedule(Static , 3)
    Do i=1 , 36
       WORK(i)
End do
!Somp end do
```



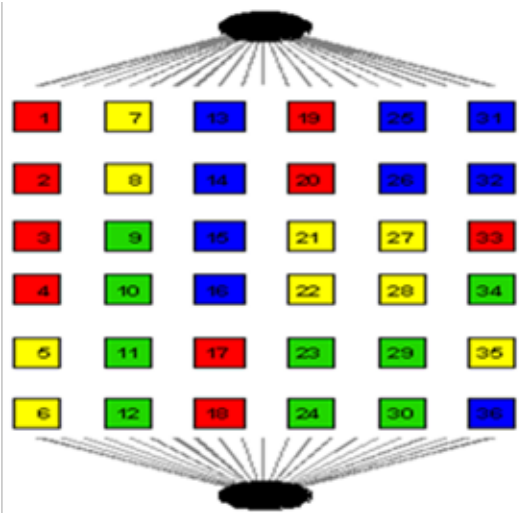
شکل ۲۲: ساختار Schedule (Static)

```
!Somp parallel do Schedule(Dynamic , 1)
    Do i=1 , 36
      WORK(i)
End do
!Somp end do
```



شکل ۲۳: ساختار Schedule (Dynamic)
[۱۱]

```
!Somp parallel do Schedule(guided , 1)
    Do i=1 , 36
      WORK(i)
End do
!Somp end do
```



شکل ۲۴: ساختار Schedule (guided)
[۱۱]

۲.۱.۱۰ ساختار Master

در این ساختار یک بلوک از کد تنها توسط Master Thread اجرا می‌شود و سایر نخ‌ها این قسمت را رد می‌کنند.

```
#pragma omp parallel
   {
        do_many_things();
  #pragma omp master
        { exchange_boundaries();}
 #pragma omp barrier
        do_many_other_things();
   }
```

شکل ۲۵: نمونه‌ای از کد با ساختار master در زبان C

در شکل ۲۳ یک ناحیه موازی وجود دارد که ابتدا هر نخ تابع do_many_things را اجرا می‌کنند. در این قطعه کد قرار است برخی شروط مرزی مسئله قبل اجرای ادامه برنامه جابجا شود. برای این منظور نیاز نمی‌باشد تا همه نخ‌ها این کار را انجام دهند در اینجا فقط Master thread این کار را انجام خواهد داد. از آنجایی که ساختار Master ابزار همگام سازی ندارد، در انتهای آن barrier قرار گرفته تا سایر نخ‌ها در این قسمت متوقف شوند [۱۰، ۱۱].

```
!Somp parallel
  do_many_things()
!Somp Master
  exchange_boundaries()
!Somp barrier
  do_many_other_things
!Somp end parallel
```

شکل ۲۶: نمونه‌ای از کد با ساختار master در زبان فرترن

۳.۱.۱۰ ساختار Single

در این ساختار یک بلوک از کد تنها توسط یک نخ (که لزوما نخ اصلی نیست) اجرا می‌شوند و سایر نخ‌ها این قسمت را رد می‌کنند. در این ساختار اولین نخ که به این قسمت برسد، آن را اجرا می‌کند. از آنجایی که Single یکی از ساختارهای توزیع کار است، بصورت پیش فرض در انتهای خود یک barrier دارد که با استفاده از عبارت nowait میتوان آن را حذف کرد.

این ساختار شبیه به ساختار Master است، با این تفاوت که اولین نخ که به آن قسمت از کد می‌رسد، آن را اجرا میکند (می‌تواند نخ اصلی یا هر نخ دیگری باشد).

```
#pragma omp parallel
 {
    do_many_things();
#pragma omp single
    {exchange_boundaries(); }
    do_many_other_things();
 }
```

شکل ۲۷: نمونه‌ای از کد با ساختار Single در برنامه C

```
!Somp parallel
  do_many_things()
!Somp Single
  exchange_boundaries()
  do_many_other_things
!Somp end parallel
```

شکل ۲۸: نمونه‌ای از کد با ساختار Single در برنامه فرترن

۴.۱.۱۰ ساختار Section

این ساختار، بلوک‌های متفاوتی از کد را به هر نخ اختصاص می‌دهد. به عبارتی این امکان را فراهم می‌سازد که کد را به بخش‌های مختلف تقسیم کرده و هر نخ یکی از این بخش‌ها را اجرا کند.

```
#pragma omp parallel
{
#pragma omp sections
   {
#pragma omp section
      X_calculation();
#pragma omp section
      Y_calculation();
#pragma omp section
      Z_calculation();
   }
}
```

شکل ۲۹: نمونه‌ای از کد با ساختار Section در زبان C

در انتهای این ساختار یک Barrier بصورت ضمنی وجود دارد. و با استفاده از عبارت nowait می‌توان آنرا از بین برد.

```
!Somp parallel

!Somp Sections

!Somp Section
  X_calculation()
!Somp section
   Y_calculation()
!Somp section
   Z_calculation()
!Somp end parallel
```

شکل ۳۰: نمونه‌ای از کد با ساختار Section در زبان فرترن

۱۱ از بین بردن وابستگی

در برخی از حلقه‌های برنامه، متغیرهایی وجود دارد که به علت وابستگی به خود و یا به متغیرهای دیگر امکان موازی سازی را مختل خواهند کرد. در بعضی از برنامه‌ها با تغییر الگوریتم کد نویسی امکان از بین بردن وابستگی وجود دارد و در برخی حالات این امکان وجود ندارد. حال اگر امکان از بین بردن وابستگی با تغییر الگوریتم کدنویسی وجود نداشته باشد در برنامه نویسی موازی به روش OpenMP، اگر متغیر وابسته از نوع اسکالر باشد با دستور Reduction وابستگی داده بصورت اتوماتیک توسط کامپایلر از بین می‌رود. ولی اگر متغیر وابسته از نوع آرایه باشد حتما باید از الگوریتم‌هایی که وابستگی داده را از بین می‌برند، استفاده شود.

فرم کلی دستور Reduction بصورت زیر است:

Reduction(op:list)

که در آن op اپراتور بکار رفته در آن عبارت و list فهرستی از متغیرها است که Reduction روی آن انجام می‌شود [۱۱].

۱.۱۱ نحوه عملکرد Reduction

یک کپی بصورت اختصاصی از متغیرهای داده شده در list گرفته می‌شود و مقداردهی اولیه می‌گردد. برای مثال در اپراتور جمع مقدار اولیه صفر و در اپراتور ضرب مقدار اولیه یک در نظر گرفته می‌شود. و هر نخ متغیر اختصاصی خود را بروز رسانی می‌کند. و در انتها متغیر اختصاصی مربوط به هر نخ با هم ترکیب شده و در داخل متغیر اشتراکی اصلی ذخیره می‌گردد [۱۰، ۱۱].

جدول ۱: مقدار اولیه و اپراتور در برنامه C در دستور Reduction

Operator	Initial Value
&	~ ۰
۱	۰
^	۰
&&	۱
ll	۰

جدول ۲: مقدار اولیه و اپراتور در برنامه Fortran در دستور Reduction

.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	۰
.IOR.	۰
.IAND.	All bits on
.EQV.	.true.

در شکل زیر نمونه‌ای از قطعه کد با وابستگی داده ارائه شده است.

```
double ave = 0.0,  A[MAX] ; int;
 #pragma omp parallel  for reduction(+:ave)

  for (i = 0; i < MAX; i++) {
    ave + = A[i];

  }
ave = ave / MAX;
```

شکل ۳۱: قطعه کد موازی شده با وابستگی داده در برنامه C

متغیر ave دارای وابستگی می‌باشد.

۱۲ موازی سازی کد دو بعدی لاپلاس

در این بخش کد معادله لاپلاس دو بعدی در برنامه Fortran بصورت سریال و موازی در دو حالت Release و Debug اجرا خواهد شد. و تاثیر تعداد نخ‌ها و تعداد نقاط شبکه[1] مورد ارزیابی قرار خواهد گرفت.

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \qquad (1)$$

شرایط اولیه $W_{ij} = 1000$ و شرایط مرزی در شکل نشان داده شده است.
با گسسته سازی معادله لاپلاس برای شبکه مربعی داریم:

$$W_{ij}^{k+1} = 0.25 * \left(U_{i-1,j}^{k} + U_{i+1,j}^{k} + U_{i,j-1}^{k} + U_{i,j+1}^{k}\right) \qquad (2)$$

تمامی مراحل موازی سازی در بخشهای قبل به توضیح آنها پرداخته شد(منظور از W مقدار جدید و U مقدار قدیم است). در برنامه، OpenMP بطور پیش فرض در ناحیه موازی از تمامی نخ‌های سیستم استفاده می‌نماید. لذا برای ایجاد تعداد نخ دلخواه در برنامه می‌توان از دستور زیر استفاده نمود.

NUM_Threads(Number Thread)

```
program main
 use omp_lib
!******************************************************
!                +---------W=0---------+
!                |                     |
!      W=100     |                     | W=100
!                |                     |
!                |                     |
!                +-----------------------+
!                          W=100
!******************************************************
Integer(Kind=4), parameter:: m=250
  Integer(Kind=4), parameter:: n=250
   real ( kind = 8 ) diff,u(m,n),w(m,n),cpuTime
        real ( kind = 8 ) :: eps = 0.0001D+00
          integer ( kind = 4 ) i,j,iterations,iterations_print
         cpuTime=Timef()
do i = 2, m - 1
     w(i,1) = 100.0D+00
  w(i,n) = 100.0D+00
 end do
    do j = 1, n
      w(m,j) = 100
     w(1,j) =  0.0D+00
   end do

 do i = 2, m - 1
   do j=2,n-1
  w(I,j)=1000
  end do
 end do

iterations = 0
   iterations_print = 1
    write ( *, '(a)' ) ' Iteration  Change'
     diff = eps
 do while ( eps <= diff )
     diff = 0
 do j = 1, n
   do i = 1, m
     u(i,j) = w(i,j)
   end do
  end do

!$omp parallel num_Threads (8) shared(u,w)
!$omp do
do j = 2,( n - 1)
   do i = 2, (m - 1)
       w(i,j) = 0.25D+00 * ( u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) )
   end do
  end do
!$omp end do
!$omp end parallel

!$omp parallel num_Threads(8) Reduction(max:diff)
!$omp do
do j = 1, n
   do i = 1, m
       diff = max(diff, abs ( u(i,j) - w(i,j) ))
   end do
  end do
!$omp end do
!$omp end parallel

  iterations = iterations + 1
      if ( iterations == iterations_print ) then
       write ( *, '(2x,i8,2x,g14.6,2x,g9.4)' ) iterations, diff,CpuTime
    iterations_print = 2* iterations_print
  end if
       cpuTime=Timef()
 end do
    write ( *, '(2x,i8,2x,g14.6,2x,G12.4)' ) iterations, diff,cpuTime
end
```

۱۳ نتایج بدست آمده از برنامه لاپلاس

کد معادله دو بعدی لاپلاس در اینتل فرترن در حالت سریال اجرا شده و نتایج زیر بدست آمده است.

جدول ۳: نتایج زمان اجرای برنامه در حالت سریال

Time (s)	Number Mesh	Intel Fortran 2016
۱۱ s	۲۵۰	Release
۱۷۷ s	۲۵۰	Debug

جدول ۴: نتایج زمان اجرای برنامه در حالت موازی بصورت Release

Time (s)	Number Mesh	Number Thread
۱۲/۳ s	۲۵۰	۱
۸/۱ s	۲۵۰	۲
۷ s	۲۵۰	۴
۶/۳ s	۲۵۰	۶
۵ s	۲۵۰	۸

جدول ۵: نتایج زمان اجرای برنامه در حالت موازی بصورت Debug

Time (s)	Number Mesh	Number Thread
۱۷۸/۲ s	۲۵۰	۱
۱۰۴ s	۲۵۰	۲
۹۱/۲ s	۲۵۰	۴
۸۲ s	۲۵۰	۶
۷۶ s	۲۵۰	۸

[1] Grid Point

تمامی حالات برنامه در سیستم CPU core i7 دارای چهار هسته(نخ) واقعی و هشت هسته مجازی اجرا شده است. با توجه به جداول ۴ و ۵ مشاهده می‌شود که اجرای برنامه با یک نخ در حالت موازی عملکرد نامطلوبی نسبت به حالت سریال دارد. به دلیل آنکه در حالت موازی با یک نخ عملکرد برنامه کاملا بصورت سریال خواهد بود و عملکرد نامطلوب به دلیل هزینه فراخوانی دستورات موازی سازی خواهد بود. و همچنین مشاهده می‌شود که عملکرد اجرای برنامه، با افزایش تعداد نخ‌ها از ۴ نخ بیشتر، افزایش چشمگیری نداشته است. علت آن کم حجم بودن محاسبات می‌باشد.

با افزایش تعداد مش‌ها در برنامه نتایج زیر بدست آمده است.

جدول ۶: نتایج زمان اجرای برنامه در حالت سریال

Time (s)	Number Mesh	Intel Fortran 2016
۶۵/۴ s	۴۰۰	Release
۵۸۶ s	۴۰۰	Debug

جدول ۷: نتایج زمان اجرای برنامه در حالت موازی بصورت Release

Time (s)	Number Mesh	Number Thread
۶۷ s	۴۰۰	۱
۳۳ s	۴۰۰	۲
۲۴ s	۴۰۰	۴
۱۷/۱ s	۴۰۰	۶
۱۳/۴ s	۴۰۰	۸

جدول ۸: نتایج زمان اجرای برنامه در حالت موازی بصورت Debug

Time (s)	Number Mesh	Number Thread
۲۸۷/۵ s	۴۰۰	۱
۲۵۷/۷ s	۴۰۰	۲
۲۱۵/۶ s	۴۰۰	۴
۱۸۸ s	۴۰۰	۶
۱۶۸ s	۴۰۰	۸

با توجه به جداول ۷ و ۸ مشاهده می‌شود که با افزایش تعداد مش‌ها عملکرد OpenMP مطلوب‌تر نسبت به جداول ۴ و ۵ بوده است. با افزایش تعداد مش‌ها حجم محاسبات افزایش یافته است. لذا عملکرد پردازش موازی در محاسبات حجیم‌تر مطلوب‌تر خواهد بود.

با توجه به جدول ۴ عملکرد برنامه با هشت نخ در حدود ۲/۲ برابر افزایش یافته است. اما با توجه به جدول ۷ عملکرد برنامه با هشت نخ در حدود ۵ برابر افزایش یافته است. این امر نشان می‌دهد که عملکرد پردازش موازی در محاسبات حجیم‌تر مطلوب‌تر خواهد بود.

۱۴ نتیجه‌گیری

تقریبا تمامی مسائل حاکم بر صنعت و طبیعت از نوع معادلات دیفرانسیل غیرخطی می‌باشند و حل تحلیلی دقیقی برای آنها وجود ندارد و استفاده از وسایل آزمایشگاهی برای تحلیل این نوع معادلات بسیار گران قیمت خواهد بود. تنها راه پیش‌رو حل عددی معادلات می‌باشد. اما با وجود این همه پیشرفت پردازنده‌ها و کامپیوترها، زمان اجرای برنامه‌ها بسیار طولانی خواهد بود. لذا در این پژوهش به بررسی عملکرد اجرای برنامه‌های محاسباتی پرداخته شد. یکی از راهکارهای افزایش کارایی برنامه‌ها استفاده از پردازش موازی می‌باشد که دارای روش‌های متعددی است. در این پژوهش به عملکرد روش OpenMP پرداخته شد. اهمیت مطالب ارائه شده، ایجاد یک روش مطلوب با هزینه و زحمت کمتر برای افزایش کارایی برنامه‌ها خواهد بود. و همچنین دانستن دانش عمیق از دستورات پردازش موازی به الگوریتم طراحی یک کد بهینه موازی کمک بسیار فراوانی خواهد نمود. با اعمال روش OpenMP بر روی برنامه معادله لاپلاس دو بعدی عملکرد اجرای برنامه در دو حالت Release و Debug افزایش یافت. و با افزایش تعداد نخ‌ها در سیستم، عملکرد اجرای برنامه افزایش یافت و مشاهده شد که عملا پردازش موازی بر روی محاسبات حجیم‌تر عملکرد مطلوب‌تری خواهد داشت. همانطور که در مقاله ذکر شد برنامه موازی OpenMP از سیستم حافظه مشترک برای تبادل داده بین پردازنده‌ها استفاده می‌نماید. و در نتیجه به دلیل ایجاد شرایط مسابقه کد موازی شده ایمن نخواهد بود. پیشنهاد می‌شود برای ایمن‌تر شدن کدها از روش‌های دیگر پردازش موازی مانند روش MPI که از سیستم حافظه توزیع یافته بهره می‌برد برای افزایش کارایی برنامه‌ها استفاده شود. یا استفاده از روش OpenCL که ترکیبی از CPU و GPU می‌باشد برای افزایش کارایی کدهای محاسباتی استفاده شود.

مراجع

[1] L.Turner & HonyHU, (2001), “A parallel CFD rotor code using OpenMP”, Advances in Engineering software, 32:665-671

[2] Hoeflinger & et al, (2001), “producing scalable performance with OpenMP :Experiments with two CFD applications”, parallel computing, 27:391-413

[3] D.mininni & et al, (2011), “A hybrid MPI-Open MP scheme for scalable parallel pseudospectral computing for Fluid turbulence”, parallel computing, 37:316-326

[4] Jin & et al, (2011), “High performance computing using MPI and Open MP on Multi-core parallel systems”, parallel computing, 37:562-575

[5] Gorobets & et al, (2011), “Hybrid MPI+Open MP parallelization of on FFT-based 3D passion solver with one periodic direction”, computers & Fluid, 49:101-109

[6] Amritkar & et al, (2012), “openMP parallelism for fluid and fluid-particulate systems”, parallel computing, 38:501-517

[7] Shoukourian & et al, (2013), “Monitoring power Data: A first step towards a unified energy efficiency evalution toolset for HPC data centers”, Enviromental Modelling & software, XXX:1-14

[8] Hassani & et al, (2014), “Improving HPC Application performance inpublic cloud”, International conference on future information Engineering, 10:169-176

[9] Shao & M.faltinsen,(2014), “A harmonic polynomial cell (HPC) method for 3D laplace equation with application in marine hydrodynamics”, Journal of computational physics, 274:312-332

[10] https://computing.llnl.gov/tutorials/openMP/

[11] www.OpenMP.org